Algorithmic Building Blocks for Relationship Analysis over Large Graphs

Stephan Seufert

Dissertation
zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Saarbrücken
2015

Abstract

Over the last decade, large-scale graph datasets with millions of vertices and edges have emerged in many diverse problem domains. Notable examples include online social networks, the Web graph, or knowledge graphs connecting semantically typed entities. An important problem in this setting lies in the analysis of the relationships between the contained vertices, in order to gain insights into the structure and dynamics of the modeled interactions.

In this work, we develop efficient and scalable algorithms for three important problems in relationship analysis and make the following contributions:

- We present the Ferrari index structure to quickly probe a graph for the existence of an (indirect) relationship between two designated query vertices, based on an adaptive compression of the transitive closure of the graph.
- In order to quickly assess the relationship strength for a given pair of vertices as well as computing the corresponding paths, we present the PathSketch index structure for the fast approximation of shortest paths in large graphs. Our work extends a previously proposed prototype in several ways, including efficient index construction, compact index size, and faster query processing.
- We present the Espresso algorithm for characterizing the relationship between two sets of entities in a knowledge graph. This algorithm is based on the identification of important events from the interaction history of the entities of interest. These events are subsequently expanded into coherent subgraphs, corresponding to characteristic topics describing the relationship.

We provide extensive experimental evaluations for each of the methods, demonstrating the efficiency of the individual algorithms as well as their usefulness for facilitating effective analysis of relationships in large graphs.

Kurzfassung

Im Laufe des letzten Jahrzehnts hat die Modellierung von direkt sowie indirekt verknüpften Entitäten in Form von Graphen – wie z. B. von Personen in sozialen Netzwerken oder semantisch annotierten Konzepten in Wissensbasen – eine wichtige Rolle in vielen Geschäfts- und Forschungsfragestellungen eingenommen. Der Umfang der in dieser Form modellierten Daten liegt in vielen Fällen in der Größenordnung von Millionen von Entitäten und Verknüpfungen. Ein wichtiges und vielversprechendes Problemfeld in diesem Bereich ist die effiziente und effektive Analyse der (indirekten) Beziehungen zwischen benutzerspezifizierten Entitäten, um Erkenntnisse über die Struktur und Dynamik der modellierten Interaktionen zu erhalten.

In dieser Arbeit betrachten wir drei fundamentale Fragestellungen in der Analyse von Beziehungen in graphstrukturierten Daten und stellen die folgenden Beiträge vor:

- Erreichbarkeitsanalyse befasst sich mit der effizienten (d. h. in Echtzeit ausführbaren) Überprüfung des Graphen auf die Existenz einer möglichen Beziehung zwischen zwei Entitäten. In dieser Arbeit stellen wir den FERRARI-Algorithmus zur schnellen Verarbeitung dieser Analysevariante vor, basierend auf einer adaptiven Form der Kompression der transitiven Hülle des Graphen.
- Wir stellen die PATHSKETCH Indexstruktur vor, ein System zur schnellen Approximation der Stärke einer Beziehung zwischen zwei Entitäten, sowie der Identifikation eines oder mehrerer zugehörigen Pfade. Das in dieser Arbeit vorgestellte System erweitert einen vorausgehend publizierten Prototyp hinsichtlich effizienter Durchführung des Indexaufbaus, Repräsentation der Indexeinträge sowie schnellerer Anfragebearbeitung.
- Für die semantisch tiefergreifende Analyse von graphstrukturierten Wissensbasen stellen wir den ESPRESSO-Algorithmus zur Charakterisierung der Beziehungen zwischen zwei Mengen von Entitäten vor. ESPRESSO basiert auf der Identifikation wichtiger Ereignisse, welche anschliessend zu dem Lösungskonzept der dichten Teilgraphen erweitert werden. Diese Strukturen entsprechen maßgeblichen Themenbereichen, die zur Beschreibung der wechselseitigen Beziehungen dienen.

Die Effizienz und der praktische Nutzen der genannten Algorithmen und Ansätze werden in umfangreichen Experimenten evaluiert.

Contents

Part I

Introduction

1 Analyzing Relationships at Web-Scale

Personal friendships, interactions between proteins, functional dependencies within software projects – *relationships* among entities such as people, biological molecules, or units of code, appear in various problem domains and in a multitude of application scenarios. Typically, such relationships are modeled in abstract form as (directed or undirected) graphs, a formalism that allows for the expressive as well as intuitive analysis of the modeled relationships. The wide applicability of algorithms defined in abstract terms over such graph structures has fueled tremendous research efforts over recent years, encompassing infrastructure for efficient processing as well as newly developed concepts that allow novel kinds of analysis of graph-structured data. Processing and analyzing massive-scale graphs with billions of entities and relationships has become feasible in several important problem scenarios.

Many of these massive graphs have emerged due to the advent of the World Wide Web, including large online social networks like Facebook and Twitter, interlinked web-pages extracted from the (hyperlink-)web-graph, as well as knowledge graphs, comprising hundreds of millions of facts about entities. The scale and availability of such datasets offers exciting opportunities, both from an academic as well as a commercial viewpoint.

1.1 Challenges and Opportunities

The enormous growth of graph-structured data available for processing, together with the ever-increasing need for more sophisticated forms of analysis, imposes great challenges.

1. While some important problems, such as the computation of PageRank scores over the web graph, can be expressed conveniently and computed efficiently in newly emerged *big data* processing frameworks such as MapReduce (Dean and Ghemawat, 2004) and Pregel/BSP (Malewicz et al., 2010), real-time analysis of large graphs remains a major challenge. As an example, the latency inherent to distributed processing infrastructure renders these solutions infeasible for the interactive exploration of graphs. Usually, low-latency systems rely on centralized processing, which, in turn, either requires high-end hardware or increased engineering efforts. This encompasses, but is not restricted to, the efficient processing of graphs stored on external-memory, requiring novel, I/O-efficient and cache-aware algorithms.
2. The growth observed in graph-structured datasets is not restricted to the structure itself, as measured by the number of vertices and edges, but also includes additional information associated with the entities or relationships. Important examples include labeled graphs with semantically typed edges, such as knowledge graphs representing facts about entities. Enriched graph structures of this kind call for novel solutions to compute semantically more meaningful results, thus extending beyond traditional graph algorithms, which operate on the mere structure.

1.2 Contributions

In this thesis, we address the problem of **relationship analysis** – the processing of graph-structured data with the purpose of gaining insight into the structure and dynamics of the encoded relationships – with respect to both challenges mentioned above.

1. With the goal of enabling real-time analysis of massive-scale graph-structures in the centralized setting, we address the relationship analysis problem from the perspective of *efficiency*. To this end, we devise methods to preprocess graph-structured datasets – similar in spirit to index structures in relational database systems – in such a way that the three most fundamental operations in graph processing, that is, probing the graph for the *(i) existence* as well as the *(ii) strength* and *(iii) structure* of a relationship, can be performed in real-time. The index structures we propose exhibit response times in the order of milliseconds, even on very large instances of graphs.
2. Second, we shift our focus towards devising an *expressive* new paradigm for relationship analysis, based on the annotation of entity-relationship graphs

with additional knowledge and data, such as entity type information or textual descriptions. To this end, we study the problem of *explaining the relationship* between sets of entities in knowledge graphs.

In the next section, we introduce the different contributions we make in this thesis in detail.

1.3 Key Problems in Relationship Analysis

On a high level, this work proposes algorithms and index structures for relationship analysis over large graphs. More specifically, we propose four individual, but related techniques, which we refer to as the *key problems in relationship analysis*. Informally, these components correspond to the questions

Q1: Is there a relationship between two entities?

Q2: How strong is the relationship?

Q3: Who participates in the relationship?

Q4: How can the relationship be characterized?

Technically, these questions correspond to the well-known graph-theoretic concepts of reachability in directed graphs (Q1), distances (Q2) and shortest paths (Q3), as well as a new concept that we develop in the final chapter of this thesis, relatedness cores (Q4). We discuss these individual components in the following sections.

1.3.1 Question 1: « Is there a relationship? »

Given a large graph and a pair of individual vertices, (s, t), or sets of vertices, the problem of graph reachability ($s - t$-connectivity), corresponds to determining whether the graph contains a direct or indirect interaction (in the graph-theoretic terminology referred to as *path*) between the query entities. While this problem is easily solved for undirected graphs (that is, networks that only contain symmetric relationships), the general case of directed graphs is much more challenging. Probing the graph for the existence of a relationship is a very fundamental primitive and has many important practical applications, ranging from discovery of functional dependencies in software projects to the use of reachability queries as a building-block in higher-level graph analysis. The contribution we make in this thesis towards this specific problem lies in the development of an extremely fast, memory-resident index structure that provides a direct control over the query processing vs. precomputation/memory requirement tradeoff. In typical application scenarios, our index structure occupies space much smaller than the size of the input graph, and provides query processing times in the order of microseconds over graphs comprising several millions of vertices and edges. Results of this work, discussed in detail in Chapter 4, has been published in the following research paper:

Stephan Seufert, Avishek Anand, Srikanta J. Bedathur, and Gerhard Weikum. *FERRARI: Flexible and Efficient Reachability Range Assignment for Graph Indexing.* In ICDE'13: Proceedings of the 26th IEEE International Conference on Data Engineering, Brisbane, Australia. Pages 1123-1134. IEEE, 2013.

1.3.2 Question 2: « How strong is the relationship? » Question 3: « Who participates in the relationship? »

The second and third question (of what we have identified as key questions in relationship analysis) correspond to assessing the strength and participants of an (in general indirect) relationship between two or more query entities. The fundamental structure in this context, paths between the query vertices, are sequences of vertices – starting at the source query vertex and ending at the target query vertex – such that every pair of consecutive vertices in the path exhibits a direct interaction in the graph. The most important class of paths between the query vertices are *shortest paths*, that is, paths that contain the smallest possible number of intermediate vertices (for the case of unweighted edges), or the sequence of vertices that exhibit the *strongest* interactions (measured by the sum of reciprocal weights of the constituent edges) for the case of weighted graphs. The contribution we make in this thesis towards this specific problem is an index structure based on the precomputation and materialization of carefully selected paths in the graph, that are combined into an approximate solution to a shortest-path query at runtime. This index structure is especially geared towards very large graph datasets that are potentially too large to hold in main-memory in their entirety. The technique we propose empirically allows for the accurate estimation of the distance (length of the shortest path) between a pair of query vertices while adhering to a user-specified bound on the online query processing cost. A major benefit of our index structure lies in its ability to generate *multiple* short paths connecting the query vertices rather than just estimate their distance. Since our technique offers the user a direct control over the tradeoff between efficiency of query execution and accuracy of the computed estimates, it is possible to use the resulting index structure not only for generating short paths (corresponding to strong relationships) between the query vertices, but also to apply the query processing framework as an algorithmic building block within other graph analysis applications.

Our index structure is based on the prototype presented in the publication

Andrey Gubichev, Stephan Seufert, Srikanta J. Bedathur, and Gerhard Weikum. *Fast and Accurate Estimation of Shortest Paths in Large Graphs.* In CIKM'10: Proceedings of the 18th ACM International Conference on Information and Knowledge Management, Toronto, Canada. Pages 1009-1019. ACM, 2010,

which itself is not a contribution of this thesis. The salient contributions we make in this thesis lie in the complete re-engineering of the proposed prototype, including the integration of previously proposed, more efficient BFS traversal algorithm for index construction, which allows to construct the index significantly faster due

to the representation of several Bfs-forests in main memory at the same time and limiting the amount of random accesses to the graph structure. This modification permits the efficient indexing of graphs comprising billions of edges. Further, we propose a novel physical index layout, including the integration and evaluation of several compression techniques, leading to a large reduction in required disk space for storing the computed index. The basic framework is complemented by a budgeted query processing variant, offering a direct control over the query processing time/accuracy tradeoff.
Finally, we highlight how our index structure can be extended in order to more effectively compute multiple paths between source and target, and to handle additional constraints, such as requiring all paths to pass through a vertex of a specified type. This work is discussed in detail in Chapter 5, and has been submitted for publication in the following research paper:

> Stephan Seufert, Andrey Gubichev, Srikanta J. Bedathur, and Gerhard Weikum. *The Path Sketch Index for Efficient Path Queries over Large Graphs. (under submission)*, 2014.

1.3.3 Question 4: « How can the relationship be characterized? »

In the second part of this thesis, we shift our attention from a purely efficiency-focused point of view towards facilitating a more expressive form of relationship analysis. The main point of study for this work are *knowledge graphs*, which contain vertices corresponding to real-world entities such as organizations (*UNICEF*, *European Union*, etc.) or individuals (*Barack Obama*, *Angela Merkel*, etc.), as well as semantically typed relationships, corresponding for example to the membership of an individual in an organization, interpersonal relationships such as *married to*, etc. Vertices are further assigned semantic types (e. g. *politician*, *country*) which are organized hierarchically in a taxonomic structure. This enrichment (of the mere structural information encoded in the underlying graph) with semantics allows for techniques that compute semantically meaningful answers to certain queries. The specific application scenario we target with our algorithm is the *characterization* or *explanation* of the relationship between two input sets of entities. More specifically, given sets of entities such as *North American politicians* and *European politicians* (that correspond to sets of individual vertices in the knowledge graph), the goal is to extract small subgraphs that represent important *events* (for example political scandals, high-profile political meetings) that involve entities from both query sets. Since the concept we present as a solution to this problem – relatedness cores – is first proposed in this thesis, our contribution is twofold: first, we discuss the computational model underlying our graph-based relationship explanations. Afterwards, we address how this general model can be modified, both in order to achieve scalable computation as well as to compute more insightful answers when additional information such as information about the importance of individual entities over time is available.

Knowledge graphs enriched in this way allow us to define a notion of coherence, based on both features of the individual entities, as well as the correlation of entity popularities over time. The individual contributions have been submitted for publication in the following research papers:

> Stephan Seufert, Klaus Berberich, Srikanta J. Bedathur, Sarath Kumar Kondreddi, Patrick Ernst, and Gerhard Weikum. *ESPRESSO: Explaining Relationships between Sets of Entities. (under submission)*, 2015,
>
> Stephan Seufert, Patrick Ernst, Sarath Kumar Kondreddi, Klaus Berberich, Srikanta J. Bedathur, and Gerhard Weikum. *Instant ESPRESSO: Interactive Analysis of Relationships in Knowledge Graphs. System Demonstration (under submission)*, 2015,

and are discussed in detail in Chapter 6.

1.4 Thesis Organization

This thesis is organized as follows: In Chapter 2, we formally introduce the (mostly graph-theoretic) concepts used throughout the remainder of this work. In Chapter 3, we offer a detailed discussion of the state of the art of the computation infrastructure proposed for processing and querying large-scale graph datasets. Following this introduction, Chapter 4 discusses our first algorithmic building block, the Ferrari index structure for the reachability problem over large directed graphs. Chapter 5 discusses the PathSketch index structure for estimating the distance between a pair of vertices and generating the corresponding short paths in large directed graphs. Chapter 6 presents the Espresso framework for characterizing the relationship between two sets of entities in knowledge graphs. In Chapter 7 we summarize the individual contributions made in this thesis and discuss future directions for research in relationship analysis.

2 Preliminaries

In this chapter, we discuss the basic concepts used throughout the remainder of this thesis. We first introduce the fundamental graph-theoretic terminology and establish the necessary notation used in the subsequent chapters.

2.1 A Primer on Graph Theory

As outlined in the previous chapter, the term *graph-structured data* broadly refers to collections of interconnected objects, for example a set of web pages together with their associated hyperlinks, that induce relationships between pairs of pages. Formally, the term *graph* is defined as follows:

Definition 2.1 (Graph). *A (directed)* ***graph*** *is a triple $G = (V, E, w)$ with a set of vertices/nodes V, a set $E \subseteq V \times V$ of edges, and a weight function $w : E \to \mathbb{R}$. In the special case of unit edge weights, that is, $w(e) = 1$ for all $e \in E$, the graph G is called* ***unweighted graph****.*

This formalism imposes a clear distinction on the source and target of the relationship represented by the edge (s, t). Continuing above example, the source s of the edge corresponds to the page containing the hyperlink, whereas the target t refers to the page being pointed to.

In many scenarios, the relationships expressed by the edges are symmetric, and no direction can be associated with the edge. Examples include classical social networks (where edges represent friendship), co-occurrence graphs (e. g. products that have been purchased together), and protein-protein interaction networks in computational biology. Conceptually, such **undirected graphs** can be regarded as the special case of a directed graph where the set of edges E contains for each edge

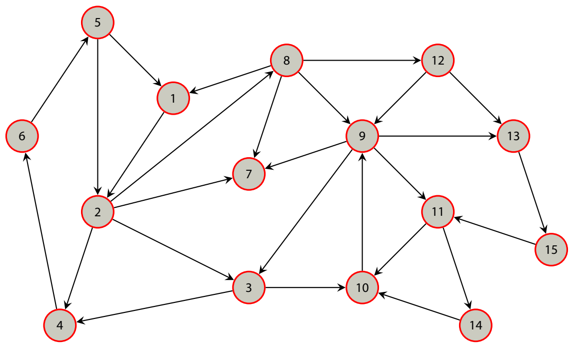


Figure 2.1: Directed Graph

(s,t) the corresponding reverse edge (t,s). Due to their importance, this class of graphs is commonly represented by a dedicated formalism where we denote an individual edge as a set of vertices $\{s,t\}$, rather than as a pair of tuples $(s,t),(t,s)$. However, if there is no risk for confusion we will denote an undirected edge $\{s,t\}$ by the tuple (s,t) or (equivalently) by (t,s), for simplicity of notation. In this work, we restrict our focus to simple graphs, that is, we assume that the edge set does not contain loops, i. e. $(s,t) \in E \Rightarrow s \neq t$ and every pair of vertices s,t is connected through at most one edge $(s,t) \in E$.

In **labeled** graphs, every vertex and edge is annotated with one or more *labels* from a specified label space Σ_V, Σ_E:

Definition 2.2 (Labeled Graph). *A **labeled graph** is specified by a tuple*

$$G = (V, E, \ell_V, \Sigma_V, \ell_E, \Sigma_E)$$

with a set of vertices/nodes V, a set $E \subseteq V \times V$ of edges, a set of vertex labels Σ_V, a vertex-labeling function $\ell_V : V \to 2^{\Sigma_V}$, a set of edge labels Σ_E, and an edge-labeling function $\ell_E : E \to 2^{\Sigma_E}$.

As an example, for a graph modeling the aviation network of an airline, vertices correspond to airports and are connected with directed edges correspond to flight connections. In this case, a vertex v would be labeled with the airport code, i. e. $\ell_V(v) = \{FRA\}$ and an edge would be labeled with the codes of the corresponding flights, e. g. $\ell_E(e) = \{UA090, LH120\}$, etc.

For a vertex $v \in V$, we denote the **successors** (outgoing neighbors) and **predecessors** (incoming neighbors) of v by the sets

$$\begin{aligned} &\mathcal{N}^+(v) = \{v' \in V \mid (v,v') \in E\} \\ \text{and} \quad &\mathcal{N}^-(v) = \{v' \in V \mid (v',v) \in E\}, \end{aligned} \tag{2.1}$$

that is, by the sets of vertices with an outgoing edge ending at vertex v, and the vertices with an incoming edge originating from v, respectively. Closely related to this notion, we define the **out**- and **in-degree** of a node as the number of successors and predecessors, respectively, denoted by

$$\delta^+(v) = |\mathcal{N}^+(v)| \quad \text{and} \quad \delta^-(v) = |\mathcal{N}^-(v)|. \tag{2.2}$$

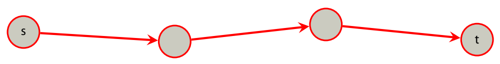


Figure 2.2: Path of length 3

Finally, the **degree** of v is given by $\delta(v) = \delta^+(v) + \delta^-(v)$. For the case of undirected graphs, no distinction is made between incoming and outgoing neighbors. In this case it thus holds $\delta(v) = \delta^+(v) = \delta^-(v)$.

Connectivity

The notion of **paths** generalizes the concept of an individual edge in order to capture indirect connections between vertices. Formally, a (directed) path from vertex s (source) to vertex t (target) is given by a sequence

$$p_{s\to t} = (s = v_0, v_1, \ldots, v_l = t), v_i \in V,$$

with $(v_{i-1}, v_i) \in E, 1 \leq i \leq l$.
The **length** $\|p_{s\to t}\|$ of this path is specified by the sum of weights of the edges between the adjacent vertices in the path, i. e. – by abuse of notation – we have

$$\|p_{s\to t}\| = \sum_{i=1}^{l} w(v_{i-1}, v_i), \tag{2.3}$$

We denote the set of all paths originating at vertex s and ending at vertex t by $\mathcal{P}_{s\to t}$.
The **distance** between the pair of vertices (s,t) is defined as the length of the shortest path connecting s and t:

Definition 2.3 (Distance). *Given a graph $G = (V, E, w)$ and a pair of vertices $(s,t) \in V \times V$, the distance between s and t is given by*

$$d(s,t) = \begin{cases} \min_{p\in\mathcal{P}_{s\to t}} \|p\| & \text{if } \mathcal{P}_{s\to t} \neq \emptyset \\ \infty & \text{otherwise.} \end{cases} \tag{2.4}$$

In general, the distance function is not symmetric, i. e. it holds $d(s,t) \neq d(t,s)$. Paths providing the “best” connection between a pair of nodes are of special interest:

Definition 2.4 (Shortest Path). *Given a graph $G = (V, E, w)$ and a pair of vertices $(s,t) \in V \times V$, we define the* shortest paths *from s to t as*

$$\textsc{Sp}(s,t) = \arg\min_{p\in\mathcal{P}_{s\to t}} \|p\| = \{p \in \mathcal{P}_{s\to t} \mid \|p\| = d(s,t)\}, \tag{2.5}$$

that is, the set of paths starting at s and ending at t that have minimal length.

One of the most important metrics for characterizing graphs – the notion of the **diameter** – is derived from vertex distances:

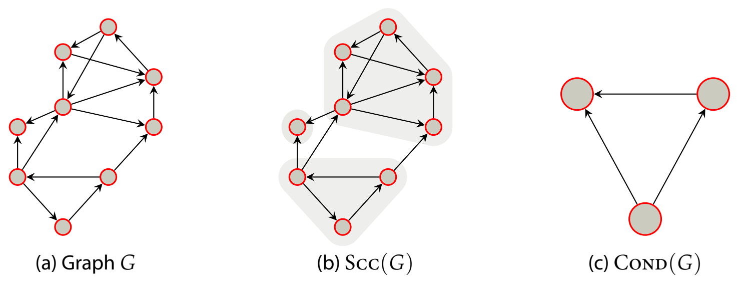


Figure 2.3: Connected Components

Definition 2.5 (Diameter). *Given a graph $G = (V, E)$, the* diameter *of G, denoted by* $\operatorname{diam}(G)$*, corresponds to the maximum distance among all pairs of vertices, i. e. the length of the longest shortest path in the graph:*

$$\operatorname{diam}(G) = \max_{s,t \in V} d(s, t). \tag{2.6}$$

Connected Components

As Equation (2.4) indicates, not all pairs of vertices (s, t) might be connected via a path, in which case it holds that $P_{s \to t} = \emptyset$. A graph that does not provide a directed path between all pairs of vertices is called disconnected. In order to formalize the notion of **connectivity** in a graph, we define the reachability relation:

Definition 2.6 (Reachability). *For a graph $G = (V, E)$ and a pair of vertices, (s, t), we call t* reachable from s*, denoted by $s \sim t$, if G contains a directed path originating from s and ending at t:*

$$s \sim t \iff \mathcal{P}_{s \to t} \neq \emptyset. \tag{2.7}$$

Further, we denote the set of vertices reachable from $v \in V$ in G by

$$\mathcal{R}_G(v) = \{w \in V \mid v \sim w\}. \tag{2.8}$$

The set $\mathcal{R}_G(v)$ is called the *reachable set* of v. When the context is clear, we will drop the subscript. Note that the notion of reachability is reflexive and transitive an thus induces a partial order on the set of vertices. A graph G is strongly connected if the reachability relation is symmetric in G:

Definition 2.7 (Strongly Connected Components). *For a graph $G = (V, E)$, we define the* strongly connected components *of G as the* maximal *sets of vertices S_1,S_2, . . .,$S_N \subseteq 2^V$, $1 \leq N \leq |V|$, such that for every set $S_i, 1 \leq i \leq N$, the included vertices are mutually reachable, i. e.*

$$\forall (s, t \in S_i) : s \sim t. \tag{2.9}$$

We denote the set of strongly connected components of a graph G by $\textsc{Scc}(G)$ and, for a vertex $v \in V$, we denote the strongly connected component containing v as $[v]$. Finally, we define the **condensed graph** of G:

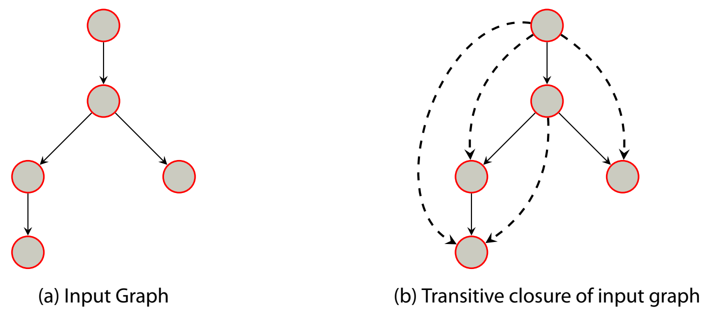

(a) Input Graph (b) Transitive closure of input graph

Figure 2.5: Transitive Closure

Definition 2.8 (Condensed Graph). *Let $G = (V, E, w)$ denote a directed graph. We define the* condensed graph *of G as*

$$\textsc{Cond}(G) = (V_c, E_c) \quad \textit{with} \quad V_c = \textsc{Scc}(G) \tag{2.10}$$

$$\textit{and} \quad E_c = \{([u], [v]) \mid (u, v) \in E\}, \tag{2.11}$$

that is, the graph containing a vertex (called supervertex) for every strongly connected component of G, where two supervertices $[u] \neq [v]$ are connected via a directed edge if there exists an edge from a vertex $u \in [u]$ to a vertex $v \in [v]$.

An illustration of the concepts introduced in this section is shown in example in Figure 2.3.

Transitive Closure

The concept of the **transitive closure** of a graph is closely related to reachability (and plays an important role in our work on index support for reachability queries in Chapter 4). Conceptually, given a graph $G = (V, E)$, the transitive closure of G – denoted by $\textsc{Tc}(G)$ – is the minimal superset of the edge set E, that contains an edge for every pair of distinct vertices (s, t) reachable in the input graph:

Definition 2.9 (Transitive Closure). *Let $G = (V, E)$ denote a graph. We define the* transitive closure *of G as*

$$\textsc{Tc}(G) = \{(s, t) \in V \times V \mid s \sim_G t, s \neq t\}. \tag{2.12}$$

An example is depicted in Figure 2.5. Here, the input graph is shown on the left. The transitive closure of the input graph, shown in the right figure, contains the edges of the input graph as well as additional edges (dashed).

Cyclic Structures

Cyclic structures are a concept frequently occurring in graph theory:

Definition 2.10 (Cycle). *Let $G = (V, E)$ denote a directed graph. A subgraph $C = (V_C, E_C), V_C \subseteq V, E_C \subseteq E$ of G is called (simple)* ***cycle*** *if it holds*

$$\forall (v \in V_C) : \delta(v) = 2, \tag{2.13}$$

where $\delta(v)$ denotes the degree of v. Thus, each vertex contained in C has exactly two neighbors in the graph C.

The special case of directed cycles is of particular importance:

Definition 2.11 (Directed Cycle). *Let $G = (V, E)$ denote a directed graph. A subgraph $C = (V_C, E_C), V_C \subseteq V, E_C \subseteq E$ of G is called* ***directed cycle*** *if it holds*

$$\forall (v \in V_C) : \delta^+(v) = \delta^-(v) = 1, \tag{2.14}$$

that is, for every vertex in $v \in C$, there exists exactly one incoming and outgoing edge in E_C, respectively.

Thus, a directed cycle corresponds to a path in a directed graph starting and ending at the same vertex.

Directed Acyclic Graphs (DAGs) and Trees

Graphs that do not contain directed cycles play an important role in many applications:

Definition 2.12 (Directed Acyclic Graph). *Let $G = (V, E, w)$ denote a directed graph. We call G* directed acyclic graph (DAG) *if G does not contain directed cycles.*

Formally, for DAGs it holds

$$\forall (u, v \in V) : u \sim_G v \wedge v \sim_G u \implies u = v. \tag{2.15}$$

A special case of directed acyclic graphs, **trees**, are among the most important classes of graphs, with a manifold of applications. For the case of undirected graphs, trees are defined as connected, cycle-free graphs. If the edges are directed, we define:

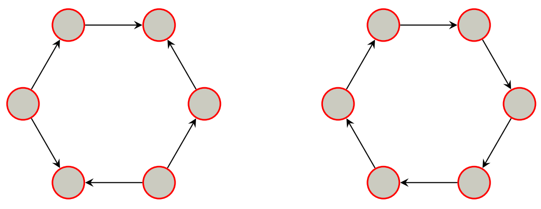
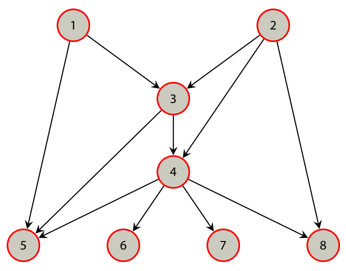
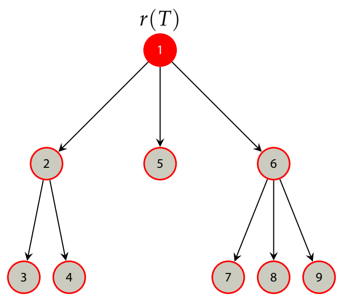

(a) Cycle (b) Directed Cycle

Figure 2.6: Cyclic Structures



(a) Directed, Acyclic Graph



(b) Directed, Rooted Tree

Figure 2.7: Acyclic Graphs

Definition 2.13 (Directed Rooted Tree). *Let $T = (V_T, E_T, w)$ denote a directed graph. T is called* directed, rooted tree, *if G is cycle-free and there exists a vertex $r(T) \in V_T$, such that*

$$\forall (v \in V_T) : \; r(T) \sim_T v, \tag{2.16}$$

that is, all vertices in the tree are reachable from r.

We refer to the vertex $r(T)$ as **root** of the tree T. We depict examples for both types of graphs in Figure 2.7.
For the case of undirected edges, a tree is a graph with exactly one path between each pair of vertices.

Assessing Vertex Importance

Many graphs encountered in present-day applications exhibit certain noteworthy properties. One of the most important classes of graph are the so-called **scale-free graphs** (Barabási and Réka, 1999). Prominent examples include social networks and the web graph. In a scale-free graph, the fraction of vertices with degree k is proportional to a value $k^{-\gamma}$ for some constant $\gamma > 0$ (Bollobás and Riordan, 2004), i. e. it holds

$$\text{Prob}\left[\delta(v) = k\right] \; \propto \; k^{-\gamma}. \tag{2.17}$$

The highly skewed degree distribution commonly found in scale-free graphs suggests that we can identify *important* vertices of high degree, so-called *hubs*. We formalize this notion of vertex importance, based on three different concepts: degree centrality, closeness centrality, and betweenness centrality.

Definition 2.14 (Degree Centrality). *Given a graph $G = (V, E)$, we define the* degree centrality *of a vertex $v \in V$ as*

$$c_D(v) = \delta(v). \tag{2.18}$$

Apart from this rather straightforward measure, a commonly used approach for evaluating vertex importance is based on the distance of a vertex to the remaining nodes in the graph:

Definition 2.15 (Closeness Centrality). *Given a graph $G = (V, E)$, we define the* closeness centrality *of a vertex v as*

$$c_C(v) = \sum_{w \in V \setminus \{v\}} 2^{-d(v)w}. \tag{2.19}$$

The third measure, betweenness centrality, evaluates the score of an individual vertex $v \in V$ as the fraction of shortest paths between pairs of vertices $(u, w) \in V^2$ that pass through the intermediate vertex v:

Definition 2.16 (Betweenness Centrality). *Given a graph $G = (V, E)$, we define the* betweenness centrality *of v as*

$$c_B(v) = \sum_{u \in V \setminus \{v\}} \sum_{w \in V \setminus \{u,v\}} \frac{sp(u,w)}{sp(u,v,w)}, \tag{2.20}$$

where $sp(u, w)$ denotes the number of shortest paths between u and w and $sp(u, v, w)$ denotes the number of such paths that contain vertex v, respectively.

GRAPH ALGORITHMS

In this section we review fundamental algorithms on graphs, that will play an important role in later chapters of this work.

Search Algorithms

The two most fundamental algorithms over graphs – **Breadth-First Search (BFS)** and **Depth-First Search (DFS)** – can be categorized as *search* or *exploration algorithms*. In this setting, given a start node $v \in V$, the surroundings of v are explored in a recursive manner, the difference between the two algorithms being the way in which the next vertex for expansion is selected. The general procedure underlying BFS and DFS is outlined in Algorithm 1.

In this algorithm, the actual "work" is done in the task-specific visit function called in line 7. As an example, an algorithm for checking reachability (i. e. the existence of a directed path from vertex s to a vertex $t \in V$) would employ a visit function that terminates with a positive answer if it is invoked on the target vertex t. As a result, the graph contains a path from s to t if and only if the GRAPHSEARCH algorithm execution was terminated by the respective visit function with a positive answer. The fundamental difference between Breadth-First and Depth-First Search lies in the way in which the next vertex is selected for expansion, or, in other words, the type of queue used in the algorithm. For BFS, a FIFO (*first-in-first-out*) queue is used. Thus, the vertices that are added to the queue in line 10 of the algorithm are added to the *end* of Q. Consequently, the order in which vertices are explored corresponds to their distance to the start vertex s. More precisely, let $u, v \in V$ denote two vertices with the property that v is explored *after* u is explored. For the respective distances to the start vertex s it will hold

$$d(s, u) \leq d(s, v).$$

Algorithm 1: GraphSearch(G, s)

```
Input: graph $G = (V, E)$, vertex $s \in V$
begin
    $Q \leftarrow$ MakeQueue()
    Push$(Q, s)$
    $S \leftarrow \{s\}$
    while $Q \neq \emptyset$ do
        $v \leftarrow$ Dequeue$(Q)$
        Visit$(v)$
        for $w \in \mathcal{N}^+(v)$ do
            if $w \notin S$ then
                Enqueue$(Q, w)$
                $S \leftarrow S \cup \{w\}$
```

In contrast, the Depth-First Search algorithm uses a LIFO (*last-in-first-out*) queue which makes it more intuitive to express the procedure in a recursive manner. As the name suggests, the exploration proceeds by exploring the *deepest* unexplored vertex (in terms of distance from the start node).
Important problems that can be solved with this kind of search algorithm include computation of shortest paths in unweighted graphs (Bfs), checking reachability (Bfs, Dfs), computing a topological ordering (Dfs), identifying (strongly) connected components (Dfs) and many more. In terms of computational complexity, the basic Bfs and Dfs algorithms exhibit runtime linear in the size of the graph (in terms of number of vertices and edges), i. e. with time complexity of $O(m+n)$ and space complexity of $O(n)$, where $n = |V|$ and $m = |E|$.

Dijkstra's Algorithm

Another very prominent technique, **Dijkstra's algorithm**, is closely related to the above search algorithms. This algorithm, displayed in pseudocode in Algorithm 2, is primarily used for the computation of shortest paths and distances in weighted (directed or undirected) graphs. The input to the algorithm is the weighted graph and a vertex s, the output of the algorithm is a vector of the distances (Distance) from s to all other vertices as well as for each vertex v the last vertex Parent$[v]$ in one of the shortest paths from s to v. A fundamental property of this algorithm is, that for the vertex identified in line 9 of the algorithm (the vertex v in the queue with minimal distance value Distance$[v]$), it holds

$$\text{Distance}[v] = d(s, v).$$

that is the distance value assigned to v at this point during the execution of the algorithm corresponds to the actual distance from the start vertex s. Consequently, v is added to the set S and called *settled node*, signifying that the true distance from the start node s is known. In contrast, for the remaining vertices in the queue, the

Algorithm 2: Dijkstra(G, s)

```
Input: weighted graph $G = (V, E, w)$, vertex $s \in V$
Result: DISTANCE$[v \in V]$: vector of distances from $s$ to the vertices in $V$,
        PARENT$[v \in V]$: vector containing penultimate vertex in shortest path
        from $s$ to vertex $v$
1  begin
2      foreach $v \in V$ do
3          DISTANCE$[v] \leftarrow \infty$
4          PARENT$[v] \leftarrow v$
5      DISTANCE$[s] = 0$
6      $Q \leftarrow \{s\}$
7      $S \leftarrow \emptyset$
8      while $Q \neq \emptyset$ do
9          $v \leftarrow \arg\min_{v \in Q}$ DISTANCE$(v)$
10         $S \leftarrow S \cup \{v\}$
11         for $w \in \mathcal{N}^+(v)$ do
12             if $w \notin Q$ then
13                 $Q \leftarrow Q \cup \{w\}$
14             $d \leftarrow$ DISTANCE$[v] + w(v, w)$
15             if $d <$ DISTANCE$[w]$ then
16                 DISTANCE$[w] \leftarrow d$
17                 PARENT$[w] \leftarrow w$
18 return DISTANCE$[v \in V]$, PARENT$[v \in V]$
```

currently assigned DISTANCE$[w], w \in Q$ can further decrease over the course of the algorithm. In practical implementations, the data structure used as vertex queue is a heap, where the use of Fibonacci heaps results in the best known computational complexity for Dijkstra's algorithm of $O(m + n\log(n))$, $n = |V|, m = |E|$. As a result, for the case of Fibonacci heaps, identifying and removing from the queue the vertex to mark as settled (line 9) can be achieved in (amortized) logarithmic time in the size of the queue, accounting for the logarithmic factor in the overall runtime complexity. An update to the distance value assigned to a vertex (line 16) requires a DECREASEKEY operation over the heap, which can be achieved in constant time. This operation decreases the tentative distance from the source s to the vertex w in the priority queue Q, which provides access to the closest unsettled vertex. The DECREASEKEY operation potentially causes a reorganization of the heap data structure underlying this priority queue. As mentioned above, the main use-case of Dijkstra algorithm lies in the computation of shortest paths and distances. While the latter can be directly inferred from the assigned distance value, the actual shortest path from s to a prescribed node t can be reconstructed by following the PARENT pointers backwards to the root.

Random Walks with Restarts

Algorithms based on random walks with restarts (RWR) represent an important and widely used class of graph algorithms. Typically, this type of algorithm is executed over edge-weighted, directed or undirected graphs with the goal of **ranking** the vertices based on a certain criterion. The most prominent example of an RWR-algorithm is the PageRank algorithm (Brin and Page, 1998), used to determine the relative importance of a web page for use in search result ranking. Random-walk based algorithms are traditionally expressed in linear algebra notation, hence we switch our graph representation to matrix form. In this setting, a weighted graph $G = (V, E, w)$ is represented by the (weighted) adjacency matrix

$$W \in \mathbb{R}^{n \times n},\ w_{ij} = \begin{cases} w(v_i, v_j) & \text{if } (v_i, v_j) \in E, \\ 0 & \text{otherwise,} \end{cases} \tag{2.21}$$

where $w(v_i, v_j)$ denotes the weight of the edge (v_i, v_j). For the case of undirected graphs, W is a symmetric matrix, i. e. it holds $W^T = W$. Based on the adjacency matrix W, a random walk algorithm defines a transition matrix $T \in \mathbb{R}^{n \times n}$ with stochastic column vectors, i. e.

$$\sum_{i=1}^{n} t_{ij} = 1 \quad \text{for} \quad 1 \leq i \leq n, \tag{2.22}$$

and $t_{ij} \leq 1$ for all $1 \leq i, j \leq n$, which can be regarded as the column-normalized adjacency matrix W. If we consider the web graph, where web pages correspond to vertices and hyperlinks to edges, the intuition behind the transition matrix is to capture the probabilities that a random surfer located at the web page represented by vertex $v \in V$ proceeds to the page corresponding to vertex $w \in V$ after following a hyperlink. As a second input to the algorithm, the initial probability for the surfer to start at vertex v is given by the respective entry in vector $\mathbf{x}_0 \in \mathbb{R}^{n \times 1}$, often corresponding to the uniform distribution, i. e. $\mathbf{x}_0[v] = n^{-1}$, but other choices are appropriate in certain settings in order to bias the random walk process. The concept of start probabilities is extended to a probability distribution $\mathbf{x}_t \in \mathbb{R}^{n \times n}$, where $\mathbf{x}_t[v]$ denotes the probability that the random surfer is located at vertex v at time t in the process. This probability distribution is computed recursively following the recurrence

$$\mathbf{x}_{i+1} = (1 - \alpha)\mathbf{x}_0 + \alpha T \mathbf{x}_i, \tag{2.23}$$

where $(1 - \alpha) \in [0, 1]$ denotes the restart probability, that is, the probability that the surfer restarts the random walk process at a node $w \in V$, selected proportional to its start probability $\mathbf{x}_0[w]$. The desired output of the RWR algorithm is the final assignment of scores signifying relative importance of the vertices, given by vector $\mathbf{x}_F \in \mathbb{R}^{n \times 1}$. This vector corresponds to the fixed point of Equation (2.23), that is the stationary probability distribution $\mathbf{x}_F$ where we have $\mathbf{x}_F = \mathbf{x}_{F-1}$. Regarding the computation of the final vector $\mathbf{x}_F$, the standard way is to iterate Equation (2.23) until convergence, known as **Jacobi power iteration** method.

2.1.1 Important Graph Classes

In this section, we discuss several important *classes* of graphs from different application domains, that will be referred to from later chapters.

Social Networks

Social networks had been subject of research in the social sciences long before popular online social networking sites such as Facebook or LinkedIn were established. Vertices in a (pure) social network graph correspond to *persons* that are interconnected via typed or untyped relationships e. g. based on interpersonal friendship, business relationships, and the like. While most social networks contain undirected relationships, there are examples for directed social network graphs, for example (subscription-based) content-generating sites like the microblogging service Twitter. From a graph-theoretic point of view, social networks exhibit several noteworthy properties. A prominent result, the theory of *six degrees of separation* states that, in the modern society, each pair of persons is connected via a chain of at most five intermediary contacts (Travers and Milgram, 1969). Over the last decade, the massive growth of online social networks has permitted to perform in-depth analysis of these graph structures at large scale. Using crawls of several social networking sites, Mislove et al. (2007) confirm several properties attributed to social networks, such as small diameter, power-law degree distribution and the scale-free property.

Citation Graphs

Citation networks represent another class of graphs with distinctive properties. In this setting, vertices correspond to articles (e. g. scientific papers or patent applications). A citation graph contains a *directed* edge from article X to article Y, if Y is *referenced* in article X. Naturally, since only articles that have been published in the past can be referenced, citation networks are by definition acyclic, directed graphs, a property that renders this class of graphs particularly interesting for reachability analysis.

Road Networks

Due to their important role in route planning and associated applications, road networks (transportation networks) are arguably one of the most widely studied classes of graphs. In this setting, vertices typically correspond to street crossings, which in turn represent the directed or undirected, weighted edges. Algorithms over road networks, most notably for shortest path computation, are at the core of modern GPS-based navigational systems. The resource limitations of mobile GPS devices have led to a large body of research in algorithm engineering as well as theoretical studies, with the goal of identifying special properties of road networks that can be exploited for fast processing of shortest path queries. Such properties include hierarchical structure (Geisberger et al., 2008, Sanders and Schultes, 2005), enabling query processing techniques such as transit-node routing (Bast et al., 2007).

Knowledge Graphs

Knowledge graphs, a term made popular by the recently revealed knowledge base behind the Google search engine, refer to collections of entities such as persons, organizations, etc. that are interconnected via semantically typed edges that encode facts about the respective incident entities, e. g. personal relationships, membership in organizations, and the like. Knowledge graphs – such as the YAGO2 ontology (Hoffart et al., 2013) – play an increasingly important role in many applications, ranging from named entity disambiguation (Hoffart et al., 2011) to query understanding (Pound et al., 2012) and relationship explanation (Fang et al., 2011), to name a few.

Web Graphs

We have discussed web graphs earlier in this chapter during the discussion of PageRank algorithm. In web graphs, vertices correspond to web pages (or URLs) and directed edges correspond to hyperlinks, thereby capturing the structure induced by the links embedded into websites. This graph structure underlying the World Wide Web represents an important resource for many applications. While as of 2008 the number of web-pages was estimated to more than a trillion[1], important characteristics of the web graph have been derived from smaller, crawled portions of the web. In their seminal work, Broder et al. (2000) analyze several fundamental properties of web graph, with supporting evidence for power-law degree distribution. Further, the macroscopic structure of the web is detailed, comprising a strongly connected core, a portion of pages linking into and out of the core, respectively, and so-called *tendrils*, pages disconnected from the SCC. All four sets were found to be of roughly the same size. In this thesis, we consider web graphs both for reachability analysis as well as for the computation of shortest paths.

With this, we conclude our brief introduction to graph theory. For a more comprehensive overview of graph-theoretic concepts, the reader is referred to the classical literature surveying the field (Diestel, 2010), an interdisciplinary perspective is given by Easley and Kleinberg (2010). Fundamental graph algorithms are covered by Cormen et al. (2009).

2.2 Approximation Algorithms

In the later chapters of this work, we will repeatedly encounter the notion of **approximation algorithms**. Typically proposed for NP-hard optimization problems, this class of algorithms offers a tradeoff between the optimality of the returned solution and the time- and/or space-complexity of the underlying algorithm. In concordance with the terminology proposed by Vazirani (2001), we formally define:

Definition 2.17 (**α-Approximation Algorithm**). *Let Π denote a minimization problem and I a problem instance of Π, respectively. We define the cost of the optimal solution of I – as determined by the objective function specific to Π – by OPT_I. An*

[1] `http://googleblog.blogspot.de/2008/07/we-knew-web-was-big.html`

*algorithm A is called α-**approximation algorithm** for Π if for every problem instance I it holds*

$$c\big(A(I)\big) \leq \alpha \cdot \mathsf{OPT}_I, \ \alpha \geq 1 \tag{2.24}$$

where $A(I)$ denotes the solution returned by the algorithm A for the problem instance I and c is to the objective function quantifying the cost of the solution.

While we used a minimization problem in above definition, the principle carries over to the class of maximization problems in a straightforward way.

3 Graph Data Processing: State of the Art & Directions

In this chapter, we discuss the state of the art in processing large-scale graph-structured data. In concordance with the terminology proposed by Robinson et al. (2013), we make a distinction between systems for online processing – e. g. querying – of graphs (graph databases), and offline analytic processing.

3.1 Representing, Indexing, and Querying Graph-Structured Data

Graph-structured data can be expressed in many ways. We briefly review the three fundamental representations that are commonly used in practice.

- In the simplest case, a graph can be represented by simply recording all edges in the form of a list of source-target pairs. This **edge list** representation is particularly well-suited for graph algorithms in the (semi-)streaming model (Feigenbaum et al., 2004) of computation.
- For the concise description of algorithms, it is sometimes convenient to express a graph in terms of matrix notation. In this setting, an unweighted graph $G = (V, E)$ is given by the **adjacency matrix** $A \in \mathbb{R}^{n \times n}$ where $A_{ij} = 1$ if and only if the graph contains an edge (v_i, v_j), and zero otherwise. For weighted graphs, we have $W \in \mathbb{R}^{n \times n}$ with $W_{ij} = w(v_i, v_j)$ if $(v_i, v_j) \in E$, where $w(v_i, v_j)$ denotes the weight of the edge from vertex v_i to vertex v_j. Certain algorithms (e. g. the PageRank algorithm discussed in Section 2.1) can be expressed in a convenient manner using this representation.

- The most prevalent graph representation used in practice are **adjacency lists**. In this setting, every vertex v of the graph is directly assigned the list of successors, i. e. the vertices in $\mathcal{N}^{+}(v)$ (cf. page 12). In some settings, it makes sense to access the incoming edges of a vertex directly, in which case we also assign a list containing the vertices $\mathcal{N}^{-}(v)$, with an outgoing edge pointing to v.

In the next sections, we present an overview of different paradigms for querying graph-structured data, starting with the simplest scenario: structural querying (i. e. based purely on graph topology).

3.1.1 Structural Graph Queries

Structural graph queries are concerned only with the *structure* (or topology) of the graph and are thus oblivious of additional data such as properties or labels assigned to vertices and edges. Examples for this fundamental kind of query include reachability queries, asking for the existence of a connection (i. e. path) between a pair of specified query vertices, or path queries, requiring identification of a path between the query vertices. In the majority of cases, structural graph queries can be answered by online search algorithms (cf. Section 2.1). Queries of this kind represent the basic form of *relationship analysis*, and play an important role in graph exploration. Further, reachability and path queries can be used as algorithmic building blocks to support other, more expressive, querying mechanisms such as the pattern matching queries presented in the subsequent section. Due to the application of structural graph queries in interactive querying as well as the use as a building block, high efficiency of query execution is imperative. For this reason, *index structures* that support fast query execution have been an active area of research over recent years.
Given a query type (such as reachability or shortest path queries), the common theme of structural graph indexes lies in the idea of precomputing – in an offline indexing stage – certain parts of the answers to the potential user-issued queries. This precomputed information is later used at query time to quickly generate answers without having to start from scratch using a plain search algorithm. In the extreme case, a complete precomputation of answers to all possible queries is feasible, depending mainly on the size of the input graph as well as on structural properties of the graph (certain classes of graphs – such as trees – often permit a complete precomputation of all possible query results which would otherwise be infeasible over general graphs). As an example, consider the case of reachability querying. A complete precomputation would correspond to the materialization of the *transitive closure* of the graph (cf. Definition 2.1), e. g. in the form of a boolean $(n \times n)$-matrix R for which it holds

$$R[s,t] = 1 \quad \iff \quad s \sim_G t,$$

i. e. the matrix entry in row s and column t is set to one if and only if the input graph contains a directed path starting at vertex s and ending at vertex t. Clearly, such a complete precomputation approach, implying indexing time complexity of $O(n^3)$ and index size $O(n^2)$, has limited applicability in practice. In order to achieve fast processing of reachability queries over massive graphs, comprising millions of vertices and edges, recently proposed index structures either rely on sophisticated compression techniques (van Schaik and de Moor, 2011) or compute more coarse-grained index entries (which can be considered a lossy, but space-efficient compression of the transitive closure) that are combined with restricted online search at query time (Seufert et al., 2013, Yıldırım et al., 2010, 2012). One contribution we make in this thesis is the Ferrari index structure for reachability query processing, presented in Chapter 4.

For other problem scenarios, like shortest-path querying, index size and precomputation requirements are even more demanding. In many cases, index structures for path computation exploit structural properties of the graph under consideration, e. g. hierarchical structure of road networks, or small diameter for the case of social networks. The second major contribution made in this thesis, the Path-Sketch index for shortest path approximation, is presented in Chapter 5. We provide a detailed discussion of the state-of-the-art in structural graph indexing for reachability and path queries in the section on related work in the respective chapters.

As hinted above, queries of this type operate exclusively over the *topology* of the graph. In practice, vertices and edges are typically annotated with additional information (see Definition 2.2). As an example, for the case of graph-structured knowledge bases, vertices are labeled with an identifier (e. g. *Barack Obama*) and edges are assigned a label (so-called property) capturing the underlying semantics of the connection (e. g. *presidentOf*). Enriched graph structures of this kind are typically referred to as *property graphs*. We continue our description with an overview of actual *graph database* systems, which integrate the properties of vertices and edges with the structure of the graph in order to allow for expressive querying.

3.1.2 Pattern Matching Queries

Graph Databases

Graph databases – such as Neo4j[1], Titan[2], OrientDB[3], and InfiniteGraph[4] – provide CRUD (Create, Read, Update, Delete) operations over a graph data model in a transactional system (Robinson et al., 2013). While graphs can be represented in a straightforward way using a traditional, relational database system (e. g. by using a single large relation containing the source and target vertex identifiers for each edge), many recent graph database implementations can be considered *native* in the sense that they make use of a physical database layout optimized for traversal operations, the de-facto standard for querying graph databases.

[1]`http://neo4j.org`
[2]`http://thinkaurelius.github.io/titan/`
[3]`http://www.orientechnologies.com/orientdb/`
[4]`http://www.objectivity.com/infinitegraph`

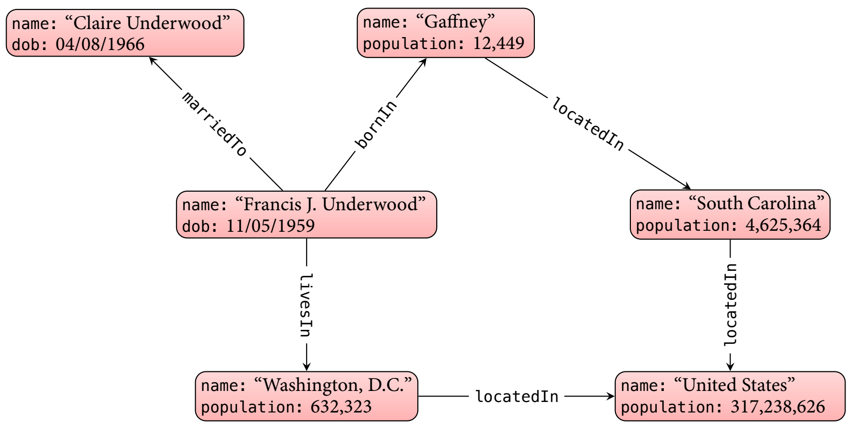


Figure 3.1: Property Graph

Neo4J

We exemplify this basic principle of graph databases using the example of Neo4j, the most widely used graph database implementation available. In the scope of graph databases, querying the data is equivalent to *traversing* the modeled graph. The performance advantage of graph databases is due to the so-called principle of *index-free adjacency* (Robinson et al., 2013), which means that connections in the data are retained in the way the data is stored on disk. While there is no standard query language for graphs (such as SQL for relational databases), the recently proposed Cypher query language[5] has emerged as the query language of choice for property graphs. The pattern matching approach followed by Cypher is best explained as *query-by-example*. For illustration, consider the property graph depicted in Figure 3.1. Suppose we are interested in all people born in the United States. In Cypher, this query is expressed as

```
START us=node:countries(name="United States")
MATCH (p)-[:bornIn]->(l)-[:locatedIn:*]->(us)
RETURN p
```

As the example illustrates, queries correspond to the specification of a subgraph (in this case a path with edge label `locatedIn`) satisfying certain structural properties (in this case on the edge labels, as well as on the fixed endpoint). Vertices of interest are represented as variables (in the example `p`, `l`), that are bound to the actual vertices representing the query result.
While graph databases have gained popularity in recent years, the the lack of standardization is still hindering a wider adoption. However, recent efforts, such as the Blueprints API[6] try to overcome this limitation in the graph database space. In the next section, we focus on a graph-structured data representation backed by the W3C: RDF and the corresponding query language, SPARQL.

[5] `http://www.neo4j.org/learn/cypher`
[6] `http://www.tinkerpop.com`

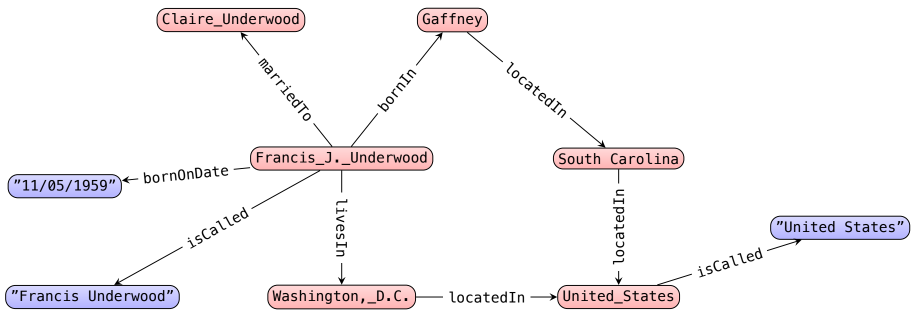


Figure 3.2: RDF Graph

3.1.3 The Resource Description Framework

The **Resource Description Framework** (RDF[7]) is a model for data interchange, primarily in the scope of Semantic Web applications, that can be regarded as an edge-labeled multi-graph representation. RDF data is commonly represented in the form of *triples*, comprising the three components *subject*, *predicate*, and *object*. The subject and object of a triple can be regarded as either *concepts* (entities), or literals such as strings or numbers. The predicate can be regarded as the "label" of the edge, capturing the semantics of the expressed relationship. Thus, in contrast to the concept of property graphs used in graph databases, the atomic units in RDF are triples, and attributes of individual vertices are represented as adjacent vertices rather than being directly associated with the vertex. To illustrate the difference, we express (a subset of) the previous property graph example in RDF, shown in Figure 3.2. Encoded in the form of triples (TTL/N3 format), this graph representation can be factorized into triples as follows:

```
Francis_J._Underwood <bornIn> Gaffney .
Francis_J._Underwood <livesIn> Washington,_D.C. .
Francis_J._Underwood <marriedTo> Claire_Underwood .
Francis_J._Underwood <isCalled> ”Francis Underwood” .
Francis_J._Underwood <bornOnDate> ”11/05/1959” .
Gaffney <locatedIn> South_Carolina .
South_Carolina <locatedIn> United_States .
Washington,_D.C. <locatedIn> United_States .
United_States <isCalled> ”United States” .
```

As shown in the figure, attributes of vertices – such as the date of birth of a person in the example) are represented as literals (strings, numbers, etc.) – are connected to the respective vertex via a labeled edge.
Apart from the amenities of schema-free nature combined with rich querying facilities that RDF shares with graph databases, a major advantage of RDF is its standardization backed by the W3C, which has led to a widespread adoption of RDF in several problem domains, most notably the Semantic Web. RDF has become the prevalent representation for facts about entities, ranging from the biological domain (e. g. the UniProt initiative[8]) to comprehensive knowledge bases such as

[7] http://www.w3.org/RDF/
[8] http://www.uniprot.org

YAGO2 (Hoffart et al., 2013). The standard query language for RDF is SPARQL, a declarative language inspired by the SQL standard for relational databases. The current standard SPARQL 1.1[9] enhances the basic query language with more expressive querying facilities, most notably *property paths*, that allow navigational queries via the introduction of a regular expression-like syntax for path specifications. Using this syntax, the query from the previous example can be expressed in SPARQL 1.1 as follows:

```
SELECT ?name
WHERE {  ?person isCalled ?name .
         ?person bornIn/locatedIn∗ ?country .
         ?country isCalled “United States”
      }
```

Here, the variable length subpath of edges with property `locatedIn` is expressed using Kleene star notation. In SPARQL, joins are expressed using dot notation. For brevity, we omit namespace declarations.

RDF-3X

Numerous systems for efficient processing of SPARQL queries have been proposed in the past (Abadi et al., 2009, Broekstra et al., 2003, Gurajada et al., 2014, Huang et al., 2011, Wilkinson et al., 2003). We briefly introduce RDF-3X (Neumann and Weikum, 2010), a state-of-the-art query processor for RDF data. At its core, RDF-3X can be considered a RISC-style engine that relies on aggressive indexing of the all possible permutations of the SPO-triples contained in the database into separate, clustered B^+ trees, and can thus be regarded as exhaustive indexing of the so-called *giant triples table* approach (which exists only virtually in RDF-3X). The high compression ratio of the index (due mapping of literals to consecutive numeric identifiers, delta-encoding of gaps and variable byte-length encoding), makes this approach feasible for very large RDF datasets. For efficient processing of SPARQL queries, RDF-3X is relying primarily on merge-joins, with a query optimizer concentrating on the join order.

This concludes our overview of the two most important paradigms for graph-structured data, graph databases and the RDF standard. The discussed pattern matching query processing approaches (query by example) represent the *standard* way of querying graph-structured data. In the next section, we discuss a third graph querying paradigm (apart from structural and pattern matching queries), which we refer to as *subgraph extraction queries*. Here, rather than querying a graph by specifying an example pattern, the input provided consists of certain terminal vertices of the graph and a property that has to satisfied by the returned subgraph.

[9] `http://www.w3.org/TR/sparql11-overview/`

3.1.4 Subgraph Extraction Queries

As discussed before, primitive operations such as reachability checks and shortest path queries are common and useful in many problem scenarios. They are mostly used as building blocks of other algorithms and as heuristics used in pattern matching-based query processing approaches like SPARQL and Cypher discussed in the previous section. These query languages allow for the expressive querying for instances of the subgraph pattern specified in the query. While querying mechanisms based on pattern matching cover many information needs, in some settings a third graph querying paradigm is required, where the desired solution is specified by a combination of terminal vertices that have to be included in the query result as well as properties of the requested subgraph. We refer to this kind of queries as *subgraph extraction queries*.

For illustration, consider a so-called *Steiner tree query*, where we are interested in the minimum cost graph (in terms of number of edges) connecting a set of specified vertices. Here, the query consists of a set of vertices together with a desired property (minimum connection cost). An example for subgraph extraction queries, the third major contribution of this thesis is a framework for relationship analysis over knowledge graphs, where we are interested in a coherent subgraph connecting two sets of specified query entities. This algorithm is discussed in detail in Chapter 6.

This concludes our overview of the three major querying paradigms for graph-structured data. In the next section, we discuss several practical aspects of processing large graphs, ranging from an overview of disk-based approaches to distributed querying and analysis of graphs.

3.2 Large-Scale Graph Processing

Over the last two decades, with the advent of the big data era, the scale of graph datasets available for study has grown massively. This growth is not only affecting the sheer *structure* of the graph – i. e. the number of vertices and edges – but also encompasses additional data that associated with the graph, including labels and weights associated with the vertices and edges, temporal information allowing to study evolving graph structures over time by examining the graph at different snapshots, and many more. Many different paradigms for processing large-scale graph-structured data have emerged, both in a centralized as well as a distributed setup.

3.2.1 Centralized Processing

In the centralized setting, the complete graph is stored in a single compute node, either in main-memory, or on secondary memory, i. e. the hard disk (rotating or solid state disk (SSD)). Hybrid solutions are possible, e. g. representing the structure (i. e. edges) but not the entire vertex/edge data in main memory, storing only certain, frequently accessed subgraphs in main memory, etc.

In this section we discuss the handling of massive graphs (i. e. graphs comprising billions of edges) on a single machine, focusing on the two main approaches for handling disk-resident graphs: external-memory algorithms and streaming algorithms. We elaborate both on the underlying computational models used for assessing the cost of an algorithm in these settings, as well as on actual algorithm implementations used in practice.

External-Memory Algorithms

The standard model used in analysis of sequential (graph) algorithms is the **Random Access Machine (RAM)** model, based on the von Neumann-model of computer architecture (Mehlhorn and Sanders, 2008), first introduced by Sheperdson and Sturgis (1963). In this setting, a single processor works on a computing task and has access to an infinite amount of memory cells and a limited number of registers. The model supports machine instructions for loading and storing the content of a memory cell into a register and vice-versa, arithmetic and logical operations, comparisons, assignment of constant values to registers, and jumps in the execution of the program. Each machine instruction can be executed in constant time (Mehlhorn and Sanders, 2008). The abstraction achieved by the RAM model allows for a convenient theoretical analysis of sequential algorithms. In contrast, the **external memory** model, proposed by Aggarwal and Vitter (1988), considers the scenario where the input data to the algorithm is too large to fit into the main memory of the machine. More formally, the amount of available working memory, denoted by M, is restricted to a certain size S. An unlimited amount of additional, slow (external) memory is available. In order to perform computation on data stored in the external memory, it is required to transfer the respective data blocks to the fast main memory. For this purpose, designated *Input-Output operations (I/O operations)* are available to transfer data blockwise – in blocks of size B – from the unlimited and slow to the restricted and fast memory and back. The external memory model not only applies to the case of data stored on hard disk drives that has to be transferred into main memory. Rather, it is defined in a general way for so-called *memory-hierarchies*, that, for example, also encompasses the case where data blocks are transferred between the main memory and the L3-cache. The cost of an external memory algorithm is determined as the sum of required I/O operations, corresponding to the number of transferred data blocks, over the course of its execution. The main primitives, that can be regarded as basic building blocks of external memory algorithms, are *scanning* and *sorting* operations over the data. The number of I/Os required to scan/sort N items is denoted by $\text{scan}(N) = \Theta(N/B)$ and $\text{sort}(N) = \Theta(N/B \log_{M/B}(N/B))$, respectively.

For the case of graph algorithms, the adjacency lists of individual vertices are typically stored sequentially on disk. Retrieving the neighbors of a vertex then requires a seek to the respective position on disk, and transferring the entries in the adjacency list, using sequential read operations. Chiang et al. (1995) propose external memory algorithms based on the scan and sort primitives for the I/O-optimal simulation of graph algorithms in the PRAM model of computation[10] and depth-first search. Several algorithms have been proposed for Breadth-First Search in the external memory model. Munagala and Ranade (1999) propose an algorithm based on first retrieving the multi-set of all neighbors of vertices at the current level in the Bfs expansion, followed by sort and scan operations in order to remove duplicates. Afterwards, vertices from the previous two Bfs levels are removed from the set, resulting in the set of vertices contained in the next level of the Bfs. The overall I/O-complexity of this algorithm is given by $O\big(n + \text{sort}(m)\big)$ I/Os. Mehlhorn and Meyer (2002) extend this algorithm with a preprocessing phase that performs a randomized partitioning of the vertices and their respective adjacency lists, in order to improve the I/O-complexity of the Bfs phase, resulting in (worst-case) I/O complexity of $O\big(\sqrt{n \cdot \text{scan}(n+m)} + \text{sort}(n+m)\big)$. A comparative study of external memory Bfs algorithms is given by Ajwani et al. (2006). Regarding implementation aspects, Ajwani et al. (2007) propose a deterministic variant of above algorithm and reduce the overhead associated with the representation of the Bfs levels, leading to better results on graphs with large diameters. In a more recent work, Meyer and Zeh (2012) extend this algorithm to obtain a single source shortest path algorithm with good average-case performance on graphs with uniformly random edge weights, and propose a worst-case efficient algorithm for graphs with arbitrary real-valued edge weights.
Apart from Bfs, other important algorithms that have been researched in this model include connected components, minimum spanning trees, topological orderings, and diameter computation (Ajwani et al., 2012). For a thorough discussion, we refer to the overview of graph algorithms in external memory given by Katriel and Meyer (2003).

Graphs in the Streaming Model

Data stream processing is a computational model that exhibits certain overlap with the problem setting of external memory algorithms. In this scenario, the input data is processed sequentially (e. g. by scanning through a huge file on disk or in a real-time monitoring system) and the available working memory is assumed to be much smaller than the size of the input data stream (Alon et al., 1996, Flajolet and Martin, 1985, Munro and Paterson, 1980). The relevant parameters in this setting are the size of the available working memory and the number of passes over the stream that are required for computation (often only one). In addition, the amount of time spent for processing a single item from the stream is an important factor, referred to as *per-item-processing-time*. Many important problems like estimation of statistical

[10]The **Parallel Random Access Memory (PRAM)** is a machine model for parallel algorithms, based on the concept of multiple synchronously clocked processors and a shared memory comprising an infinite amount of memory cells. Programs defined in this model are of the SIMD (single instruction multiple data) type.

quantities like frequency moments and counting of distinct elements can be efficiently computed in the streaming model under memory constraints. In general, data stream algorithms have received considerable attention in recent years and a thorough treatment of this field is beyond the scope of this thesis. For this reason, we only discuss graph algorithms in the streaming model.

The drastic memory restrictions (e. g. restriction to a constant) typically found in classical streaming approaches appear too challenging for many graph problems. For this purpose, less restrictive models have been proposed, most notably the semi-streaming model (Feigenbaum et al., 2004). In this setting, the available space is restricted by $O(n \cdot \text{polylog}(n))$, where $\text{polylog}(n)$ refers to a polylogarithmic function of the number of vertices in the input graph, that is, a polynomial in the logarithm of n. Informally, the semi-streaming allows to store data associated with the vertices of the graph, but not the full set of edges. Algorithms that permit efficient approximation schemes under this model include weighted bipartite matchings, computation of the graph diameter (cf. Equation 2.6) and shortest paths in weighted graphs (Feigenbaum et al., 2004). Demetrescu et al. (2009) discuss how the available memory can be traded off against the number of passes for several graph problems.

For further details, an overview of algorithms and applications for (general) data stream processing is given by Muthukrishnan (2005), while Ruhl (2003) discusses several variants of stream processing.

3.2.2 Distributed Processing

External memory algorithms and the semi-streaming model allow for the processing of massive graphs in a centralized setting. However, distributed data processing in large data centers of commodity hardware has become standard in data processing in the big data era. In this section we review the current state-of-the-art in graph processing in the distributed setting, elaborating on the three most popular contemporary approaches, MapReduce, BSP (*bulk synchronous parallel*, e. g. Pregel), and asynchronous graph- parallel computation (GraphLab).

MapReduce

The term **MapReduce** refers to a data parallel programming model proposed by Dean and Ghemawat (2004) for parallel, distributed computation based on the definition of just two functions, *map*, which transforms input key-value pairs into output key-value pairs; followed by *reduce*, which aggregates the different values of a key. While the basic functions originate in functional programming, the tremendous impact of the MapReduce paradigm on modern-day data processing is due to the fact that it allows for user-friendly implementation of distributed algorithms by abstracting to a level where only two simple functions have to be specified. Many important tasks can be efficiently computed by a MapReduce system, for example the construction of inverted indexes for text retrieval. Regarding graphs, also the power iteration method for computing the PageRank scores of individual web pages is conveniently expressed in terms of *map* and *reduce* functions. One drawback of the MapReduce paradigm for graph processing is the need to explicitly pass the original graph structure together with the computed intermediary key-value pairs from mappers to reducers in order to execute iterative algorithms over the graph. While some improvements have been proposed in the past regarding this problem, including the Schimmy design pattern proposed by Lin and Schatz (2010), many graph algorithms can be implemented more efficiently over a *vertex-centric* model, that maintains the graph at individual workers. These approaches are discussed next.

Bulk-Synchronous Parallel

The **Bulk-Synchronous Parallel (BSP)** model was introduced by Valiant (1990) as a *bridging model* between software and hardware for parallel computation. The model can be regarded as a generalization of the PRAM model. BSP relies on a combination of the following basic principles:

1. N components execute computing tasks in parallel,
2. an intermediary message router delivers information between pairs of components, and
3. a synchronization mechanism that is executed in pre-defined time-intervals allowing (subsets) of the components to synchronize their status.

Based on these components, a task is processed as follows: in a so-called *superstep*, each component can receive messages from other components, perform local computation on the available data, and send messages to other components. The original task is then processed in a sequence of supersteps. After a certain amount of time units, the barrier synchronization mechanism checks whether the current superstep has been completed and, based on the result, either initiates the next or allocates more time units to the currently active superstep. The *communication* cost plays an important role in the BSP model. Sending and receiving messages can be regarded as non-local write and read requests (McColl, 1996).

The BSP model has emerged as an important paradigm for graph processing. In this setting, the graph is partitioned and distributed among the machines in the cluster for distributed computation. Similar to MapReduce, the user fills in a function, in this setting a *compute* function that is applied at the individual vertices in parallel. Starting with Google's Pregel framework (Malewicz et al., 2010), several approaches have been proposed recently. Salihoglu and Widom (2013) propose GPS, extending the Pregel API in several ways, most notably with a dynamic repartition scheme that reassigns vertices among different machines based on the observed communication. Further, Salihoglu and Widom (2014) discuss the efficient implementation of graph algorithms in this computational model, advocating serial computation on a small fraction of the input graph orthogonal to the vertex-centric computation model. In their recent work, Tian et al. (2013) propose Giraph++, a graph- (rather than vertex-) centric programming model for distributed processing. In this setting, the partition structure is being made transparent to the user, which can be exploited for algorithm-specific optimizations that can result in substantial performance gains.

Asynchronous Graph-Parallel Computation

Originally proposed as a framework for parallel machine learning, GraphLab (Low et al., 2010, 2012) is now established as a general purpose distributed data processing environment. (Distributed) GraphLab is marketed as *asynchronous, dynamic, graph-parallel* programming model and operates in the shared-memory setting. In contrast to the vertex-centric Bulk Synchronous Parallel paradigm, in GraphLab, individual vertices can read as well as write data to the incident edges and adjacent vertices of the graph. As a result, rather than communicating via messages, interaction among vertices is achieved via shared memory. Consequently, serializability is ensured by avoiding the simultaneous execution of neighboring vertex programs (Gonzalez et al., 2012). Kyrola et al. (2012) port the GraphLab model to the centralized setting, where the graph resides on the hard disk of a single machine. In order to address and exploit the characteristics of graphs exhibiting a power-law degree distribution (such as social networks), Gonzalez et al. (2012) propose the PowerGraph abstraction, a generalization of GraphLab and BSP to the so-called *gather-apply-scatter* model of computation, which factorizes a vertex-program into three phases which allows to parallelize the vertex-programs for high-degree vertices. In addition, the authors propose a partitioning scheme based on the vertex-cut, effectively minimizing the number of compute nodes “spanned” by an individual vertex.

With this overview of the state-of-the-art in large-scale graph processing, we conclude the introductory part of this thesis. In the subsequent chapters, we present our individual contributions for relationship analysis of large graphs.

Part II

Algorithmic Building Blocks

4 Reachability Analysis

Q1: Is there a relationship?
«Is there an interaction between two specific genes in a regulatory network?»

«Which research papers have influenced a certain publication?»

4.1 Problem Definition

In this chapter, we introduce the first of the three algorithmic building blocks that make up this work, an index structure for the reachability problem. In this setting, given a directed graph and a designated source and target vertex, the task of a **reachability index structure** is to determine whether the graph contains a path from the source to the target. This kind of query is a fundamental operation in graph mining and algorithmics, and ample work exists on index support for reachability problems.
The index structure we propose in this chapter solves the following two problem variants:

Problem 1: Two-Sided Reachability

Given Directed graph $G = (V, E)$, pair of vertices $(s, t) \in V \times V$.

Goal Determine **whether** G contains a directed path from vertex s to vertex t.

Problem 2: One-Sided Reachability

Given Directed graph $G = (V, E)$, vertex $s \in V$.

Goal Identify all vertices $t \in V$ that are reachable from s.

This chapter is organized as follows: in the following section, we discuss the characteristics and applications of the reachability analysis variants introduced above and continue with an overview of our proposed index structure.

4.1.1 Problem Characteristics

Computing reachability among vertices is a building block in many kinds of graph analytics, for example biological and social network analysis, traffic routing, software analysis, and linked data on the web, to name a few. As a concrete example, a taxonomic structure – e. g. the type hierarchy in YAGO (Hoffart et al., 2013) – can be represented as a directed acyclic graph (cf. Section 2.1), where the vertices correspond to types (e. g. *Person*, *Organization*, etc.) and the graph contains a directed edge from type s to type t, if t can be regarded as a *specialization* of s. Thus, a taxonomy of real-world types could contain vertices like *Artist* and *Musician* and a directed edge from *Artist* to *Musician*. An operation frequently arising in this setting is the **type subsumption** task: given a pair of types (s, t), determine whether t is a specialization of s. The transitive nature of the relationships in this setting leads to the graph-theoretic problem of determining whether the taxonomy DAG contains a directed path from s to t. Along similar lines, determining all types an entity (directly or indirectly) belongs to can be regarded as an instance of the one-sided reachability problem. Both examples are illustrated in Figure 4.1.

A fast reachability index can also prove useful for speeding up the execution of general graph algorithms – such as shortest path and Steiner tree computations –

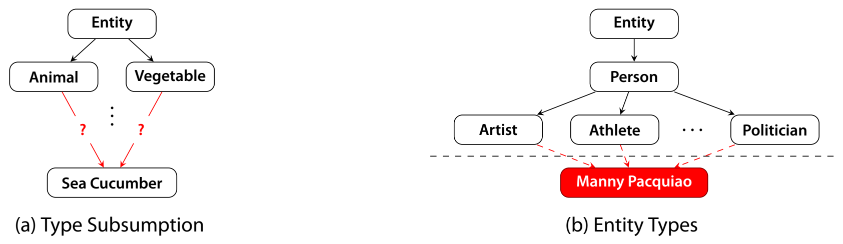


Figure 4.1: Example Applications

via search-space pruning. As an example, Dijkstra's algorithm can be greatly sped up by avoiding the expansion of vertices that cannot reach the target vertex.

While the reachability problem is a light-weight task in terms of its asymptotic complexity, the advent of massive graph structures comprising hundreds of millions of vertices and billions of edges can render even simple graph operations computationally challenging. It is thus crucial for reachability indices to provide answers in logarithmic or ideally near-constant time. Further complicating matters, the index structures, which generally reside in main-memory, are expected to satisfy an upper-bound on the size. In most scenarios, the available space is scarce, ranging from little more than enough to store the graph itself to a small multiple of its size.

Given their wide applicability, reachability problems have been one of the research foci in graph processing over recent years. While many proposed index structures can easily handle small to medium-size graphs comprising hundreds of thousands of vertices – e. g., Agrawal et al. (1989), Chen and Chen (2008), Cohen et al. (2002), Jin et al. (2008, 2009, 2011), Schenkel et al. (2004), Trißl and Leser (2007), van Schaik and de Moor (2011) and Wang et al. (2006) – massive problem instances still remain a challenge to most of them. The only technique that can cope with web-scale graphs while satisfying the requirements of restricted index size and fast query processing time, employs guided online search (Yıldırım et al., 2010, 2012), leading to an index structure that is competitive in terms of its construction time and storage space consumption, yet speeds up reachability query answering significantly when compared to a simple Dfs/Bfs traversal of the graph. However, it suffers from two major drawbacks. Firstly, given the demanding constraints on precomputation time, only *basic heuristics* are used during index construction, which in many cases leads to a suboptimal use of the available space. Secondly and more importantly, while the majority of reachability queries involving pairs of vertices that are not reachable can be efficiently answered, the important class of *positive queries* (i. e. the cases in which the graph actually contains a path from the source to target) has to be regarded as a worst-case scenario due to the need of recursive querying. This can severely hurt the performance of many practical applications where positive queries (i. e. queries that return a positive result since a path indeed exists) occur frequently.

4.1.2 Contribution

The reachability index structure we propose in this chapter – coined Ferrari (for Flexible and Efficient Reachability Range Assignment for gRaph Indexing) – is specifically designed to mitigate the limitations of existing approaches by *adaptively compressing the transitive closure during its construction*. This technique enables the efficient computation of an index geared towards minimizing the expected query processing time, given a user-specified constraint on the resulting index size. Our proposed index supports positive queries efficiently and outperforms GRAIL, the best prior method, on this class of queries by a large margin, while in the vast majority of our experiments also being faster on randomly generated queries. To achieve these performance gains, we adopt the idea of representing the transitive closure of the graph by assigning identifiers to individual vertices and encoding sets of reachable vertices by intervals, first introduced by Agrawal et al. (1989) (explained in Section 4.2). Instead of materializing the full set of identifier ranges at every vertex, we adaptively merge adjacent intervals into fewer yet coarser representations at construction time, whenever a certain space budget is exceeded. The result is a collection of *exact and approximate* intervals that are assigned as labels of the vertices in the graph. These labels allow for a guided online search procedure that can process positive as well as negative reachability queries significantly faster than previously proposed size-constrained index structures.
The interval assignment underlying our approach is based on the solution of an associated interval cover problem. Efficient algorithms for computing such a covering structure together with an optimized guided online search facilitate an efficient and flexible reachability index structure.

In summary, we make the following technical contributions:

- a space-adaptive index structure for reachability queries based on the selective compression of the transitive closure using exact and approximate reachability intervals,
- efficient algorithms for index construction and querying that allow extremely fast query processing on web-scale real world graphs, and
- extensive experiments that demonstrate the superiority of our approach in comparison to GRAIL, the best prior method that satisfies index size constraints.

The remainder of the chapter is organized as follows: In Section 4.2 we discuss the basic idea of interval labeling. Afterwards, we give a short introduction to approximate interval indexing in Section 4.3, followed by an in-depth treatment of our proposed index (Section 4.4). An overview of our query processing algorithm is given in Section 4.5, followed by the experimental evaluation and concluding remarks.

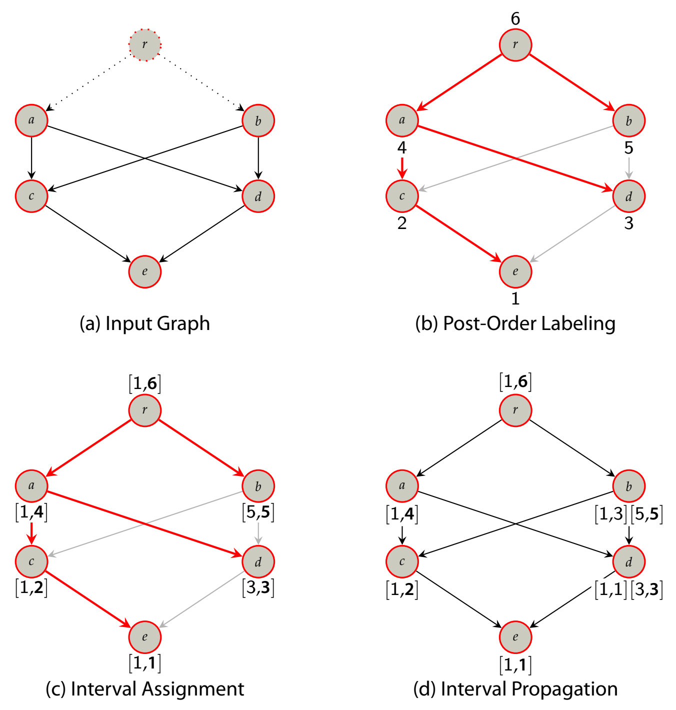


Figure 4.2: Post-Order Interval Assignment

4.2 Interval Labeling

In this section, we introduce the concept of vertex identifier intervals for reachability processing, first proposed by Agrawal et al. (1989), which provided the basis of many subsequent indexing approaches, including our own.

The natural first step – carried out by virtually all reachability indexing approaches – is the condensation of the input graph $G = (V, E)$, that is, collapsing the strongly connected components of the graph into supervertices and computing the index over the resulting DAG, denoted by $\textsc{Cond}(G) = (V_c, E_c)$ (for a discussion of these graph-theoretic concepts, see Chapter 2). Then, at query processing time the query vertices (s, t) are mapped to the identifiers of the respective strongly connected components, $[s], [t]$. If it holds $[s] = [t]$, a positive answer is returned (since both vertices belong to the same strongly connected components and therefore a path exists). Otherwise, the reachability of supervertex $[t]$ from supervertex $[s]$ is determined from the index computed over the DAG $\textsc{Cond}(G)$.

The key idea underlying Agrawal's algorithm is to assign numeric identifiers to the vertices in the graph and represent the reachable sets of vertices in a compressed form by means of interval representations. This technique is based on the construction of a tree cover of the graph followed by post-order labeling of the vertices. In general, the directed acyclic graph $\textsc{Cond}(G)$ contains more than one root vertex (vertex without incoming edges), and thus it is not possible to extract a spanning tree. As a solution, $\textsc{Cond}(G)$ is *augmented* by the addition of a virtual root vertex

r that is connected to every vertex with no incoming edge:

$$(V_c', E_c') := \big(V_c \cup \{r\}, E_c \cup \{(r,v) \mid v \in V, \mathcal{N}^-(v) = \emptyset\}\big). \tag{4.1}$$

Note that this modification has no effect on the reachability relation among the existing vertices of COND(G).

For ease of notation, we will denote the augmented, condensed graph by G' in the remainder of this chapter. Now, let $T = (V_T, E_T)$ denote a tree cover of G'. In order to assign vertex identifiers, the tree is traversed in depth-first manner. In this setting, a vertex v is visited after all its children have been visited. The post-order number $\pi(v)$ corresponds to the order of v in the sequence of visited vertices.

EXAMPLE. Consider the augmented example graph depicted in Figure 4.2(a) with the virtual root vertex r. In this example, the children of a vertex are traversed in lexicographical order, leading to the spanning tree induced by the edges shown in bold in Figure 4.2(b). The first vertex to be visited is vertex e, which is assigned post-order number 1. Vertex a is visited as soon as its children $\{c, d\}$ have been visited. The last visited vertex is the root r.

SPANNING TREE LABELING

The enabling feature, which makes post-order labeling a common ingredient in reachability indices, is the resulting *identifier locality*: For every (complete) subtree of T, the ordered identifiers of the included vertices form a contiguous sequence of integers. The vertex set of any such subtree can thus be compactly expressed as an integer interval. Let $T_v = (V_{T_v}, E_{T_v})$ denote the subtree of T rooted at vertex v. We have

$$\begin{aligned}\{\pi(w) \mid w \in V_{T_v}\} &= \left[\min_{w \in V_{T_v}} \pi(w),\ \max_{w \in V_{T_v}} \pi(w)\right] \\ &= \left[\min_{w \in V_{T_v}} \pi(w),\ \pi(v)\right].\end{aligned} \tag{4.2}$$

Above interval is called *tree interval* of v and will be denoted by $I_T(v)$ in the remainder of the text.
In this chapter, for integers $x, y \in \mathbb{N}, x \leq y$, we use the interval $[x, y]$ to represent the set $\{x, x+1, \ldots, y\}$. For integer intervals $I = [a, b]$ and $J = [p, q]$, we define $|I| := b - a + 1$ to denote the number of elements contained in I. Further, we call J *subsumed by* I, written $J \sqsubseteq I$, if J corresponds to a subinterval of I. Further, J is called an *extension* of I, denoted $I \mathrel{\lrcorner\!\!\sqsupset} J$, if the start-point but not the end-point of J is contained in I.

EXAMPLE (CONT'D). The subtree rooted at vertex a in Figure 4.2(b) contains the vertices $\{a, c, d, e\}$ with the set of identifiers $\{4, 2, 3, 1\}$. Thus, the vertices reachable from a in T are represented by the tree interval $[1, 4]$. The final assignment of tree intervals to the vertices is shown in Figure 4.2(c).

The complete reachability information of the spanning tree T is encoded in the collection of tree intervals. For a pair of vertices $u, v \in V$, there exists a path from u to v in T if (and only if) the post-order number of the target is contained in the tree interval of the source, that is,

$$u \sim_T v \quad \Longleftrightarrow \quad \pi(v) \in I_T(u). \tag{4.3}$$

This reachability index for trees allows for $O(1)$ query processing at a space consumption of $O(n)$.

EXTENSION TO DAGS

While above technique can be used to easily answer reachability queries on trees, the case of general DAGs is much more challenging. The reason is that, in general, the reachable set $\mathcal{R}(v)$ of a vertex v (that is, the set of all vertices reachable from v) in the DAG is only partly represented by the interval $I_T(v)$, as the tree interval only accounts for reachability relationships that are preserved in T. Vertices that can only be reached from a vertex v by traversing one or more non-tree edges have to be handled 0: instead of merely storing the tree intervals $I_T(v)$, every vertex v is now assigned a *set of intervals*, denoted by $\mathcal{I}(v)$. The purpose of this so-called *reachable interval set* is to capture the complete reachability information of a vertex. The sets $\mathcal{I}(v), v \in V$ are initialized to contain only the tree interval $I_T(v)$. Then, the vertices are visited in reverse topological order. For the current vertex v and every incoming edge $(u, v) \in E$, the reachable interval set $\mathcal{I}(v)$ is merged into the set $\mathcal{I}(u)$. The merge operation on the intervals resolves all cases of interval subsumption and extension exhaustively, eventually ensuring interval disjointness. Due to the fact that the vertices are visited in reverse topological order, it is ensured that for every non-tree edge $(s, t) \in E \setminus E_T$, the reachability intervals in $\mathcal{I}(t)$ will be propagated and merged into the reachable interval sets of *s and all its predecessors*. As a result, all reachability relationships are covered by the resulting intervals.

EXAMPLE (CONT'D). Figure 4.2(c) depicts the assignment of tree intervals to the vertices. As described above, in order to compute the reachable interval sets, the vertices are visited in ascending order of the post-order values (or, equivalently, in reverse topological order), thus starting at vertex e. The tree interval $I_T(e) = [1, 1]$ is merged into the set of vertex c, leaving $\mathcal{I}(c) = \{[1, 2]\}$ unchanged due to interval subsumption. Next, $I_T(e)$ is merged at vertex d, resulting in the reachable interval set $\mathcal{I}(d) = \{[1, 1], [3, 3]\}$. The reverse topological order in which the vertices are visited ensures that the interval $[1, 1]$ is further propagated to the vertices b, a, and r.

Query Processing. Using the reachable interval sets $\mathcal{I}(v)$, queries on DAGs can be answered by checking whether the post-order number of the target is contained in *one of the intervals* associated with the source:

$$u \sim v \quad \Longleftrightarrow \quad \exists\big([\alpha, \beta] \in \mathcal{I}(u)\big) : \alpha \leq \pi(v) \leq \beta. \tag{4.4}$$

Example. Consider again the graph depicted in Figure 4.2(d). The reachable vertex set of vertex d is given by $\mathcal{I}(d) = \{[1,1],[3,3]\}$. This set provides all the necessary information in order to answer reachability queries involving the source vertex d.

By ordering the intervals contained in a set, reachability queries can now be answered efficiently in $O(\log n)$ time on DAGs. The resulting index (collection of reachable interval sets) can be regarded as a materialization of the transitive closure of the graph, rendering this approach potentially infeasible for large graphs, both in terms of space consumption as well as computational complexity.

4.3 Approximate Interval Labeling

For massive problem instances, indexing approaches that materialize the transitive closure (or compute a compressed variant without an a priori size restriction), suffer from limited applicability. For this reason, recent work on reachability query processing over massive graphs includes a shift towards guided online search procedures. In this setting, every vertex is assigned a concise label which – in contrast to the interval sets described in Section 4.2 – is restricted by a predefined size constraint. These labels in general do not allow answering the query after inspection of just the source and target vertex labels, yet can be used to prune portions of the graph in an online search.

As a basic example, consider a reachability index that labels every vertex $v \in V$ with its topological order number $\tau(v)$. While this simple variant of vertex labeling is obviously not sufficient to answer a reachability query by means of a single-lookup, a graph search procedure can greatly benefit from the vertex labels: For a given query (s,t), the online search rooted at s can terminate the expansion of a branch of the graph whenever for the currently considered vertex v it holds

$$\tau(v) \geq \tau(t). \tag{4.5}$$

This follows from the properties of a topological ordering.
The recently proposed GRAIL reachability index (Yıldırım et al., 2010, 2012) further extends this idea by labeling the vertices with approximate intervals:

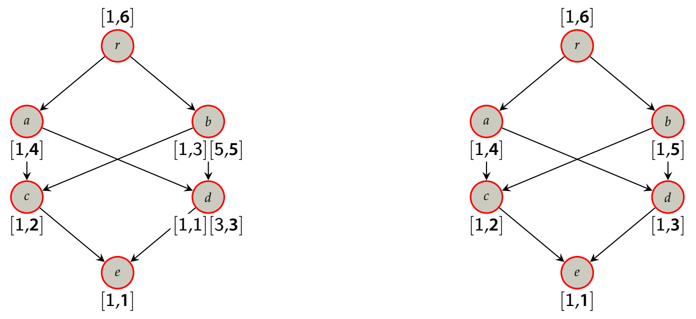
Suppose that for every vertex v we replace the set $\mathcal{I}(v)$ by a single interval

$$I'(v) := \left[\min_{w \in \mathcal{R}(v)} \pi(w), \max_{w \in \mathcal{R}(v)} \pi(w)\right], \tag{4.6}$$

spanning from the lowest to the highest reachable id. For vertex d in our running example, this interval is given by $[1,3]$. This interval is approximate in the sense that all reachable ids are covered whereas false positive entries are possible, similar to the concept of a Bloom filter:

Definition 4.1 (False Positive). *Let $v \in V$ denote a vertex with the approximate interval $I'(v) = [\alpha, \beta]$. A vertex $w \in V$ is called* false positive *with respect to $I'(v)$ if*

$$\alpha \leq \pi(w) \leq \beta \quad \text{and} \quad v \not\rightsquigarrow w. \tag{4.7}$$



(a) Exact Intervals (b) Approximate Intervals

Figure 4.3: Approximate Interval Labeling

Example (cont'd). In our example, vertex c is a false positive with respect to $I'(d)$ as $\pi(c) \in I'(d)$ whereas no path exists from d to c.

Obviously, the single interval $I'(v)$ is not sufficient to establish a definite answer to a reachability query of the form (G, v, w). However, all queries involving a target id $\pi(w)$ that lies outside the interval, i. e.

$$\pi(w) < \alpha \quad \text{or} \quad \pi(w) > \beta, \tag{4.8}$$

can be answered instantly with a negative answer, similar to the basic approach based on Equation (4.5). In the opposite case, that is,

$$\alpha \leq \pi(w) \leq \beta, \tag{4.9}$$

no definite answer to the reachability query can be given and the online search procedure continues with an expansion of the child vertices, terminating as soon as the target vertex is encountered or all branches have been expanded or pruned, respectively.

In practical applications the GRAIL index assigns a number of $k \geq 1$ such approximate intervals to every vertex, each based on a different (random) spanning tree of the graph. The intuition behind this labeling is that an ensemble of independently generated intervals improves the effectiveness of the vertex labels since each additional interval potentially reduces the remaining false positive entries. The advantage of this indexing approach over a materialization of the transitive closure is obvious: the size of the resulting labels can be determined a priori by an appropriate selection of the number (k) of intervals assigned to each vertex. In addition, the vertex labels are easily computed by means of k Dfs traversals of the graph.

Empirically, GRAIL has been shown to greatly improve the query processing time over online Dfs search in many cases. However, especially in the case of positive queries, a potentially large portion of the graph still has to be expanded. Some extensions have been proposed to GRAIL (Yıldırım et al., 2012) to improve performance on positive queries, such as the so-called positive-cut filter, where the lower part (begin to mid-point) of each interval is approximate and the upper part denotes an exact part in the sense that vertices from this interval part are definitely reachable. However, the processing time in these cases remains high. Furthermore, while an increase of the number of intervals assigned to the vertices potentially reduces false positive elements, no guarantee can be made due to the heuristic nature of the underlying algorithm. As a result, in many cases superfluous intervals are stored, in some cases *negatively* impacting query processing time.

4.4 THE FERRARI REACHABILITY INDEX

In this section, we present the FERRARI reachability index which enables fast query processing performance over massive graphs by a more involved vertex labeling approach. The main goal of our index is the assignment of a *mixture of exact and approximate* reachability intervals to the vertices with the goal of minimizing the expected query processing time, given *a user-specified size constraint on the index*[1]. Contrasting previously proposed approaches, we show both theoretically and empirically that the interval assignment of the FERRARI index utilizes the available space for maximum effectiveness of the vertex labels.

Similar to previously proposed index structures (Agrawal et al., 1989, van Schaik and de Moor, 2011, Yıldırım et al., 2010, 2012), we use intervals to encode reachability relationships of the vertices. However, in contrast to existing approaches, FERRARI can be regarded as an *adaptive transitive closure compression* algorithm. More precisely, FERRARI uses *selective interval set compression*, where a subset of adjacent intervals in an interval set is merged into a smaller number of approximate intervals. The resulting vertex label then retains a high pruning effectiveness under a given size-restriction.
Before we delve into the details of our algorithms and the according query processing procedure, we first introduce the basic concepts that facilitate our interval assignment approach.

The FERRARI index distinguishes between two types of intervals: approximate (similar to the intervals in Section 4.3) and exact (as in Section 4.2), depending on whether they contain false positive elements or not.

Let I denote an interval. To easily distinguish between interval types, we introduce an indicator variable η_I such that

$$\eta_I := \begin{cases} 0 & \text{if } I \text{ approximate,} \\ 1 & \text{if } I \text{ exact.} \end{cases} \tag{4.10}$$

As outlined above, a main characteristic of FERRARI is the assignment of size-restricted interval sets comprising approximate and exact intervals as vertex labels. Before we introduce the algorithmic steps that facilitate the index construction, it is important to explain how reachability queries can be answered using the proposed interval sets. Let (G, s, t) denote a reachability query and $\mathcal{I}(s) = \{I_1, I_2, \ldots, I_N\}$ the set of intervals associated with vertex s. In order to determine whether t is

[1]Conceptually, GRAILs positive cut-filter can also be regarded as labeling with a mixture of exact and approximate intervals. However, every interval in GRAIL is split into two such parts, rather than a real mixture of approximate and exact intervals as used in FERRARI.

reachable from vertex s, we have to check whether the post-order identifier $\pi(t)$ of t is included in one of the intervals in the set $\mathcal{I}(s)$. If $\pi(t)$ lies outside of all intervals $I_1, \ldots, I_N$, the query terminates with a negative answer. If however it holds that $\pi(t) \in I_i$ for one $I_i \in \mathcal{I}(s)$, we have to distinguish two cases:

1. if I_i is exact then s is guaranteed to reach vertex t and
2. if I_i is approximate, the neighbors of vertex s have to be queried recursively until a definite answer can be given.

Obviously, recursive expansions are costly and it is thus desirable to minimize the number of cases that require lookups beyond the source vertex.
To formally introduce the according optimization problem, we define the notion of *interval covers*:

Definition 4.2 (k-Interval Cover). *Let $k \geq 1$ denote an integer and $\mathcal{I} = \{[\alpha_1, \beta_1], \ldots, [\alpha_N, \beta_N]\}$ a set of intervals. A set $\mathcal{C} = \{[\alpha'_1, \beta'_1], \ldots, [\alpha'_l, \beta'_l]\}$ is called* k-interval cover *of $\mathcal{I}$, written as $\mathcal{C} \sqsupseteq_k \mathcal{I}$, if $\mathcal{C}$ covers all elements from $\mathcal{I}$ using no more than k intervals, i. e.*

$$\bigcup_{i=1}^{N} \{ j \mid \alpha_i \leq j \leq \beta_i \} \subseteq \bigcup_{i=1}^{l} \{ j' \mid \alpha'_i \leq j' \leq \beta'_i \} \tag{4.11}$$

$$\textit{with} \quad l \leq k. \tag{4.12}$$

Note that an interval cover of a set of intervals is easily obtained by merging an arbitrary number of adjacent intervals in the input set. Next, we address the problem of choosing an k-interval cover that maximizes the pruning effectiveness.

Definition 4.3 (Optimal k-Interval Cover). *Let $k \geq 1$ denote an integer and $\mathcal{I} = \{I_1, I_2, \ldots, I_N\}$ an interval set of size N. We define the* optimal k-interval cover *of $\mathcal{I}$ by*

$$\mathcal{I}^*_k := \operatorname*{arg\,min}_{\mathcal{C}\,:\,\mathcal{I} \sqsubseteq_k \mathcal{C}} \sum_{I \in \mathcal{C}} (1 - \eta_I)\ |I|, \tag{4.13}$$

that is, the cover of $\mathcal{I}$ with no more than k intervals and the minimum number of elements in approximate intervals.

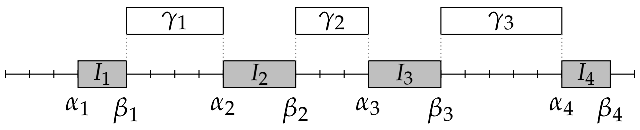
In the FERRARI index structure, we replace the set of exact reachability intervals $\mathcal{I}(v)$ by its optimal k-interval cover $\mathcal{I}^*_k(v)$, which is then used as the vertex label. This way, we retain maximal effectiveness for terminating a query. The reason is that the number of cases that require recursive querying directly corresponds to the number of elements contained in approximate intervals.

4.4.1 Computing the Optimal Interval Cover

While the special cases $k = N$ (optimal k-interval cover of $\mathcal{I}$ is the set $\mathcal{I}$ itself) and $k = 1$ (optimal solution corresponds to the single approximate interval assigned by GRAIL, see Equation 4.6) are easily solved, we next introduce an algorithm that solves the problem for general values of k:

As hinted above, an interval cover can be computed by selectively merging adjacent intervals from the original assignment made to the vertex v. In order to derive an algorithm for computing $\mathcal{I}_k^*(v)$, we first transform the interval set at the vertex v into its dual representation where the *gaps between intervals* are specified. Like before, let $\mathcal{I} = \{I_1, I_2, \dots, I_N\}$ with $I_i = [\alpha_i, \beta_i]$.

The set $\Gamma := \{\gamma_1, \gamma_2, \dots, \gamma_{N-1}\}$, $\gamma_i = [\beta_i + 1, \alpha_{i+1} - 1]$ denotes the *gaps* between the intervals contained in $\mathcal{I}$:



Note that the gap set Γ together with the boundary elements α_1, β_N is an equivalent representation of $\mathcal{I}$. For a subset $G \subseteq \Gamma$ we denote by $\zeta(G)$ the induced interval set obtained by merging adjacent intervals I_i, I_{i+1} if for their mutually adjacent gap γ_i it holds $\gamma_i \notin G$. As an illustrative example, for the interval set depicted above we have $\zeta(\{\gamma_2\}) := \{[\alpha_1, \beta_2], [\alpha_3, \beta_4]\}$.

Every induced interval set $\zeta(G)$ actually corresponds to a $|G| + 1$-interval cover of the original set $\mathcal{I}$. It is easy to see that the *optimal* k-interval cover can equivalently be specified by a subset of gaps.

In order to compute the optimal k-interval cover, we thus transform the problem defined in Equation (4.13) into the equivalent problem of selecting the "best" $k - 1$ gaps from the original gap set Γ (or, equivalently, determining the $|\mathcal{I}| - k - 1$ gaps that are not part of the solution). For a potential solution $G \subseteq \Gamma$ of at most $k - 1$ gaps to preserve, we can assess the associated *cost*, measured by the number of elements in the induced interval cover that are contained in approximate intervals:

$$c(G) := \sum_{I \in \zeta(G)} (1 - \eta'_I)\ |I|, \tag{4.14}$$

where for $I \in \zeta(G)$ it holds

$$\eta'_I := \begin{cases} 1 & \text{if } I \in \mathcal{I} \text{ and } \eta_I = 1, \\ 0 & \text{else.} \end{cases} \tag{4.15}$$

Clearly, our goal is to determine the set $\Gamma^*_{k-1} \subseteq \Gamma$ of gaps such that

$$\Gamma^*_{k-1} := \operatorname*{arg\,min}_{G \subseteq \Gamma,\ |G| \leq k-1} c(G). \tag{4.16}$$

We propose a dynamic programming approach to obtain the optimal set of $k-1$ gaps. In the following, we denote for a sequence of intervals – given by $\mathcal{I} = (I_1, I_2, \ldots, I_N)$ – the subsequence consisting of the first j intervals by $\mathcal{I}_j := (I_1, I_2, \ldots, I_j)$. Now, observe that every set of gaps $G \subseteq \Gamma, |G| \leq k-1$ represents a valid k-interval cover for each of the interval sequences $\mathcal{I}_{\max\{i \mid \gamma_i \in G\}}, \ldots, \mathcal{I}_N$, yet at different costs (the cost corresponding to each of these coverings is strictly non-decreasing). In order to obtain an optimal substructure formulation, consider the problem of computing the optimal k-interval cover for the interval sequence $\mathcal{I}_j$. The possible interval covers can be represented by a collection of sets of gaps:

$$\begin{aligned} \mathcal{G}_{k-1}(\mathcal{I}_j) &= \big\{ G \subseteq \{\gamma_1, \gamma_2, \ldots, \gamma_{j-1}\} \bigm| |G| \leq k-1 \big\} \\ &= \mathcal{G}^-_{k-1}(\mathcal{I}_j) \cup \mathcal{G}^+_{k-1}(\mathcal{I}_j) \end{aligned} \tag{4.17}$$

with

$$\begin{aligned} &\mathcal{G}^-_{k-1}(\mathcal{I}_j) := \mathcal{G}_{k-1}(\mathcal{I}_{j-1}) \\ \text{and} \quad &\mathcal{G}^+_{k-1}(\mathcal{I}_j) := \big\{ G \cup \{\gamma_{j-1}\} \bigm| G \in \mathcal{G}_{k-2}(\mathcal{I}_{j-1}) \big\} \end{aligned} \tag{4.18}$$

that is, $\mathcal{G}^-_{k-1}(\mathcal{I}_j)$ is the collection of all subsets of $\{\gamma_1, \ldots, \gamma_{j-1}\}$ comprising not more than $k-1$ elements and $\mathcal{G}^+_{k-1}(\mathcal{I}_j)$ corresponds to the collection of all sets of gaps including γ_{j-1} and not more than $k-2$ elements from $\{\gamma_1, \gamma_2, \ldots, \gamma_{j-2}\}$. From Equations (4.17,4.18) we can deduce that every k-interval cover of $\mathcal{I}_j$ and thus the optimal solution is either

- a k-interval cover of $\mathcal{I}_{j-1}$ or
- a $k-1$-interval cover of $\mathcal{I}_{j-1}$ combined with the gap γ_{j-1} between the last two intervals, I_{j-1} and I_j.

Thus, for the optimal solution $\Gamma^*_{k-1}(\mathcal{I}_j)$ we have

$$\begin{aligned} c\big(\Gamma^*_{k-1}(\mathcal{I}_j)\big) &= \min \left\{ \min_{G \in \mathcal{G}^+_{k-1}(\mathcal{I}_j)} c(G), \min_{G \in \mathcal{G}^-_{k-1}(\mathcal{I}_j)} c(G) \right\} \\ &= \min \left\{ \underbrace{\min_{G \in \mathcal{G}_{k-2}(\mathcal{I}_{j-1})} c(G),}_{\text{preserve last gap}} \right. \\ &\qquad \underbrace{\min_{G \in \mathcal{G}^+_{k-1}(\mathcal{I}_{j-1})} c(G) + |I_{j-1}| + |\gamma_{j-1}| + |I_j|,}_{\text{merge last gap, prev. interval exact}} \\ &\qquad \left. \underbrace{\min_{G \in \mathcal{G}^-_{k-1}(\mathcal{I}_{j-1})} c(G) + |\gamma_{j-1}| + |I_j|}_{\text{merge last gap, prev. interval approx.}} \right\} \end{aligned}$$

$$= \min \left\{ \begin{array}{l} c(\Gamma^*_{k-2}(\mathcal{I}_{j-1})), \\ c^+(\Gamma^*_{k-1}(\mathcal{I}_{j-1})) + |I_{j-1}| + |\gamma_{j-1}| + |I_j|, \\ c^-(\Gamma^*_{k-1}(\mathcal{I}_{j-1})) + |\gamma_{j-1}| + |I_j|) \end{array} \right\}. \tag{4.19}$$

We can exploit the optimal substructure derived in Equation (4.19) for the desired dynamic programming procedure: For each $\mathcal{I}_i, 1 \leq i \leq N$ we have to compute the k'-interval cover for $k - N + i \leq k' \leq k$, thus obtaining the optimal solution in time $O(kN)$.

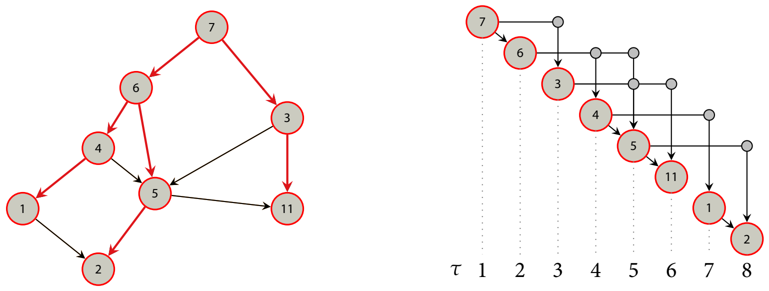
In some practical applications the amount of computation can become prohibitive, as one instance of the problem has to be solved for every vertex in the graph. Thus, in our implementation, we use a simple and fast greedy algorithm that, starting from the empty set, iteratively adds the gap $\gamma \in \Gamma$ that leads to the greatest reduction in cost given the current selection G, until at most $k - 1$ gaps have been selected, then compute the interval cover from $\zeta(G)$. While the gain in speed comes at the cost of a potentially suboptimal cover, our experimental evaluation demonstrates that this approach works well in practice.
In the next section, we explain how above interval covering technique is eventually put to use during the reachability index computation.

4.4.2 Index Construction

At precomputation time, the user specifies a certain budget $B = kn, k \geq 1$ of intervals that can be assigned to the vertices, thus directly controlling the tradeoff between index size/precomputation time and pruning effectiveness of the respective vertex labels. The subsequent index construction procedure can be broken down into the following main stages:

Tree Cover Construction

Agrawal et al. (1989) propose an algorithm for computing a tree cover that leads to the minimum number of exact intervals to store. This tree can be computed in $O(mn)$ time, rendering the approach infeasible for the case of massive graphs. While, in principle, heuristics could be used, that are based on centrality measures or estimates of the sizes of reachable sets (Cohen, 1997, Palmer et al., 2002), we settle for a simpler solution that does not yield a certain approximation guarantee yet performs well in practice. We argue that a good tree cover should cover as many reachability relationships as possible in the initial tree intervals (see Equation 4.2). Therefore, every edge included in the tree should provide a connection between as many pairs of vertices as possible. To this end, we propose the following procedure to heuristically construct such a tree cover T:
Let $\tau : V \to \{1, 2, \ldots, n\}$ denote a topological ordering of the vertices, i. e. for all $(u, v) \in E$ it holds $\tau(u) < \tau(v)$. Such a topological ordering is easily obtained by the classical textbook algorithm (Cormen et al., 2009) in time $O(m + n)$.



(a) Input Graph (b) Topological Ordering

Figure 4.4: Tree Cover Construction

We interpret the topological order number of a vertex as the number of *potential predecessors* in the graph, because the number of predecessors of a given vertex is upper bounded by its position in the topological ordering. For a vertex v with set of predecessors $\mathcal{N}^-(v)$, we select the edge from vertex $p \in \mathcal{N}^-(v)$ with highest topological order number for inclusion in the tree, that is

$$p := \arg\max_{u \in \mathcal{N}^-(v)} \tau(u). \tag{4.20}$$

The intuition is that vertex p has the highest number of potential predecessors and thus the selected edge (p, v) has the potential of providing a connection from a large number of vertices to v, eventually leading to a high number of reachability relationships encoded in the resulting tree intervals.

Example. Consider the DAG depicted in Figure 4.4(a). A topological ordering of the vertices is displayed in Figure 4.4(b). Based on this assignment of topological order ids (τ), we can extract the tree cover shown as red edges in Figure 4.4(a). For vertex 2, the potential predecessors are given by $\mathcal{N}^-(2) = \{1, 5\}$. Since $\tau(1) = 7 > 5 = \tau(5)$, the edge $(1, 2)$ is selected for inclusion in the tree.

An overview of the tree cover construction algorithm is depicted in Algorithm 3.

Interval Set Assignment

As the next step, given the tree cover T, indexing proceeds by assigning the exact tree interval $I_T(v)$, encoding the reachability relationships within the tree at each vertex v. This interval assignment can be obtained using a single depth-first traversal of T.
In order to label every vertex v with a k-interval cover $\mathcal{I}'(v)$ of its true reachable interval set, we visit the vertices of the graph in reverse topological order, that is, starting from the leaf with highest topological order, proceeding iteratively backwards to the root vertex. We initialize for vertex v the reachable interval interval set as $\mathcal{I}'(v) := \{I_T(v)\}$. For the currently visited vertex v and every outgoing edge

Algorithm 3: TreeCover(G)

```
Input: directed acyclic graph $G = (V, E)$
begin
    $T \leftarrow (V_T, E_T) \leftarrow (V, \emptyset)$
    $\tau \leftarrow$ TopologicalSort($G$)
    for $i = n$ downto 1 do
        $\{v\} \leftarrow \{u \in V \mid \tau(u) = i\}$
        if $\mathcal{N}^-(v) \neq \emptyset$ then
            $E_T \leftarrow E_T \cup \left(\arg\max_{u \in \mathcal{N}^-(v)} \tau(u), v\right)$
    return $T$
```

$(v, w) \in E$, we merge $\mathcal{I}'(w)$ into $\mathcal{I}'(v)$, such that the resulting set of intervals is closed under subsumption and extension.[2]

Then, in order to satisfy the size restriction of at most k intervals associated with a vertex, we replace $\mathcal{I}'(v)$ by its k-interval cover which is then stored as the vertex label of v in our index. It is easy to see that the resulting index consisting of the sets of approximate and exact intervals $\mathcal{I}'(v), v \in V$ comprises *at most* $nk = B$ intervals. The upper bound

$$\sum_{v \in V} |\mathcal{I}'(v)| \leq B \tag{4.21}$$

is usually not tight, i. e. in practice, much less than B intervals are assigned. As an example, in the case of leaf vertices or the root vertex, in general a single interval suffices. The complete procedure is shown in detail in Algorithm 4. The name of the algorithm – Ferrari-L – thus reflects the fact that a *local* size restriction, $|\mathcal{I}'(v)| \leq k$, is satisfied by every interval set $\mathcal{I}'(v)$.
Note that even though an optimal algorithm can be used to compute the k-interval covers, the optimality of the local result does in general not extend to the global solution, i. e. the full set of vertex labels. The reason for this is the fact that adjacent intervals that are merged during the interval cover computation are propagated to the parent vertices. As a result, at the point during the execution of the algorithm where the interval set of the parent p has to be covered, the k-interval cover is computed without knowledge of the true (exact) reachability intervals of p. More precisely, the input to the covering algorithm is a combination of approximate (thus previously merged) and exact intervals. Nevertheless, the resulting vertex labels prove very effective for early termination of reachability queries, as our experimental evaluation indicates. To further improve our reachability index, in the next section we propose a variant of the labeling algorithm that leads to an even better utilization of the available space.

[2] In our implementation we maintain non-adjacent intervals in the set, that is, for $[\alpha_1, \beta_1], [\alpha_2, \beta_2] \in \mathcal{I}$ it must hold $\beta_1 < \alpha_2$. When sets of approximate and exact intervals are merged, the type of the resulting interval is based on several factors. For example, when an exact interval is extended by an approximate interval, the result will be one long approximate range.

Algorithm 4: Ferrari-L(G, B)

```
Input: directed, acyclic graph $G$, interval budget $B = kn$
Result: set of at most $k$ approximate and exact reachability intervals $\mathcal{I}'(v)$ for
        every vertex $v \in V$
1  begin
2      $T \leftarrow$ TreeCover$(G)$
3      $I_T \leftarrow$ AssignTreeIntervals$(T)$
4      $k \leftarrow \frac{B}{n}$
5      for $i = n$ to $1$ do                          ▷ visit vertices in reverse topological order
6          $\{v\} \leftarrow \{v \in V \mid \tau(v) = i\}$
7          $\mathcal{I}'(v) \leftarrow \{I_T(v)\}$
8          foreach $w \in \mathcal{N}^+(v)$ do
9              $\mathcal{I}'(v) \leftarrow \mathcal{I}'(v) \oplus \mathcal{I}'(w)$      ▷ merge interval sets
           ▷ replace intervals by $k$-interval cover
10         $\mathcal{I}'(v) \leftarrow k$-IntervalCover$(\mathcal{I}'(v))$
11     return $\{\mathcal{I}'(v) \mid v \in V\}$
```

4.4.3 Dynamic Budget Allocation

As mentioned above, the interval assignment as described in Algorithm 4 usually leads to a total number of far less than B intervals stored at the vertices. In order to better exploit the available space, we extend our algorithm by introducing the concept of *deferred interval merging* where we can assign more than k intervals to the vertices on the first visit, potentially requiring to revisit a vertex at a later stage of the algorithm.

The indexing algorithm for this interval assignment variant works as follows: Similar to Ferrari-L, vertices are visited in reverse topological order and the interval sets of the neighboring vertices (considering outgoing edges) are merged into the interval set $\mathcal{I}'(v)$ for the current vertex v. However, in this new variant, subsequent to merging the interval sets we compute the interval cover comprising at most ck intervals, given a constant $c \geq 1$. This way, more intervals can be stored in the vertex labels. After the ck-interval cover has been computed, the vertex v is added to a min-heap structure where the vertices are maintained in ascending order of their degree. This procedure continues until the already assigned interval sets sum up to a size of more than B intervals. In this case, the algorithm repeatedly pops the minimum element from the heap and restricts its respective interval set by computing the k-interval cover. This deferred interval set restriction is repeated until the number of assigned intervals again satisfies the size constraint B.

Above procedure leads to a much better utilization of the available space and thus a higher effectiveness of the resulting reachability index. The improvement comes at the cost of increased index computation time, in practice the increase is two-fold in the worst-case, negligible in others. Our experimental evaluation suggests that a value of $c = 4$ provides a reasonable tradeoff between efficiency of construction

Algorithm 5: Ferrari-G(G, B)

```
Input: directed, acyclic graph $G$, interval budget $B = kn$, constant $c \geq 1$
Result: set of approximate and exact reachability intervals $\mathcal{I}'(v)$ for every
        vertex $v \in V$ s. t. the total number of intervals is upper-bounded by $B$
begin
    $T \leftarrow$ TreeCover$(G)$
    $I_T \leftarrow$ AssignTreeIntervals$(T)$
    $H \leftarrow$ InitializeMinHeap$()$
    $s \leftarrow 0$                                   ▷ number of currently assigned intervals
    for $i = n$ to 1 do                               ▷ visit vertices in reverse topological order
        $\{v\} \leftarrow \{v \in V \mid \tau(v) = i\}$
        $\mathcal{I}'(v) \leftarrow \{I_T(v)\}$
        foreach $w \in \mathcal{N}^+(v)$ do
            $\mathcal{I}'(v) \leftarrow \mathcal{I}'(v) \oplus \mathcal{I}'(w)$          ▷ merge interval sets
        ▷ replace intervals by $ck$-interval cover
        $\mathcal{I}'(v) \leftarrow ck$-IntervalCover$(\mathcal{I}'(v))$
        $s \leftarrow s + |\mathcal{I}'(v)|$
        if $|\mathcal{I}'(v)| > k$ then
            Heap-Push$(H, v, |\mathcal{N}^+(v)|)$
        while $s > B$ do
            $w \leftarrow$ Heap-Pop$(H)$
            $\mathcal{I}'(w) \leftarrow k$-IntervalCover$(\mathcal{I}'(w))$
            $s \leftarrow s - |\mathcal{I}'(w)| + k$
    return $\{\mathcal{I}'(v) \mid v \in V\}$
```

and resulting index quality. This second indexing variant is shown in detail in Algorithm 5. We refer to the algorithm as the *global* variant (Ferrari-G), as in this case the size constraint is satisfied over all vertices – in contrast to the local size constraint of Ferrari-L.

In the next section, we provide more details about our query answering algorithm and additional heuristics that further speed up query execution over the Ferrari index.

4.5 Query Processing

The basic query processing over Ferrari's reachability intervals is straightforward and resembles the basic approach of Agrawal et al. (1989): For every vertex v, the intervals in the set $\mathcal{I}'(v)$ are maintained in sorted order. Then, given a two-sided reachability query (G, s, t), it can be determined in time $O(\log |\mathcal{I}'(v)|)$ time whether the target id, $\pi(t)$, is contained in one of the intervals of the source. The query returns a negative answer ($s \not\sim t$) if the target id lies outside all of the intervals and a positive answer if it is contained in one of the exact intervals. Finally, if $\pi(t)$ falls into one of the approximate intervals, the neighbors of s are expanded recursively using a Dfs search algorithm.
For the case of one-sided reachability queries of the form (G, s), where only the source (target) vertex is specified and the task is to enumerate all vertices t that can be reached from s (that can reach s), the query processing algorithm works as follows: starting from the source s, all vertices whose postorder id is included in exact intervals are added to the set of reachable vertices. Then, a online search procedure is executed in order to refine the remaining, approximate intervals. More specifically, the neighbors of the current vertex are expanded in a Bfs manner until all cases of 1 within approximate intervals have been resolved.

In the next section, we introduce some heuristics that can further speed up query processing for two-sided reachability queries.

4.5.1 Seed Based Pruning

It is evident that, in the case of recursive querying, the performance of the algorithm depends on the number of vertices that have to be expanded during the online search. Vertices with a very high out-degree are especially costly as they might lead to a large number of recursive queries. In practice, such high degree vertices are to be expected due to the fact that

(i) most of the real-world graphs in our target applications will follow a power-law degree distribution and

(ii) the condensation graph obtained from the input graph produces high-degree vertices in many cases because the large strongly connected components usually exhibit a large number of outgoing edges.

To overcome this problem, we propose to determine a set of *seed vertices* $S \subseteq V$ and assign an additional label to every vertex v in the graph, indicating for every $\sigma \in S$ whether G contains a forward (backward) directed path from v to s.
This labeling scheme works as follows: Every vertex will be associated with two sets, $S^-(v)$ and $S^+(v)$, such that

$$S^-(v) := \{\sigma \in S \mid \sigma \sim v\} \quad \text{and} \quad S^+(v) := \{\sigma \in S \mid v \sim \sigma\}. \tag{4.22}$$

Without loss of generality, we describe the procedure for assigning the sets S^+: for every vertex v, we initialize the set as follows:

$$S^+(v) := \begin{cases} \{v\} & \text{if } v \in S, \\ \emptyset & \text{otherwise.} \end{cases} \tag{4.23}$$

We then maintain a FIFO-queue of all vertices, initialized to contain all leaves of the graph. At each step of the algorithm, the first vertex v is removed from the queue. Then, for every predecessor $u, (u, v) \in E$ we merge the set $S^+(v)$ into the current set $S^+(u)$. If all successors of u have been processed, u itself is added to the set. The algorithm continues until all vertices of the graph have been labeled. It is easy to see that above procedure can be implemented efficiently. The approach for assignment of the sets S^- is similar (starting from the root vertices).
Once assigned, the sets can be used by the query processing algorithm in the following way: For a query (G, s, t),

1. if $S^+(s) \cap S^-(t) \neq \emptyset$, then $s \sim t$.
2. if there exists a seed vertex σ s. t. $\sigma \in S^-(s)$ and $\sigma \notin S^-(t)$, that is, the seed σ can reach s but not t, the query can be terminated with a negative answer ($s \not\sim t$).

In our implementation we choose to elect the s vertices with maximum degree as seed vertices (requiring a minimum degree of 1). The choice of s can be specified prior to index construction, in our experiments we set $s = 32$ in order to represent an individual seed set as a 4-byte integer.

4.5.2 Pruning Based on Topological Properties

We enhance the Ferrari index with two additional powerful criteria that allow additional pruning of certain queries. First, we adopt the effective topological level filter that was first proposed for the GRAIL index (Yıldırım et al., 2012):

$$t(v) = \begin{cases} 1 & \text{if } v \text{ is a leaf,} \\ 1 + \max_{w \in \mathcal{N}^+(v)} t(w) & \text{otherwise,} \end{cases}$$

leading to the pruning criterion

$$t(w) \leq t(v) \implies v \not\sim w.$$

Second, we maintain the topological order $\tau(v)$ of each vertex v for pruning as shown in Equation (4.5).

Before we proceed to the experimental evaluation of our index, we first give an overview of previously proposed reachability indexing approaches.

4.6 Related Work

Due to the crucial role played by reachability queries in applications, reachability index structures that facilitate fast execution of this type of queries have been subject of active research. Instead of exhaustively surveying previous results, we describe some of the key proposals here. For a detailed survey, we direct the reader to the survey of Xu Yu and Cheng (2010). In this section, we distinguish between reachability query processing techniques that are able to answer queries using only the label information on vertices specified in the query, and those which use the index to speed up guided online search over the graph.

Before we proceed, it is worth noting that there are two recent proposals that aim to speed up the reachability queries from a different direction compared to the standard graph indexing approaches. First, Jin et al. (2012a) propose a novel way to compact the graph before applying any reachability index. Naturally, this technique can be used in conjunction with Ferrari, hence we consider it orthogonal to the focus of the this work. The other proposal is to compress the transitive closure through a carefully optimized word-aligned bitmap encoding of the intervals (van Schaik and de Moor, 2011). The resulting encoding, called PWAH-8, is shown to make the interval labeling technique of Nuutila (Nuutila, 1996) scale to larger graph datasets. In our experiments, we compare our performance with both Nuutila's Intervals assignment technique as well as the PWAH-8 variant.

4.6.1 Direct Reachability Indices

Indices in this category answer a reachability query (G, s, t) using just the labels assigned to s and t. Apart from the classical algorithm of Agrawal et al. (1989) described in Section 4.2, another approach based on tree covering was proposed by Wang et al. (2006) and focuses on sparse graphs (targeting near-tree structures in XML databases). This approach labels each vertex with its tree interval and computes the transitive closure of non-tree edges separately.

Apart from trees to cover the graph being indexed, alternative simple structures such as chains and paths have also been used. In a chain covering of the graph, a vertex u can reach v if both belong to the same chain and u precedes v. Jagadish (1990) presents an optimal way to cover the graph with chains in $O(n^3)$, later reduced by Chen and Chen (2008) to $O(n^2 + dn\sqrt{d})$, where d denotes the diameter of the graph. Although chain covers typically generate smaller index sizes than the interval labeling and can answer queries efficiently, they are very expensive to build for large graphs. The PathTree index, proposed recently by Jin et al. (2011), combines tree covering and path covering effectively to build an index that allows for extremely fast reachability query processing. Unfortunately, the index size can become extremely large, consuming 2 $O(np)$ space, where p denotes the number of paths in the decomposition.

Instead of indexing using covering structures, Cohen et al. (2002) introduced 2-Hop labeling which, at each vertex u, maintains a subset of the ancestors and descendants of the vertex. Using this approach, reachability queries between vertices s and t can be answered by intersecting the descendant set of s with the ancestors of

t. This technique was particularly attractive for query processing within a database system since it can be implemented efficiently using SQL-statements performing set intersections (Schenkel et al., 2004). The main hurdle in using it for large graphs turns out to be its construction – optimally selecting the subsets to label vertices with is an NP-hard problem, and no bounds on the index size can be specified. HOPI indexing proposed by Schenkel et al. (2004) tried to overcome these issues by clever engineering, using a divide-and-conquer approach for computing the covering. Jin et al. (2009) propose 3-Hop labeling, a technique that combines the idea of chain- covering with the 2-Hop strategy to reduce the index size.

4.6.2 Accelerating Online Search

From the discussion above, it is evident that accurately capturing the entire transitive closure in a manner that scales to massive size graphs remains a major challenge. Some of the recent approaches have taken a different path to utilize scalable indices that can be used to speed up traditional *online search* to answer reachability queries. In GRIPP, proposed by Trißl and Leser (2007), the index maintains only one interval per vertex on the tree cover of the graph, but some vertices reachable through non-tree edges have to be replicated.

The recently proposed GRAIL index (Yıldırım et al., 2010, 2012) uses k random trees to cover the condensed graph, generating as many (approximate) intervals to label each vertex with. As we already described in Section 4.3, the query processing proceeds by using the labels to quickly determine non-reachability, otherwise recursively querying the vertices underneath in the DAG, resulting in a worst-case query processing performance of $O(k(m+n))$. Although GRAIL was shown to be able to build indices over massive scale graphs quite efficiently, it suffers from the previously discussed drawbacks. In our experiments, we compare various aspects of our Ferrari index against GRAIL which is, until now, the only technique that deals effectively with massive graphs while satisfying a user-specified size-constraint.

4.7 Experimental Evaluation

We conducted an extensive set of experiments in order to evaluate the performance of Ferrari in comparison with the state of the art reachability indexing approaches, selected based on recent results. In this section, we present the results of our comparison with: GRAIL (Yıldırım et al., 2012), PathTree (variant PTree-1) (Jin et al., 2011), Nuutila's Intervals (Nuutila, 1996), and PWAH-8 (van Schaik and de Moor, 2011). For all the competing methods, we obtained original source code from the authors, and set the parameters as suggested in the corresponding publications.

4.7.1 Setup

Fortunately, all indexing methods are implemented using C++, making the comparisons fairly accurate without any platform-specific artifacts. All experiments were conducted using a Lenovo ThinkPad W520 notebook computer equipped with 8 Intel Core i7 CPUs at 2.30 GHz, and 16 gigabyte of main memory. The operating system in use was a 64-bit installation of Linux Mint 12 using kernel 3.0.0.22.

4.7.2 Methodology

The metrics we compare on are:

1. **Construction time** for each indexing strategy over each dataset. Since the input to all considered algorithms is always a DAG, we do not include the time for computing the condensation graph into our measurements.

2. **Query processing time** for executing 100,000 reachability queries. We consider *random* and *positive* (i. e. reachable) sets of queries and report numbers for both workloads separately.

3. **Index size** in memory that each index consumes. It should be noted that although both Ferrari and GRAIL take as input a size restriction parameter, the resulting size of the index can be quite different. PathTree, (Nuutila's) Intervals and PWAH-8 have no parameterized size, and depend entirely on the dataset characteristics.[3]

4.7.3 Datasets

We used the selection of graph datasets (Table 4.1a) that, over the recent years, has become the benchmark set for reachability indexing work. These graphs are classified based on whether they are small (with 10-100s of thousands of vertices and edges) or large (with millions of vertices and edges), and dense or sparse. We refer to the detailed description of these datasets by Yıldırım et al. (2012) and Jin et al. (2011).

We term these datasets as *benchmark datasets* and present results accordingly in Section 4.7.4.

In order to evaluate the performance of the algorithms under real-world settings, where massive-scale graphs are encountered, we use additional datasets derived from publicly available sources. These include RDF data, an online social network, and a World Wide Web crawl. To the best of our knowledge, these constitute some

[3] More precisely, the individual index sizes are computed as follows. For Ferrari, we report as index size the number of bytes necessary to represent the intervals (start and end-point as 4-bytes integers, exactness encoded as 1-byte flags), seed bitmaps (in experiments two 32-bit integers), topological order id (one integer per vertex), topological level id (1 integer per vertex), and postorder id (1 integer) i. e. overall 9 bytes/interval in addition to 20 bytes/vertex. For GRAIL we have 3 integers (12 bytes) per interval (begin, middle, end) and 4 bytes per vertex to store the topological level id. For PWAH/Interval we simply report the output size in bytes of the index as reported by the program. For PathTree, we compute the index size as 4 times the reported transitive closure size (which reports number of integers) in addition to three integers assigned to each vertex (Ruan, 2012).

Dataset	Type	$\|V\|$	$\|E\|$	Source
ArXiV	small, dense	6,000	66,707	Jin et al. (2009)
GO	small, dense	6,793	13,361	Jin et al. (2009)
Pubmed	small, dense	9,000	40,028	Jin et al. (2009)
Human	small, sparse	38,811	39,816	Jin et al. (2008)
CiteSeer[1]	large	693,947	312,282	Yıldırım et al. (2012)
Cit-Patents	large	3,774,768	16,518,947	Yıldırım et al. (2012)
CiteSeerX	large	6,540,401	15,011,260	Yıldırım et al. (2012)
GO-Uniprot	large	6,967,956	34,770,235	Yıldırım et al. (2012)

(a) Benchmark Datasets

Dataset	$\|V\|$	$\|E\|$	$\|V_C\|$	$\|E_C\|$
GovWild	8,027,601	26,072,221	8,022,880	23,652,610
YAGO2	16,375,503	32,962,379	16,375,503	25,908,132
Twitter	54,981,152	1,963,263,821	18,121,168	18,359,487
Web-UK	133,633,040	5,507,679,822	22,753,644	38,184,039

(b) Web Datasets

Table 4.1: Dataset Overview

of the largest graphs used in evaluating the effectiveness of reachability indices to this date. In the following, we briefly describe each of them, and summarize the key characteristics of these datasets in Table 4.1b.

- **GovWild** is a large RDF data collection consisting of about 26 million triples representing relations between more than 8 million entities.[4]

- **YAGO2** is another large-scale RDF dataset representing an automatically constructed knowledge graph (Hoffart et al., 2012). The version we used contained close to 33 million edges (facts) between 16.3 million vertices (entities).[5]

- The **Twitter** graph (Cha et al., 2010) is a representative of a large-scale social network. This graph, obtained from a crawl of `twitter.com`, represents the follower relationship between about 50 million users.[6]

- **Web-UK** is an example of a web graph dataset (Boldi et al., 2008). This graph contains about 133 million vertices (hosts) and 5.5 billion edges (hyperlinks).[7]

We present the results of our evaluation over these web-scale graphs in Section 4.7.5.

[1] We used the version of the files provided at `http://code.google.com/p/grail/`
[4] `http://govwild.hpi-web.de/project/govwild-project.html`
[5] `http://www.mpi-inf.mpg.de/yago-naga/yago/`
[6] `http://twitter.mpi-sws.org/`
[7] `http://law.di.unimi.it/webdata/uk-union-2006-06-2007-05/`

Dataset	Ferrari-L	Ferrari-G	Grail	PathTree	Interval	PWAH-8
ArXiV	15.84	26.62	**7.86**	4,537.39	34.54	70.10
Pubmed	14.28	24.54	**8.21**	326.54	20.35	44.41
Human	23.36	23.37	15.93	348.48	**2.70**	3.82
GO	6.48	6.91	**4.83**	89.83	5.06	8.67
CiteSeer	450.12	459.90	2,015.90	26,479.70	**251.10**	416.41
CiteSeerX	14,110.20	16,233.40	20,528.40	–	**5,808.79**	14,444.09
GO-Uniprot	26,105.90	29,611.90	34,518.40	–	**15,213.55**	26,745.61
Cit-Patents	**20,665.50**	32,366.20	21,621.70	–	–	751,984.08

(a) Index Construction Time [ms]

Dataset	Ferrari-L	Ferrari-G	Grail	PathTree	Interval	PWAH-8
ArXiV	243.86	275.33	**234.38**	338.07	1,364.99	315.24
Pubmed	**283.68**	413.06	351.56	419.03	1,523.83	358.96
Human	768.88	770.30	1,061.04	458.01	160.56	**160.22**
GO	200.37	251.01	265.35	133.30	180.58	**81.86**
CiteSeer	13,933.90	13,934.29	43,371.69	9,221.61	7,733.94	**6,723.36**
CiteSeerX	158,046.72	242,236.08	408.775.06	–	430,913.36	**152,354.44**
GO-Uniprot	429,564.04	442,301.79	435,497.25	–	774,081.33	**249,883.80**
Cit-Patents	**151,631.73**	239,609.23	235,923.00	–	–	5,462,135.76

(b) Index Size [kB]

Table 4.2: Benchmark Graphs – Indexing

4.7.4 Results over Benchmark Graphs

Tables 4.2a,b summarize the results for the selected set of benchmark graphs. In the tables, we provide the absolute values – time in milliseconds and index size in KBytes. In all the tables missing values are marked as "–" whenever a dataset could not be indexed by the corresponding strategy – either due to memory exhaustion or for taking too long to index (timeout set to 1M milliseconds). The best performing strategy for each dataset is shown in bold. For GRAIL we set the number of dimensions as suggested in (Yıldırım et al., 2012), that is, to 2 for small sparse graphs (Human), 3 for small dense graphs (ArXiV, GO, PubMed) and to 5 for the remaining large graphs. The input parameter value for Ferrari was also set correspondingly for a fair comparison.

Index Construction

Table 4.2a presents the construction time for the various algorithms. The results show that the GRAIL index can be constructed very efficiently on small graphs, irrespective of the density of the graph. On the other hand, the performance of PathTree is highly sensitive to the density of the graph as well as the size. While GRAIL and FERRARI's indexing time increases corresponding to the size of the graphs, PathTree simply failed to complete building for 3 of the larger graphs – Cit-Patents and CiteSeerX due to memory exhaustion, GO-Uniprot due to timeout.

The transitive closure compression algorithms Interval and PWAH-8 can index quite efficiently even the large graphs and their index is also surprisingly compact. A remarkable exception to this is the behavior on the Cit-Patents dataset, which

Dataset	Ferrari-L	Ferrari-G	Grail	PathTree	Interval	PWAH-8
ArXiV	23.69	13.91	100.92	**3.41**	4.17	23.22
Pubmed	7.58	4.88	12.27	**2.76**	3.16	28.58
Human	**0.78**	**0.78**	4.98	1.21	1.07	1.06
GO	4.10	2.96	4.83	**2.04**	2.47	4.45
CiteSeer	6.13	6.24	8.05	**5.01**	8.28	12.39
CiteSeerX	15.88	9.31	41.23	–	**9.27**	21.32
GO-Uniprot	28.30	28.92	**5.94**	–	16.82	48.70
Cit-Patents	778.09	**502.20**	578.83	–	–	1,514.91

(a) 100.000 Random Queries – Query Processing Time [ms]

Dataset	Ferrari-L	Ferrari-G	Grail	PathTree	Interval	PWAH-8
ArXiV	62.64	37.98	220.31	**4.94**	5.95	17.74
Pubmed	31.31	20.28	85.38	**4.42**	6.21	43.58
Human	2.08	1.96	14.48	**1.30**	1.79	6.07
GO	10.72	4.64	19.59	**2.04**	3.26	11.43
CiteSeer	13.37	13.47	85.22	**6.12**	15.17	30.60
CiteSeerX	82.76	43.06	700.49	–	**30.38**	69.21
GO-Uniprot	65.00	64.72	131.46	–	**31.76**	54.55
Cit-Patents	4,086.21	2,667.38	5,409.82	–	–	**1,739.30**

(b) 100.000 Positive Queries – Query Processing Time [ms]

Table 4.3: Benchmark Graphs – Query Processing

seems to be by far the most difficult graph for reachability indexing. The Interval index failed to process the graph within the given time limit. The related PWAH-8 algorithm finished the labeling only after around 12 minutes and ended up generating the *largest index* in all our experiments (including the indices for the Web graphs). This is rather surprising, as both algorithms were able to index the larger and denser GO-Uniprot.

When compared to other algorithms, the construction times of Ferrari-L and Ferrari-G are highly scalable and are not affected much by the variations in the density of the graph. On all graphs, Ferrari constructs the index quickly, while maintaining a competitive index size. For the challenging Cit-Patents dataset, it generates the most compact index among all the techniques considered, and very fast – amounting to a 23x-36x speedup over PWAH-8. Further – with the exception of ArXiV – Ferrari-L generates smaller indices than GRAIL and exhibits comparable indexing time on the set of benchmark graphs. With a few more clever engineering tricks (e. g., including the PWAH-8-style interval encoding), it should be possible to further reduce the size of Ferrari.

Dataset	Ferrari-L	Ferrari-G	Grail	Interval	PWAH-8
YAGO2	27,713.50	26,865.30	17,163.00	**5,844.87**	9,236.71
GovWild	12,998.80	18,045.30	**6,756.67**	15,060.55	20,703.06
Twitter	13,065.40	13,897.20	9,717.39	36,480.57	**8,219.09**
Web-UK	17,604.90	18,754.40	**12,275.90**	–	166,531.10

(a) Index Construction Time [ms]

Dataset	Ferrari-L	Ferrari-G	Grail	Interval	PWAH-8
YAGO2	372,150.70	448,139.06	447,767.66	182,962.96	**137,878.21**
GovWild	**206,475.06**	297,724.03	219,504.74	921,605.13	311,359.35
Twitter	384,049.21	384,368.44	495,500.69	**85,648.12**	97,859.81
Web-UK	616,486.63	647,050.45	622,169.95	–	**266,342.83**

(b) Index Size [kB]

Table 4.4: Web Graphs – Indexing

Query Processing

Moving on to query processing, we consider random and positive query workloads, with results depicted in Tables 4.3a and 4.3b, respectively. These results help to highlight the consistency of Ferrari in being able to efficiently process both types of queries over all varieties of graphs very efficiently. Although for small graphs PathTree is the fastest as we explained above, it cannot be applied on larger datasets. As graphs get larger, the Interval indexing turns out to be the fastest. This is not very surprising, since Interval materializes the exact transitive closure of the graph. Ferrari-G consistently provides competitive query processing times for both positive as well as random queries over all datasets. As a remarkable result of our experimental evaluation consider the CiteSeerX dataset. In this setting, the Interval index consumes almost twice as much space as the corresponding Ferrari-G index, yet is only faster by 0.04 milliseconds for random and 12.68 milliseconds for positive queries.

4.7.5 Evaluation over Web Datasets

As we already pointed out in the introduction, our goal was to develop an index that is both compact and efficient for use in many analytics tasks when the graphs are of web-scale. For this reason, we have collected graphs that amount to up to 5 billions of edges before computing the condensation graph. These graphs are of utmost importance because the resulting DAG exhibits special properties absent from previously considered benchmark datasets. In this section, we present the results of this evaluation. Due to its limited scalability, we do not use PathTree index in these experiments. Also, through initial trial experiments, we found that for GRAIL the suggested parameter value of $k = 5$ does not appear to be the optimal choice, so instead we report the results with the setting $k = 2$, which is also used for Ferrari.

Dataset	Ferrari-L	Ferrari-G	Grail	Interval	PWAH-8
YAGO2	12.00	10.95	16.56	**10.45**	12.62
GovWild	60.27	31.77	42.62	**13.33**	33.30
Twitter	**5.55**	5.65	19.27	8.66	10.32
Web-UK	**19.11**	19.29	39.21	–	20.45

(a) 100.000 Random Queries – Query Processing Time [ms]

Dataset	Ferrari-L	Ferrari-G	Grail	Interval	PWAH-8
YAGO2	59.39	38.43	97.99	**21.70**	44.19
GovWild	171.46	85.12	228.98	**29.84**	126.96
Twitter	10.24	**10.18**	76.07	18.21	36.01
Web-UK	25.54	**18.01**	95.25	–	43.73

(b) 100.000 Positive Queries – Query Processing Time [ms]

Table 4.5: Web Graphs – Query Processing

Index Construction

When we consider the index construction statistics in Tables 4.4a and 4.4b, it seems that there is no single strategy that is superior across the board. However, a careful look into these charts further emphasizes the superiority of Ferrari in terms of its consistent performance. While GRAIL can be constructed fast, its size is typically larger than the corresponding FERRARI-L index. On the other hand, PWAH-8 can take an order of magnitude more time to construct than Ferrari as well as GRAIL as we notice for Web-UK. In fact, the Interval index which is much smaller than Ferrari for Twitter and YAGO2, fails to complete within the time allotted for the Web-UK dataset. In contrast, Ferrari and GRAIL are able to handle any form of graph easily in a scalable manner.

As an additional note, the index size of Interval is sometimes smaller than Ferrari which seems to be counter-intuitive at first glance. The reason for this lies in the additional information maintained at every vertex by Ferrari, for use in early pruning heuristics. In relatively sparse datasets like Twitter and YAGO2 the overhead of this extra information tends to outweigh the gains made by interval merging. If needed, it is possible to turn off these heuristics easily to get around this problem. However, we retain them across the board to avoid dataset-specific tuning of the index.

Query Processing

Finally, we turn our attention to the query processing performance over Web-scale datasets. As the results summarized in Tables 4.5a and 4.5b demonstrate, the Ferrari variants and the Interval index provide the fastest index structures. For Web-UK and Twitter, the Ferrari variants outperform all other approaches. The performance of GRAIL, as predicted, and PWAH-8 are inferior in comparison to Interval and Ferrari-G when dealing with both random and positive query loads.

In summary, our experimental results indicate that Ferrari, in particular the global budgeted variant, is highly scalable and efficient in indexing a wide variety of graphs as well as answering queries extremely fast. This, we believe, provides a compelling reason to use Ferrari-G on a wide spectrum of graph analytics applications involving large to massive-scale graphs.

4.8 Summary

In this chapter, we presented the first contribution of this thesis: an efficient and scalable reachability index structure, Ferrari, that allows to directly control the query processing/space consumption tradeoff via a user-specified restriction on the resulting index size. The two different variants of our index allow to either specify the maximum size of the resulting vertex labels, or to impose a global size constraint which allows the dynamic allocation of budgets based on the importance of individual vertices. Ferrari assigns a mixture of exact and approximate identifier ranges to vertices so as to speed up both random as well as positive reachability queries. Using an extensive array of experiments, we demonstrated that the resulting index can scale to massive-size graphs quite easily, even when some of the state of the art indices fail to complete the construction. Ferrari provides very fast query execution, demonstrating substantial gains in processing time of both random and positive queries when compared to the previous state-of-the-art method, GRAIL.

5 Distance and Shortest Path Approximation

Q2: How strong is the relationship?
«How closely related are two genes in the metabolic network of a complex organism?»

Q3: Who participates in the relationship?
«What is the best way to establish contact with an expert on a particular problem domain?»

5.1 Problem Definition

In the previous chapter, we have presented an index structure enabling the user to quickly probe a given graph for the existence of specific relationships. In this chapter, we shift our focus towards a higher-level graph primitive and study the second algorithmic building block of this work, an index structure for the fast approximation of distances, and the computation of corresponding paths.

Distance and path queries are prevalent in many disciplines, e. g. over graphs that model interactions among proteins in bioinformatics, spatial proximity in transportation networks, relationships between entities in knowledge graphs, as well as graphs representing taxonomic hierarchies in ontologies. In each of these scenarios, computing the shortest paths between two vertices is one of the fundamental and most widely used operations.

Apart from being an important application in its own right, shortest paths between two vertices are often used as a building block in many applications, such as search result ranking (Vieira et al., 2007) and team formation (Majumder et al., 2012) in social networks, query evaluation over semi-structured data (Gou and Chirkova, 2008), analysis of microarray gene expression data (Zhou et al., 2002), and many more.
The index structure we propose in this chapter solves the following problem variants:

Problem 1: (Budgeted) Distance Estimation

Given Graph $G = (V, E)$, pair of vertices $(s, t) \in V \times V$.

Goal Estimate the **distance** (length of shortest path) from vertex s to vertex t in G (while satisfying a certain budget on online processing effort).

Problem 2: (Budgeted) Shortest-Path Approximation

Given Graph $G = (V, E)$, pair of vertices $(s, t) \in V \times V$.

Goal Identify one or more short paths from vertex s to vertex t, i. e. paths with small relative error (while satisfying a certain budget on online processing effort).

In particular, we concentrate on two important aspects of shortest-path processing. First, since one application of the proposed index structure is the use as an algorithmic building block, it is desirable to provide a direct control over the query processing time vs. accuracy tradeoff. For this purpose, our index provides a "budgeted" query processing variant. Second, in many cases it is required to compute not only one, but many short(-est) paths from source to target in order to gain insight into the structure of a relationship. We study this problem variant in detail in this chapter.

5.1.1 Problem Characteristics

Numerous variants of the shortest path problem have been addressed in the literature. On general graphs, the single-source shortest path problem is solved by the classical textbook algorithms – Breadth-First Search (Bfs) for unweighted graphs and Dijkstra's algorithm (Dijkstra, 1959) in the presence of edge weights. These algorithms exhibit asymptotic time complexities of $O(m+n)$ and $O(m+n\log m)$, respectively, where n denotes the number of vertices in the graph and m the number of edges.

While the problem of finding shortest paths is computationally lightweight, massive graphs with billions of edges render the traditional algorithms infeasible for analytic tasks that require many shortest-path calls and need to provide interactive response time to end-users. In these settings, index structures are mandatory for efficient online processing. In this chapter, we present *Path Sketch*, an index structure for massive graphs that may reside on external memory (i. e., disk or flash). Path Sketch enables rapid processing of shortest path queries by precomputing and storing selected paths in the graph. These so-called *path sketches* – stored as labels of individual vertices – are combined at query time to quickly generate approximate answers to shortest path and distance queries between pairs of vertices.

One of the main characteristics of the Path Sketch index structure is its ability to generate a large number of shortest path candidates in a budget-constrained setting, by just inspecting the index entries of the source and target vertex and potentially expanding – i. e. retrieving the adjacency lists of – a restricted number of contained vertices. In order to scale our indexing approach to massive graphs, we impose an a-priori restriction on the index size, trading off scalability for query processing accuracy. As a result, the returned paths connecting the query vertices are possibly longer than the true shortest path. However, as shown in the experimental evaluation of this chapter, the approximation quality of our algorithms is very good, with running times in the order of milliseconds over massive graphs. Our index structure allows to compute approximate shortest paths with a specified budget on the number of vertex random accesses to the graph. This property provides direct control over the tradeoff between approximation quality (or number of generated paths), and query processing time. The Path Sketch index is mainly targeted towards the class of graphs that exhibit a small effective diameter, a property that is prevalent especially for the case of online social networks.

5.1.2 Contribution

We build on the idea of selecting certain *seed vertices* from the graph and precomputing an index structure based on the shortest paths connecting these designated seeds with the remaining vertices of the graph (Das Sarma et al., 2010). However, previously proposed methods limited themselves to estimating the distances between vertices (Das Sarma et al., 2010, Potamias et al., 2009). In our first prototype of Path Sketch (Gubichev et al., 2010), we have advanced this prior work to the next level of returning actual paths (not just distances), enabling the generation of multiple short paths connecting the query vertices. This generalization not only improves the estimation accuracy of distance queries, but also solves the more challenging problem of quickly synthesizing the actual paths from source to target corresponding to the estimated vertex distance.

This chapter emphasizes the problem of quickly computing multiple approximate shortest paths (abbreviated ASP), given a budget on the number of random accesses to the graph at query processing time. Downstream applications often require multiple paths, say top-k shortest paths or several approximately shortest paths. Our method allows controlling the tradeoff between query processing time on one hand, and number of paths as well as approximation quality on the other hand on a per-query basis. This is especially relevant when shortest paths are used as a building block in higher-order graph analytics. We also address the problem variant of generating paths that satisfy certain structural constraints: computing paths that do or do not pass through vertices of certain types (i. e., with specific types of vertex labels).

In summary, this chapter makes the following technical contributions to advance our previously proposed prototype of the Path Sketch index:

1. methods for fast index construction by integrating previously proposed, optimized Bfs algorithms with the simultaneous computation of multiple trees, together with a combination of effective index compression techniques,

2. an algorithm for fast, budget-constrained query processing, providing a direct control over the tradeoff between query processing time and accuracy,

3. discussion of novel problem scenarios such as the generation of restricted as well as multiple diverse paths.

The efficiency and output quality of our methods is demonstrated by a comprehensive set of experimental results with real-world graphs from various domains, including web graphs and social networks, with hundreds of millions of vertices and billions of edges. The full source code of the Path Sketch software is publicly available under the Creative Commons CC-BY 3.0 license at the URL

`http://mpi-inf.mpg.de/~sseufert/pathsketch.`

5.1.3 Extension over Previous Work

In an early version of the Path Sketch index (Gubichev et al., 2010) – which is not a contribution claimed by this thesis – we introduced the notion of path sketches and associated techniques for shortest path approximation. In this chapter, we develop several major improvements over this prior work, making the index space-efficient and index construction scalable. We also present a new query processing framework, based on the optimized index structure, that gives faster query processing compared to the original implementation of Gubichev et al. (2010). Finally, we extend the index structure to allow for querying in the presence of processing budget constraints (limiting the number of vertex expansions at query time), and develop support for type restrictions in path computations.

5.1.4 Organization

The remainder of this chapter is organized as follows. In Section 5.2, we introduce the basic principle of seed-based distance estimation. The basic concept of our index structure, the notion of path sketches and their use in shortest path approximation, is presented in Section 5.3. We discuss the extension of the query processing framework to support approximate shortest path (ASP) queries in a budgeted online processing setting in Section 5.4. In Section 5.5, we discuss modifications to our index that enable the computation of restricted as well as multiple, diverse paths. We discuss our algorithms for efficient index construction in Section 5.6, followed by the description of the physical index layout in Section 5.7. A review of previous work on shortest path computation and related problems is presented in Section 5.8. In Section 5.9, we present the experimental evaluation of our techniques, followed by concluding remarks and a discussion of future work in Section 5.10.

5.2 Seed-Based Distance Estimation

The seed-based estimation of vertex distances in graph data has been an active topic of research over recent years (Das Sarma et al., 2010, Kleinberg et al., 2004, Potamias et al., 2009, Qiao et al., 2012). In this setting, a set of designated *seed* vertices, also referred to as *landmarks*, is selected, and every vertex is assigned a vector of distances to and from the seeds. In this so-called *embedding* technique, all vertices are anchored with respect to the seed vertices, similar to a coordinate system. In order to estimate the distance from vertex s to vertex t, one can compute an approximate solution as

$$\tilde{d}(s,t) = \min_{\sigma \in S} d(s,\sigma) + d(\sigma,t), \tag{5.1}$$

where S denotes the set of seed vertices. The estimated distance then corresponds to the length of the path obtained by concatenating the paths from s to the seed vertex σ and from σ to the target vertex t, respectively. It is important to note that the distance estimate $\tilde{d}(s,t)$ is an upper bound on the true distance, that is, the distance is never underestimated. For a simple proof, consider that the graph contains an actual path from s to t of length $\tilde{d}(s,t)$: the path $p_{s \to \sigma^* \to t}$, passing through the seed vertex

$$\sigma^* = \arg\min_{\sigma \in S} d(s,\sigma) + d(\sigma,t). \tag{5.2}$$

In general, the distances in a graph follow the triangle inequality:

$$d(s,t) \leq d(s,\sigma) + d(\sigma,t) \quad \forall(\sigma \in V). \tag{5.3}$$

Since the distances from and to the seeds can be precomputed, a distance estimation algorithm based on Equation (5.1) allows for an extremely efficient processing of distance queries. This framework is especially attractive for massive graphs, which potentially reside on secondary storage. In these settings, online-search methods like Breadth-First Search or Dijkstra's algorithm severely suffer from the high cost incurred by the required vertex expansions:

Definition 5.1 (Vertex Expansion). *Let $v \in V$ denote a vertex. The operation of retrieving the adjacency list $\mathcal{N}^+(v)$ of v is called **vertex expansion**.*

When the graph is stored in main memory, a vertex expansion is equivalent to a memory reference. For graphs stored on disk, a vertex expansion corresponds to a random access to the hard disk, which is a costly operation especially for the case of rotational drives.

The seed-based distance estimation framework stores the distance vectors of a vertex (comprising the distances between the vertex and the seeds) as vertex labels of an index structure. As a result, two lookups in the index are sufficient in order to estimate the distance. The index entries (vertex labels) in this context are referred to as *distance sketches*:

Definition 5.2 (Distance Sketch).
Let $v \in V$ denote a vertex and $S = \{\sigma_1, \ldots, \sigma_k\} \subseteq V$ a set of seed vertices. The sets

$$\ell^+(v) = \{(\sigma_1, d(v, \sigma_1)), \ldots, (\sigma_k, d(v, \sigma_k))\}, \tag{5.4}$$

$$\ell^-(v) = \{(\sigma_1, d(\sigma_1, v)), \ldots, (\sigma_k, d(\sigma_k, v))\}, \tag{5.5}$$

*are called outgoing and incoming **distance sketches** of v.*

The distance estimate can be computed easily by stepping itemwise through the distance sketches of source and target vertex, given that the entries in a distance sketch are kept in order.

Seed Selection. An integral part of a seed-based distance estimation algorithm is the actual selection of seed vertices. Several approaches have been proposed in the literature: A natural and efficient choice is the selection of k randomly sampled vertices as seeds. More elaborate strategies select vertices based on structural properties, such as degree or centrality scores. As remarked by Potamias et al. (2009), the optimal choice for a single seed is the selection of the vertex with highest betweenness centrality, which corresponds to the fraction of pairwise shortest paths passing through a vertex. The optimal selection of seeds is generally a hard problem (Potamias et al. (2009) show NP-hardness for a slightly modified betweenness centrality measure). Thus, in practice, seeds are selected by various heuristics. Apart from the vertex degree mentioned above, selection based on closeness centrality (which is much easier to approximate than betweenness centrality) as well as partitioning-based strategies have been studied (Potamias et al., 2009). Goldberg and Harrelson (2004) use a greedy selection algorithm that repeatedly picks the vertex that is farthest (in terms of minimum distance) from the set of already selected seeds.
Das Sarma et al. (2010) propose a set-based seed selection strategy. In this setting, instead of repeatedly computing the distances from and to a single seed vertex, *sets* of seed vertices are used. Every vertex in the graph then records the distance from

and to the *closest* seed vertex in the set. In their approach, Das Sarma et al. use randomly sampled seed sets $S_1, S_2, \dots, S_K$ of exponentially increasing size $|S_i| = 2^{i-1}$ in order to derive an upper bound on the approximation error.

In the next section, we discuss our index structure for shortest path approximation, which extends the seed-based distance estimation framework to synthesize the actual paths connecting the query vertices, and obtain more accurate distance estimates.

5.3 The Path Sketch Index

5.3.1 Basic Idea

The basic principle underlying the Path Sketch index structure is the materialization of actual *paths*, rather than mere distances, from and to the seed vertices.The resulting index entries (sets of paths) we assign to each vertex are referred to as *path sketches*:

Definition 5.3 (Path Sketch).
Let $v \in V$ denote a vertex and $S = \{\sigma_1, \dots, \sigma_k\} \subseteq V$ a set of seed vertices, and $p_{v \to w}$ a path from v to w. The sets

$$\ell^+(v) = \{(\sigma_1, p_{v\to\sigma_1}), \dots, (\sigma_k, p_{v\to\sigma_k})\}, \tag{5.6}$$

$$\ell^-(v) = \{(\sigma_1, p_{\sigma_1\to v}), \dots, (\sigma_k, p_{\sigma_k\to v})\}, \tag{5.7}$$

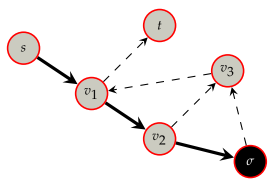
*are called outgoing and incoming **path sketches** of v. In contrast to a distance sketch, we store for a vertex v* paths *from v to its k associated seeds in the* outgoing path sketch *$\ell^+(v)$, and the paths from the k seeds to vertex v in the* incoming path sketch, *$\ell^-(v)$.*

In the remainder of this section and the subsequent Section 5.3.2, we discuss the basic query processing over path sketches, as proposed by Gubichev et al. (2010). Let $(s, t) \in V^2$ denote a pair of query vertices. We want to compute an ASP from vertex s to vertex t. For this purpose, the Path Sketch query processing algorithm obtains the outgoing path sketch $\ell^+(s)$ of the source s, and the incoming path sketch $\ell^-(t)$ of the target t from the index. Recall that $\ell^+(s)$ consists of a set of k paths connecting s to k seed vertices, and $\ell^-(t)$ consists of a set of k paths connecting k seed vertices with t.

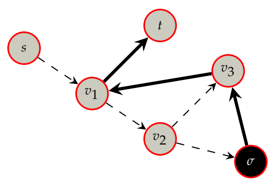
Similar to the distance estimation based on Equation (5.1), we obtain a first approximate solution by simply concatenating all paths sharing a common endpoint seed, i. e.

$$\begin{aligned} &\tilde{d}(s,t) = \|p_{s\to\sigma^*} \circ p_{\sigma^*\to t}\| \\ \text{with} \quad &\sigma^* = \operatorname*{arg\,min}_{\sigma\in S} \|p_{s\to\sigma} \circ p_{\sigma\to t}\|. \end{aligned} \tag{5.8}$$

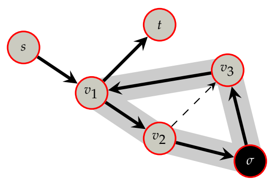
This way, we can provide the distance estimate, $\tilde{d}(s,t)$, as well as the corresponding ASP $p_{s\to\sigma^*} \circ p_{\sigma^*\to t}$ as the solution.



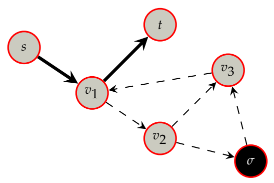
(a) Path from s to σ



(b) Path from σ to t



(c) Concatenated path from s to t through σ



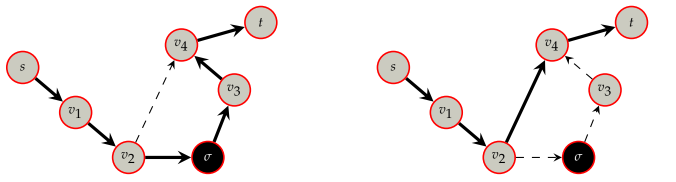
(d) Final path from s to t after cycle elimination

Figure 5.1: Example: Cycle Elimination

For undirected, connected graphs, this approach guarantees to find at least one path connecting the query vertices. For directed graphs, it is possible that the outgoing path sketch of the source does not share a seed vertex with the incoming path sketch of the target. In order to guarantee that a path will be found if the target is reachable from the source in the input graph, additional steps are required. For vertices that belong to the same strongly connected component (SCC), it suffices to pick a seed vertex from the same SCC. For pairs of vertices from different SCCs one approach is the extension of the vertex labels with reachability information as in the Ferrari index presented in the previous chapter. Then, whenever that target is reachable from the source but the path sketch intersection is empty, we can combine our approach with an online algorithm. However, in our experimental evaluation we demonstrate that in the vast majority of cases a path between source and target can be generated by the Path Sketch index.

5.3.2 Improving Estimation Accuracy

Apart from enabling the computation of actual paths, how can the information about intermediate vertices, which is explicitly encoded in the path sketches, lead to a better approximation quality of the query processing algorithm? In the following, we present two concrete ideas that exploit the additional information towards this goal:



(a) Path from s to t. The original graph contains the edge (v_2, v_4)

(b) Path from s to t after shortcutting

Figure 5.2: Example: Shortcutting

(1) Cycle Elimination. Eq. (5.8) displays the basic distance estimation scheme. However, by the use of path sketches rather than distance sketches, we now have knowledge about the individual vertices contained in the paths. This can be exploited in order to identify shorter paths from source to target. First, observe that each pair of paths $p_{s \to \sigma} \in \ell^+(s)$, $p_{\sigma \to t} \in \ell^-(t)$ sharing a common seed endpoint, might have more vertices in common than just σ. The information about intermediate vertices allows us to obtain a shorter path by removing cycles that are inevitably present in the concatenated path $p_{s \to \sigma} \circ p_{\sigma \to t}$ if the constituent paths $p_{s \to \sigma}, p_{\sigma \to t}$ share more than one vertex.

Example. Consider the paths depicted in Figure 5.1: Suppose $\sigma \in V$ is a common seed for the vertices $s, t \in V$, and the path sketches $\ell^+(s)$ and $\ell^-(t)$ contain the paths $p_{s \to \sigma} = (s, v_1, v_2, \sigma)$ and $p_{\sigma \to t} = (\sigma, v_3, v_1, t)$, respectively. The concatenated $s - t$-path of length 6 is given by

$$p_{s \to \sigma \to t} = p_{s \to \sigma} \circ p_{\sigma \to t} = (s, v_1, v_2, \sigma, v_3, v_1, t).$$

We can obtain a shorter path (and thus better distance estimate) by removing the cyclic subpath $(v_1, v_2, \sigma, v_3, v_1)$, which is highlighted in Figure 5.1(c), thus obtaining the path (s, v_1, t) of length 2.

(2) Shortcutting. As in the previous example, we denote by $p_{s \to \sigma \to t} = p_{s \to \sigma} \circ p_{\sigma \to t}$ the concatenated path from s to t via the common seed vertex σ. Two vertices u, v in the path might actually have a closer connection than the one provided by the respective subpath of $p_{s \to \sigma \to t}$. Consider the following example:

Example. In Figure 5.2, the vertices v_2 and v_4 are connected by (v_2, σ, v_3, v_4), a subpath of $p_{s \to \sigma \to t}$ of length 3. However, this is not a shortest path from v_2 to v_4, since the input graph contains the edge (v_2, v_4). We can thus improve the approximated short path by substituting the subpath from v_2 to v_4 in $p_{s \to \sigma \to t}$ by the single edge (v_2, v_4), obtaining the path (s, v_1, v_2, v_4, t). Thus, the estimated distance is improved from 6 to 4.

Note that since the paths $p_{s \to \sigma}$ and $p_{\sigma \to t}$ are already shortest paths, potential shortcuts can only exist between vertices u, v such that $u \in p_{s \to \sigma}$ and $v \in p_{\sigma \to t}$. In order to improve a pair of paths by detecting shortcuts, it suffices to check for every inner vertex $v \in V(p_{s\to\sigma}) \setminus \{s, \sigma\}$ of the path $p_{s\to\sigma}$ whether it is directly connected to any vertex of the path $p_{\sigma\to t}$. Alternatively, we can check whether any inner vertex of the path $p_{\sigma\to t}$ has a direct connection to a vertex in $p_{s\to\sigma}$. Thus, in order to detect shortcuts it is necessary to perform at most $\min\{\|p_{s\to\sigma}\|, \|p_{\sigma\to t}\|\} - 2$ vertex expansions, i. e. obtaining the adjacency lists of inner vertices and intersecting them with the other path.

5.3.3 Path Sketch Query Processing

In the examples above, we considered individual pairs of paths $p_{s\to\sigma}, p_{\sigma\to t}$ sharing a common seed, with a pairwise application of cycle elimination and shortcut detection. However, combining the paths associated with the source s and target t into respective *tree structures* opens up additional possibilities for query processing:

Tree Representation of Path Sketches

As before, let $\ell^+(s)$ denote the outgoing path sketch of s. Conceptually, $\ell^+(s)$ consists of k shortest paths $p_{s\to\sigma_i^+}$, $1 \leq i \leq k$. Since all these paths share the source vertex s as common endpoint, the union of all paths in $\ell^+(s)$ yields a tree-structured subgraph $T^+(s)$ of G, rooted at s:

Definition 5.4 (Path Sketch Tree).
Let $s \in V$ denote a vertex and $p_{s\to\sigma_1^+}, \ldots, p_{s\to\sigma_k^+}$ shortest paths originating at s and ending at the seed vertices $\sigma_1^+, \ldots, \sigma_k^+$. We define the (outgoing) path sketch tree *of v as the union of the paths*

$$T^+(s) := p_{s\to\sigma_1^+} \cup p_{s\to\sigma_2^+} \cup \ldots \cup p_{s\to\sigma_k^+}, \tag{5.9}$$

yielding a tree-structured subgraph rooted at v.

Analogously, for the target vertex t with the incoming path sketch $\ell^-(t)$, we obtain an “inverted” tree structure (i. e. a tree on reversed edges), denoted by $T^-(t)$. We remark that the union of the paths in a sketch can yield a DAG-structured subgraph. However, we enforce tree structure by discarding all cross-edges in the union of the paths. This has no negative impact on the approximation quality of our algorithms, which only depends on the set of vertices included in the sketches and the property that all vertices in a sketch are connected via a shortest path to the root vertex.

We now discuss how the aggregated representation of the index entries as tree structures allows for a more effective query processing approach than operating merely on the level of pairs of individual paths.

Shortest-Path Approximation over Path Sketch Trees

For a pair of query vertices (s,t), shortest path approximation with path sketch trees works as follows.

Step 1 The outgoing path sketch $\ell^+(s)$ of the source, and the incoming path sketch $\ell^-(t)$ of the target are obtained from the index, followed by the construction of the respective path sketch trees $T^+(s)$ and $T^-(t)$.

Step 2 In order to identify better (shorter) connections from s to t than provided by the paths passing through the common seeds, we conduct a bidirectional breadth-first search over *the trees*, $T^+(s)$ and $T^-(t)$. More precisely, we start two BFS processes, one rooted at the source $s \in T^+(s)$ and operating on outgoing edges, and one rooted at the target $t \in T^-(t)$, operating on incoming edges. In contrast to a regular bidirectional search, we only expand vertices that are contained in the respective path sketch trees. When the first common vertex v is discovered in the interleaved BFS expansions, we can either terminate the algorithm and return the path $p_{s\to v\to t}$ as solution, or finish the expansion of the current level in either of the trees in order find a better path (this can be the case if there exists a connection across the fringe of the two BFS processes). Optionally, in order to generate more paths, we can terminate the algorithm only after all vertices in the two path sketch trees have been expanded.

This query processing approach of bidirectional search, restricted to the expansion of only the vertices contained in the path sketch trees, can lead to a suboptimal solution. The resulting path can be longer than the true shortest path from s to t. This is the case when the shortest path passes through an intermediate vertex not present in the common 1-hop neighborhood (the set of vertices directly connected to vertices in both path sketch trees) of the path sketch trees. In the worst case, this common 1-hop neighborhood can even be empty, in which case no approximate path can be found, even though the input graph contains a path from s to t. However, as we will show in our experimental evaluation, in practice we obtain a very accurate distance and path approximation and fail to find a path only in a small fraction of cases. At the same time we perform much fewer vertex expansions (translating to random disk accesses for external-memory graphs), when compared to regular (one- or bidirectional) BFS.

5.3.4 Computational Complexity

We can derive a theoretical bound on the index size as follows: let k denote the number of seeds we use for index construction. The number of paths that are stored in the index entry of a vertex v is thus upper-bounded by k. Furthermore, since the length of each path is upper-bounded by the diameter of the graph, the total size of our index structure is bounded by $O(2n \cdot k \cdot \text{diam}(G))$. In practice, many of the vertex-seed paths in the union will overlap – i. e. share common subpaths – and thus the true index size is much smaller.

Since the diameter of the graph directly affects the space complexity, the proposed techniques are applicable mainly for graphs that exhibit a small diameter. However, many of the most interesting graph structures found in current real-world applications satisfy this property, most notably social network structures and web graphs.
Regarding query processing complexity, the bottleneck of the algorithm lies in the vertex expansions, i. e. the accesses to the graph in order to obtain the respective adjacency lists – which is especially costly for external-memory graphs – and the ensuing online processing effort. The number of such vertex expansions is bounded by the number of vertices in the path sketch trees, i. e. $O(2k \cdot \text{diam}(G))$.

In the next section, we discuss how the Path Sketch query processing algorithm can be extended to satisfy an online processing budget, i. e. handle the case where a certain number of vertex expansions (that may be issued during query processing) is specified by the user.

5.4 Budgeted Query Processing

When using approximate shortest paths (ASP) computation as a building block in graph analytics, it is desirable to directly control the tradeoff between query execution time and result quality, in order to adapt to a wide range of problem scenarios. For this purpose, we discuss a variant of query processing over path sketch trees where only a certain budget β of vertex expansions is available, and these expansion have to be carefully allocated to the vertices in the path sketch trees.

In this section, we describe the modified query processing algorithm for this setting. Our approach is based on (i) estimating an upper bound of the distance from s to t, and (ii) carefully selecting the most "promising" vertices for expansion in the path sketch trees, based on the initial distance estimate as well as structural properties of the vertices. Thus, rather than performing bidirectional Bfs over the trees, we only select a subset of the vertices for expansion and determine the order of expansion based on certain "features" of the vertices (not necessarily corresponding to distance to the respective root).

We now discuss how a first distance estimate can be obtained:

(i) Initial Distance Estimate. A first upper bound on the distance $d(s,t)$ from s to t is obtained from the *intersection*

$$S = V\big(T^{+}(s)\big) \cap V\big(T^{-}(t)\big) \tag{5.10}$$

of the vertex sets of the two path sketch trees. This intersection can be computed immediately after retrieving the respective index entries. The set S consists of all vertices present in both path sketch trees, and is thus a superset of the common seed vertices. Since all subpaths of a shortest path are shortest paths themselves, the vertices in S that are not designated seed vertices can be regarded as additional seeds, since the true distance from s to every vertex in $\sigma \in S$ as well as the distance from every $\sigma \in S$ to t is known.

The initial distance estimate, denoted by $\overline{D}$, is obtained by

$$\begin{aligned}\overline{D} &= \min_{\sigma \in S}\ d(s,\sigma) + d(\sigma,t)\\ &= \min_{\sigma \in S}\ \text{depth}_{T^{+}(s)}(\sigma) + \text{depth}_{T^{-}(t)}(\sigma).\end{aligned} \tag{5.11}$$

Note that, by applying the same reasoning as in Section 5.2, this estimate is an upper bound on the true distance.

This first estimation scheme can be regarded as an extension of cycle elimination. For the example depicted in Fig. 5.1, the common vertex v_1 is an additional seed for the query pair (s,t). Thus, the distance can be estimated using path (s, v_1, t) via v_1 rather than σ, which yields an estimate equivalent to cycle elimination. However, note that the estimation scheme of Equation (5.11) is more powerful, since it essentially allows to obtain a distance estimate from a pair of paths $p_{s\to\sigma}, p_{\sigma'\to t}$ with $\sigma \neq \sigma'$, i. e. paths that *do not* share a global seed, but still have a non-empty intersection.

(ii) Restricted Expansion. Having obtained the initial distance estimate, the goal of the second phase of the query processing algorithm is the identification of potential shortcuts, similar to the bidirectional Bfs over the path sketch trees, but adhering to budget constraints. For this purpose, we selectively expand certain vertices in the path sketch trees in order to find a shorter connection from $T^{+}(s)$ to $T^{-}(t)$, than the one corresponding to the initial distance estimate, $\overline{D}$.
Formally, an edge $(v,w), v \neq w$ provides a shortcut from path sketch tree $T^{+}(s)$ to $T^{-}(t)$ with respect to the estimated distance $\overline{D}$, if it holds

$$\text{depth}_{T^{+}(s)}(v) + 1 + \text{depth}_{T^{-}(t)}(w) < \overline{D}. \tag{5.12}$$

Here, the left-hand value corresponds to the length of the tree path in $T^{+}(s)$ from s to v, followed by the edge (v,w), and the tree path from w to t in $T^{-}(t)$. For shortcut detection, vertices from both the incoming as well as the outgoing path sketch tree of the query vertices can be expanded, i. e. the set of successors/predecessors is retrieved in a random access to the graph. W. l. o. g., for the outgoing tree, if any of the neighbors $w \in \mathcal{N}^{+}(v)$ is contained in the tree $T^{-}(t)$, we obtain a new shortest path candidate: the concatenated path $p = p_{s\to v} \circ (v,w) \circ p_{w\to t}$. If it holds $\|p\| < \overline{D}$, we have identified a shortcut and the distance estimate $\overline{D}$ can be improved (i. e. reduced) accordingly.

The expansion of a vertex on the incoming tree $T^-(t)$ is performed in a similar way, with the difference that for the current vertex $v \in V\big(T^-(t)\big)$, we determine the neighbors of the vertex to be the vertices $\mathcal{N}^-(v)$, that is, the vertices of the graph with an edge *to* vertex v. Note that, just like in the standard query processing algorithm, only vertices that are contained in the original tree $T^+(s), T^-(t)$ are expanded. For each expanded vertex its successors/predecessors are added to the tree and can thus be discovered from the respective other tree, but will not be expanded themselves.

The expansions of vertices in the path sketch trees continue until either no shorter path can be found (a sufficient termination criterion is discussed in the subsequent section) or the specified expansion budget is exhausted. In both cases, the query processing algorithm terminates and the last discovered path $p_{s\to t}$ with $\|p_{s\to t}\| = \overline{D}$ is returned as the best path, with the previously discovered paths as additional solutions.

We now discuss how to select vertices from the path sketch trees for expansion.

(iii) Avoiding Unnecessary Expansions. While some vertices can provide shortcuts between the path sketch trees, there are likewise vertices for which we can reason that, due to certain structural properties, no such shortcut can exist and it is thus not necessary to expand these vertices. We will discuss several such properties for the case of the outgoing path sketch trees $T^+(s)$. All results hold analogously for the incoming path sketch trees $T^-(t)$.
The following classes of vertices do not need to be considered during query processing, since their expansion cannot provide a shorter path:

Common Vertices. For a vertex $v \in S = V\big(T^+(s)\big) \cap V\big(T^-(t)\big)$ present in both path sketch trees $T^+(s), T^-(t)$, the path $p_{v\to t}$ is already a shortest path and thus no shortcut exists from v to $T^-(t)$.

Deep Vertices. Let $\overline{D}$ denote the current upper bound on the distance from s to t (e. g. the initial distance estimate), that is potentially further improved over the course of the algorithm. A vertex $v \in V\big(T^+(s)\big)$ with $\text{depth}_{T^+(s)}(v) \geq \overline{D} - 1$ can not provide a shortcut since the minimum length of a path from s to t passing through v is given by

$$\|p_{s\to v\to t}\| \geq \text{depth}_{T^+(s)}(v) + 1 \geq \overline{D}. \tag{5.13}$$

We can integrate this criterion, which is also used as termination criterion for the plain bidirectional Bfs search, with the information about the current stage of expansion in the opposite (incoming) path sketch tree, $T^-(t)$. Suppose we have expanded all vertices up to level l in $T^-(t)$, and no shortcut from $T^-(t)$ to the current vertex $v \in V\big(T^+(s)\big)$ has been discovered. In this case, the vertex v in $T^+(s)$ does not provide a shortcut if it holds

$$\text{depth}_{T^+(s)}(v) \geq \overline{D} - (l+2). \tag{5.14}$$

Shallow Vertices. Path sketches can also help to avoid unnecessary expansion of vertices that are *too close* to the source vertex s. Consider a vertex $w \in V$ for which the distances $d(s,w)$ as well as $d(t,w)$ are known. By the triangle inequality it holds

$$d(s,w) \leq d(s,t) + d(t,w) \;\Leftrightarrow\; d(s,t) \geq d(s,w) - d(t,w).$$

We can use this property to compute a lower bound on the true distance $d(s,t)$ by

$$d(s,t) \;\geq\; \underline{D} := \max_{\sigma \in S} \text{depth}_{T^+(s)}(\sigma) - \text{depth}_{T^+(t)}(\sigma),$$

where $S = V\big(T^+(s)\big) \cap \big(\big(T^+(t)\big)$ denotes the common vertices in $T^+(s)$ and $T^+(t)$. Note that we need to obtain *both outgoing path sketch trees* in order to compute this lower bound. We can use this result to avoid the unnecessary expansion of vertices $v \in V\big(T^+(s)\big)$ for which it holds

$$\text{depth}_{T^+(s)}(v) \leq \underline{D} - 1. \tag{5.15}$$

The above criteria can help to avoid the expansion of vertices that cannot provide a shortcut.

We briefly discuss additional strategies that can help further avoid expansion of certain vertices. The highly skewed degree distribution is a well-known property of power law graphs. While in these graphs few vertices exist with very high degree, the vast majority of the vertices exhibits a very small degree. In order to avoid expansions of these low-degree vertices, we can – at index construction time – fix a constant $c \geq 1$ and include the successors of all vertices $v \in T^+(s)$ with $\delta^+(v) \leq c$ in the path sketch tree $T^+(s)$ (accordingly for the predecessors of $v \in T^-(t)$). Then, since the direct neighbors of all low-degree vertices in the path sketch tree are already known, this approach can further reduce the number of candidates for expansion while bloating the index size by no more than the constant factor c.

A second approach considers the use of probabilistic, memory-resident set data structures without false negatives, such as Bloom filters (Bloom, 1970), in order to represent the neighborhood of the individual vertices. More precisely, we precompute a probabilistic representation of the set of successors (predecessors) for the individual vertices, each consuming not more than a fixed amount of $B = \lfloor M/(2n) \rfloor$ bits, given a main memory restriction of M bits. Then, we can avoid the expansion of vertex $v \in V\big(T^+(s)\big)$ if set membership tests are unsuccessful for all vertices of interest in $T^-(t)$.

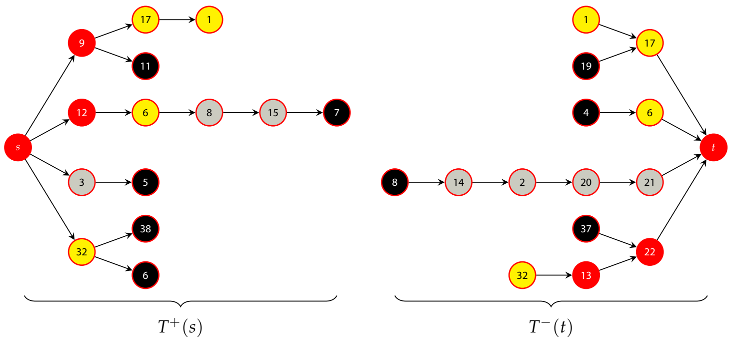


Figure 5.3: Example: Outgoing and incoming path sketch tree

(iv) Expansion Heuristics. Regarding the selection of the most promising vertices for expansion, consider the following heuristic, that restricts vertex expansions to selected *branches* of the sketch trees. As before, let S denote the set of common vertices in the trees $T^+(s), T^-(t)$. For each vertex $v \in S$, we follow the path from v up to the root of the tree, i. e. s in the outgoing tree and t in the incoming tree. All vertices along the way are marked as (in principle) *expandable*. Then, for shortcut detection, we consider only vertices for expansion that have been marked. The intuition is that shortcuts between the sketch trees are more likely to appear in parts of the tree for which a connection is already known to exist.

Example. For a pair of query vertices (s, t), the respective outgoing and incoming trees, $T^+(s)$ and $T^-(t)$ are depicted in Figure 5.3. In the figure we assign a numeric identifier to every vertex in order to be able to identify which vertices are present in both trees. The leaves of the trees are the seed vertices. In the figure, vertices in the intersection of the trees are shown in yellow color (vertices with ids 1,6,17,32). Further, we mark the vertices on the paths from the root of the tree to vertices in the intersection with red color. Using above heuristic only the yellow and red vertices will be expanded.

We propose the following strategies for determining the order in which (depending on the setting only the marked or all except for deep, shallow, etc.) tree vertices are selected as candidates for expansion:

Vertex Degree (VD) Expand the β candidate vertices with highest degree, since a larger number of neighbors increases the probability of detecting a shortcut as well as additional paths.

Vertex Level (VL) Expand the β candidate vertices that are located closest to the root, since shortcuts found in lower levels are more valuable in terms of relative error of the returned solution. This selection criterion will lead to a search process similar to bidirectional Bfs that is terminated as soon as the fixed budget has been spent.

The mentioned approaches and strategies are combined to obtain a budgeted query processing algorithm: first, only vertices that can provide a shortcut can be

Algorithm 6: BudgetedQuery(s, t, β)

```
Data: $s$: source vertex, $t$: target vertex, $\beta$: vertex expansion budget

▷ load index entries
$T^{+}(s) \leftarrow$ LoadOutgoingSketch$(s)$
$T^{-}(t) \leftarrow$ LoadIncomingSketch$(t)$

▷ initial distance estimate
$\overline{D} \leftarrow \min_{\sigma \in V(T^{+}(s) \cap V(T^{-}(t))} \mathrm{depth}_{T^{+}(s)}(\sigma) + \mathrm{depth}_{T^{-}(t)}(\sigma)$

▷ create paths through intersecting vertices
$P \leftarrow \emptyset$
foreach $\sigma \in V(T^{+}(s) \cap V(T^{-}(t))$ do
    $P \leftarrow P \cup \left(p_{s \to \sigma} \circ p_{\sigma \to t}\right)$

▷ order vertices for expansion
$X \leftarrow$ OrderVertices$(T^{+}(s), T^{-}(t))$

expanded $\leftarrow 0$
while expanded $< \beta$ do
    $x \leftarrow$ Pop$(X)$
    $N(x) \leftarrow$ Expand$(x)$
    expanded $\leftarrow$ expanded $+ 1$
    for $y \in N(x)$ do
        if $x$ came from $T^{+}(s)$ then
            if $y \in T^{-}(t)$ then
                $P \leftarrow P \cup \left(p_{s \to x} \circ (x, y) \circ p_{y \to t}\right)$
            if $y \notin V(T^{+}(s))$ then
                AddEdge$(T^{+}(s), (x, y))$
        else
            if $y \in T^{+}(s)$ then
                $P \leftarrow P \cup \left(p_{s \to y} \circ (y, x) \circ p_{x \to t}\right)$
            if $y \notin V(T^{-}(t))$ then
                AddEdge$(T^{-}(t), (y, x))$

return $P$
```

considered for expansion (i. e. no shallow, deep, and seed vertices). Then, at most β of the remaining vertices are considered for expansion, the order of which is governed by the expansion strategy, i. e. either by degree or distance from the respective root.

In our experimental evaluation we compare the results and tradeoffs of the different expansion strategies over a variety of expansion budgets.

In Algorithm 6 we present the (simplified) pseudocode for query processing. In the algorithm we use the procedures OrderVertices to determine the order of expansion of the vertices in the trees (in this procedure we order candidates for expansion based on the respective criteria, i. e. distance from root or degree, alternating between candidates from the source and target trees to simulate bidirectional search). The procedure Expand retrieves the adjacency list (incoming or outgoing edges) from the graph. When a vertex is expanded, the neighbors will be added to the path sketch tree in order to find connections to these new vertices in a vertex expansion from the other tree. However, these newly added vertices will not be expanded themselves; this can only happen for vertices that were initially present in the trees.

Regarding the complexity of the proposed query processing algorithms, the simplest variant returns the path obtained after the tree intersection (corresponding to the initial distance estimate) and requires exactly two accesses to the index structure and no vertex expansion. The time complexity of this step corresponds to the complexity of computing the set intersection, given by $O(k \cdot \text{diam}(G))$. Clearly, for the budgeted query processing variant, the number of vertex expansions is bounded by β. Each expansion results in additional time complexity of $O(n)$ for checking containment of each neighbor in the other tree.

5.5 Restricted and Diverse Paths

In this section, we study two important problem variants: the computation of restricted as well as diverse paths. We discuss how the Path Sketch index structure can be extended to process such queries.

5.5.1 Computing Restricted Paths

Many problem scenarios require imposing certain restrictions on paths to obtain semantically meaningful results. We specifically address graphs with *vertex types*, specified by the tuple $G = (V, E, T, \tau)$, where T denotes a set of *types* and $\tau : V \rightarrow 2^T$ assigns each vertex to a subset of the available types. Consider the following scenarios:

Relatedness of Wikipedia Articles.

In an application over the Wikipedia encyclopedia, shortest paths between two articles are computed in order to give insight into the degree of relatedness of the respective concepts. In this setting, it is desirable to exclude paths via non-informative pages of administrative types, for example, connections through lists (two persons are connected via a path of length 2 passing through the list *Living People*).

Relationship Analysis in Knowledge Graphs.
Consider a large knowledge graph like YAGO (Hoffart et al., 2013) or Freebase (Bollacker et al., 2008), comprising real-world entities (e. g. *Barack Obama*, *United States*) that are connected via edges corresponding to relationships of some type, e. g. *presidentOf*. Further, individual entities are annotated with semantic types like *Politician*, *Person*, *Country*, etc. A task in the field of relationship analysis is the extraction of (indirect) relationships *of a certain type* between two entities. As an example, consider queries of the form "*Which organizations play an important role in the relationship between the United States and Germany?*" In this setting, an answer to the query can be computed from a collection of ($k \geq 1$) shortest paths between the vertices *United States* and *Germany* that include a vertex of type *Organization*.

We refer to the former problem scenario as *type exclusion* and to the latter as *type inclusion* constraints. Since exclusion constraints simply prohibit the expansion of certain vertices, they can be easily handled by the Path Sketch framework. In the remainder of this section, we discuss the case of type inclusion constraints.
Let (s, t) denote a pair of query vertices and $\theta \in T$ a vertex type that has to be included in the computed solution, respectively. For undirected graphs, we can ensure that a feasible solution can be computed by the Path Sketch index structure by including at least one vertex of the desired type as singleton seed in the precomputed path sketches. However, in order to attain a good approximation quality, we need to ensure the inclusion of vertices of the desired type in the path sketch tree of a vertex $v \in V$, that are indeed located in close distance to v. To this end, we propose the following selection of seed vertices, based on the seed selection strategy for plain distance estimation by Das Sarma et al. (2010).

We denote by $V_\theta := \{v \in V \mid \theta \in \tau(v)\}$ the set of all vertices labeled with type θ, respectively. The following procedure is repeated $r \geq 1$ times, where r denotes the number of selection rounds:

- Randomly select a number of $k = \log(|V_\theta|)$ sets of vertices $S_1, \ldots, S_k \subseteq V_\theta$, with exponentially increasing size $|S_i| = 2^{i-1}$ from the graph.
- For each of the sets S_i and vertex $v \in V$, the union of the shortest paths from and to the *closest seed vertex* from the sets $S_i, 1 \leq i \leq k$ is added to the incoming and outgoing path sketch tree of v, respectively.

Using this seed selection scheme, vertices with label θ that are close in distance to a vertex v are likely to appear in the respective path sketch trees of v. Conceptually, the proposed seed selection strategy leads to a large number of individual seeds (when compared to traditional selection strategies), however, if the number of seed sets in the current iteration is large, a smaller number of vertices will be associated with a certain seed. Since the seed set S_1 contains only a single vertex, it is ensured (for undirected graphs and for directed graphs within a strongly connected component), that for a given pair of query vertices (s, t) at least one approximate shortest path passing through a vertex of type θ (the seed vertex $\sigma \in S_1$) can be returned. It is worth noting that the set based selection strategy can be combined with a selection based on structural properties in the following way. Let $f : V_\theta \to \{1, 2, \dots, |V_\theta|\}$ denote an ordering of the vertices of V_θ based on a certain property (e. g. based on degree in descending order). We select k sets of seed vertices of size $S_i = 2^{i-1}$ according to the following assignment:

$$S_i = \{v \in V_\theta \mid 2^{i-1} \le f(v) < 2^i\} \quad \text{for} \quad 1 \le i \le k. \tag{5.16}$$

In the next section, we address the problem of computing multiple, diverse shortest-path approximations for a given pair of query vertices.

5.5.2 Computing Diverse Paths

In many problem settings, it is a requirement to not only compute a *single* shortest path, but to identify N "best" paths from the source to the target vertex. The ability to quickly generate several shortest-path approximations is one of the advantages of the Path Sketch index structure. More precisely, for the case of k individual vertices selected as seed vertices, an intersection of the path sketch trees $T^+(s), T^-(t)$ will result in k shortest path candidates. In the worst case it could happen that, for a pair of query vertices $(s,t) \in V^2$, all path candidates $p_{s \to \sigma_i \to t}$ are identical. In this case, s and t are connected in the input graph G by the path of length $k+1$ passing through exactly the k seed vertices $\sigma_i, 1 \le i \le k$. In this section, we present (i) a heuristic method for extending the index entries in an existing Path Sketch index in order to increase its ability to compute multiple distinct paths and (ii) a modified query processing algorithm that relies on the same number of vertex expansions as the algorithm presented, but is designed for generating more path candidates

First, we formally define a notion of overlap between paths:

Let $p \neq (s,t) \neq q$ denote two simple (i. e. cycle-free) paths from vertex s to vertex t. We measure the degree of overlap of p, q by

$$\omega(p,q) = \frac{|V(p) \cap V(q)| - 2}{\min\{|V(p)|, |V(q)|\} - 2}. \tag{5.17}$$

The paths p, q are called disjoint in inner vertices if it holds $\omega(p,q) = 0$. If one of the paths is a subpath of the other (including the case $p = q$), it holds $\omega(p,q) = 1$. We call p, q *distinct* if it holds $\omega(p,q) < 1$. In order to increase the number of distinct shortest-path candidates that can be extracted from the path sketch trees in our index structure, we propose the following approach:

- Determine N seed vertices $\sigma_1, \dots, \sigma_N \in V$.
- For each σ_i, compute the "restricted" outgoing and incoming shortest path tree rooted at σ_i, that avoids all paths through any of the previously considered $i-1$ seed vertices $\sigma_1, \dots, \sigma_{i-1}$. The trees are obtained by conducting a breadth-first expansion rooted at σ_i, that does not expand any of the vertices $\sigma_j, 1 \le j < i$.

- For each vertex $v \in V$ add the paths from and to the seed vertices σ_i in the obtained restricted trees to the incoming and outgoing path sketch tree of v, respectively.

Note that the presented approach has heuristic nature and is only guaranteed to enable computation of N distinct paths from the path sketch trees of the query vertices s, t, if the structure of the input graph allows each of the N restricted Bfs expansions to reach both s and t.

The second modification we propose, in order to generate a larger number of shortest-path candidates, concerns the structure of the assigned path sketch trees as well as the query processing algorithm. More specifically, for every vertex $v \in V$ in the graph, we include the set of direct successors $\mathcal{N}^+(v)$ in the outgoing path sketch tree $T^+(v)$ and the set of direct predecessors $\mathcal{N}^-(v)$ in the incoming path sketch tree, $T^-(v)$. Note that the index size will grow by not more than the size of graph itself. The modified index entries are used as follows. Given a query (s, t), we first load the outgoing and incoming path sketch trees of s and t from the index, respectively. Then, in contrast to the previously proposed bidirectional search procedure over $T^+(s), T^-(t)$, we expand individual vertices not by retrieving their direct neighbors from the graph, but by loading their respective path sketch trees from the index. While the number of required random access operations remains unchanged, we can potentially extract many more path candidates since, for any vertex v, the overlap between the path sketch tree $T^+(v)$ and the incoming tree $T^-(t)$ is expected to be much larger than the overlap between the direct neighbors $\mathcal{N}^+(v)$ of v and $T^-(t)$.

5.6 Index Construction

We discuss two algorithms for index construction. The first variant, which is based on back-to-back breadth-first traversals, is discussed next. This algorithm was used in the prototypical index we have proposed earlier (Gubichev et al., 2010). Subsequently, we present StreamForest, a fast construction algorithm in the semi-streaming model which is used as the default index construction algorithm in the new variant of the Path Sketch index.

5.6.1 Traversal-Based Index Construction

The task of the index construction algorithm is the assignment of index entries (path sketch trees) to the vertices in the graph in order to facilitate efficient approximation of the shortest paths between a pair of query vertices. As described in the previous sections, the index entries correspond to two tree-like structures for each vertex in the graph.

The traversal-based index construction works as follows. First, a number of k seed sets are determined. In many seed selection strategies, the seed sets are singletons. However, for the strategy proposed by Das Sarma et al. (2010), it is also possible to use multiple seeds in a set and connect a vertex only to the closest seed

Algorithm 7: BuildPathSketches(G)

```
1. Determine $k$ sets of seed vertices, $S_1, S_2, \ldots, S_k \subseteq V$
2. Initialize the vertex labels as $\ell^-(v) = \emptyset$ for all $v \in V$
3. For each set $S_i$, compute shortest path forest $F_i = \{T_1^{(i)}, \ldots, T_{|S_i|}^{(i)}\}$ rooted
   at the seed vertices, i. e. $r(T_j^{(i)}) = \sigma_j^{(i)} \in S_i$
4. For every tree $T_j^{(i)} = (V_j, E_j)$ and edge $(u,v) \in E_j$, assign the tree path
   $p_{\sigma_j^{(i)} \to v} = p_{\sigma_j^{(i)} \to u} \circ (u,v)$ to vertex $v$:
       $\ell^-(v) \leftarrow \ell^-(v) \cup \{p_{\sigma_j^{(i)} \to v}\}.$
5. Repeat steps (3) and (4) on the reversed graph $G^{-1}$ to obtain the labels
   $\ell^+(v), v \in V$
```

vertex in the set. We employ the set notation in this section since the case of individual seed vertices can be expressed as singleton seed sets. We denote the seed sets by $S_1, S_2, \ldots, S_k$ and the individual seed vertices by $\sigma_1^{(i)}, \sigma_2^{(i)}, \ldots, \sigma_{|S_i|}^{(i)} \in S_i$. For every set S_i we construct a shortest path forest, rooted at the respective seed vertices, by conducting a breadth-first traversal of the graph originating from set S_i. As a result, we obtain for every set S_i a collection of shortest path trees $T_1^{(i)}, \ldots, T_{|S_i|}^{(i)}$, rooted at the respective seeds, i. e. $r(T_j^{(i)}) = \sigma_j^{(i)}$. Since above procedure is repeated for each of the seed sets $S_1, \ldots, S_k$, every vertex of the graph will be contained in up to k trees. Let $t_i(v)$ denote the tree (if any) containing vertex v in the Bfs originating from S_i. We assign to each vertex $v \in V$ a label $\ell^-(v)$ (the path sketch), containing for every tree $t_i(v), 1 \leq i \leq k$, the path from the root vertex $r(t_i(v))$ to v. We repeat the procedure with the same sets of seed vertices on the reversed graph G^{-1}, and eventually obtain the labels $\ell^+(v), v \in V$, each consisting of up to k paths originating from vertex v and ending in a seed vertex. The complete procedure is depicted in Algorithm 7.

Example. Consider the graph depicted in Figure 2.1 on page 12. Assume we have determined the two seed sets $S_1 = \{9\}$ and $S_2 = \{2, 11\}$. The corresponding shortest path forests are depicted in Figure 5.4(a),(b) and (c),(d), respectively.

In terms of computational complexity, the index construction algorithm conducts k breadth-first expansions of the graph, resulting in a time complexity of $O(k(m+n))$. The k breadth-first traversals require many random accesses in order to retrieve the adjacency lists of expanded vertices, incurring high cost. In order to alleviate this problem, we integrate a semi-streaming approach for Bfs into our optimized indexing algorithm, which is presented next.

5.6.2 Optimized Index Construction

We discuss an optimized indexing algorithm based on the following two observations. First, due to the fact that the majority of the indexing effort lies in the construction of Bfs forests, we can exploit optimized algorithms for this widely studied problem. In general, the execution of a Bfs algorithm can be broken down into several phases. In each phase, the unvisited successors of all vertices in the fringe are added to the forest and become the fringe in the subsequent phase. This step can be regarded as a self-join of the edge relation. Thus we can apply optimization techniques known from relational databases, such as switching between random and sequential accesses to the edge list whenever appropriate (i. e. favored by the cost model). In recent work, Beamer et al. (2012) apply this reasoning specifically for the case of Bfs expansions of a graph, and discuss a so-called bottom-up approach in conjunction with a hybrid algorithm, switching between random access and sequential passes over the edges to construct a Bfs tree quickly. The streaming algorithm we discuss in this section is similar in spirit to this approach, which Shun and Blelloch (2013) have developed further with applications to additional graph problems, including the computation of multiple Bfs for graph radii estimation. Our algorithm for index construction as well builds on the fact that in many cases the available main memory permits the representation of more than one forest at the same time. We can exploit this fact and update several memory-resident Bfs forests after each vertex expansion.

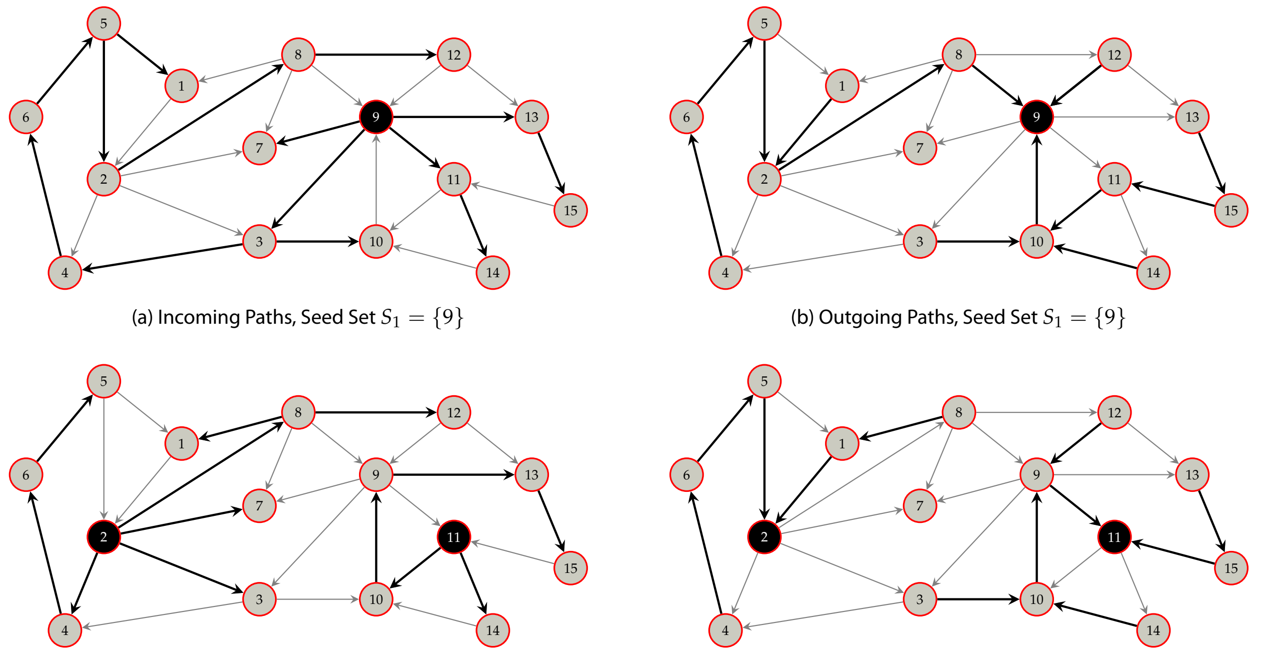
Above ideas are incorporated into the StreamForest algorithm described in this section. The basic algorithm is expressed in the *semi-streaming model* (Feigenbaum et al., 2004). This model assumes that all vertices fit into memory, with a constant amount of data per vertex. The edges of the graph might not entirely fit into memory. To perform computations involving the graph structure, we need to perform multiple passes over the edge list. In the next section, we discuss the details of the algorithm.

Overview

The basic principle of the StreamForest algorithm is to completely avoid costly random accesses to the graph and instead process the edges sequentially, at each point during execution either adding the current edge to one or more forests, or skipping it, depending on whether the source vertex of the edge is a boundary vertex (fringe vertex) in one or more partial shortest path trees[1]

The algorithm works as follows: Let $S_1, S_2, \ldots, S_{k'}$ denote $k' \leq k$ sets of seed vertices. We initialize k' empty forests $F_1, \ldots, F_{k'}$, by setting $F_i = (V, \emptyset)$. The seed vertices contained in set S_i will be the roots of the shortest path trees in forest F_i. For each forest we maintain a set of fringe vertices, Φ_i, initialized to contain the respective seeds, i. e. $\Phi_i = S_i$. Further, we keep for each forest a set Φ'_i of fringe

[1]This principle is what Beamer et al. (2012) refer to as bottom-up approach.



(a) Incoming Paths, Seed Set $S_1 = \{9\}$

(b) Outgoing Paths, Seed Set $S_1 = \{9\}$

(c) Incoming Paths, Seed Set $S_2 = \{2, 11\}$

(d) Outgoing Paths, Seed Set $S_2 = \{2, 11\}$

Figure 5.4: Path Sketch Computation (Example)

vertices for the upcoming iteration, initialized to $\Phi'_i = \emptyset$. A forest is represented by recording the parent vertex for every vertex. We denote the parent of vertex v in forest i by $p_i(v)$.

The algorithm proceeds by scanning the edges of the graph. For the current edge $(s, t) \in E$, we check for each of the forests $F_1, \ldots, F_{k'}$ whether the source vertex s is marked as a fringe vertex. Let $F = \{F_i \mid s \in \Phi_i\}$ denote the respective collection of forests. For each $F_i \in F$ we now have to check whether the target vertex of the edge, t, has already been assigned a parent $p_i(t)$ or not. In the former case, the forest remains unchanged. In the latter case, we add the edge (s, t) to the forest by setting $p_i(t) = s$ and mark t as a fringe vertex for the *upcoming* iteration by adding it to Φ'_i.

As soon as all edges have been processed by the algorithm, we replace the set of fringe vertices, $\Phi_i, 1 \leq i \leq k'$ with the fringe set Φ'_i for the next iteration, and continue with the next round by performing another pass over the edge list. We continue in this fashion as long as at least one of the forests was modified in the preceding round. The complete algorithm is depicted in Algorithm 8.

In the next section, we provide some details on the efficient implementation of the StreamForest algorithm.

Implementation Details

Throughout this section we will assume that the vertices in the graph G are consecutively numbered, starting with id 1. Assume a working memory of M bytes. Further, let size_{id} denote the size (in bytes) of an individual vertex identifier. Typically, we have $\text{size}_{\text{id}} = 4$ (sufficient for graphs with up to $\approx$4.3 billion vertices) or $\text{size}_{\text{id}} = 8$. In order to represent a forest or tree shaped substructure of G, it is sufficient to store for each vertex v the identifier of its assigned parent vertex, $p(v)$. For this purpose, we allocate a contiguous block of $\text{size}_{\text{id}} \cdot n$ bytes. Then, we record the identifier of $p(v)$ at the position $[\text{size}_{\text{id}} \cdot (v-1), \text{size}_{\text{id}} \cdot v - 1]$. The number of forests that can be represented simultaneously in main memory is thus upper-bounded by $k' = \lfloor M/(n \cdot \text{size}_{\text{id}}) \rfloor$.

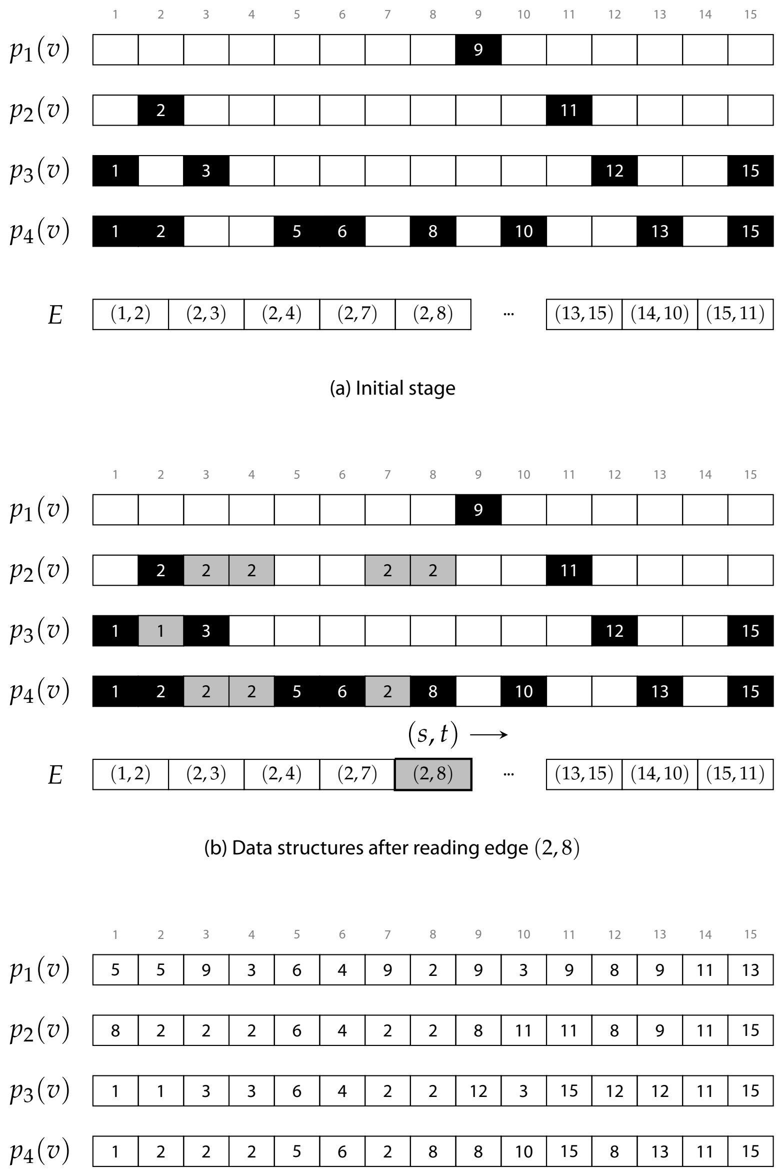
Example. Assume an available working memory of 4 GB and a graph to be indexed containing $n = 100$ million vertices. In this setting, we can represent

$$\left\lfloor \frac{M}{\text{size}_{\text{id}} \cdot n} \right\rfloor = \left\lfloor \frac{1024^3 \cdot 4}{4 \cdot 100 \cdot 10^6} \right\rfloor = 10$$

forests at the same time.

The second component that has to be kept in main memory is a representation of the current and next fringe (expansion boundary) for each of the k' forests. For this purpose, we keep two bit-vectors $f_{\text{curr}}, f_{\text{next}}$, each of size $k'n$, where it holds $f_{\text{curr}}[vk' + i - 1] = 1$ if vertex v is currently a fringe vertex in forest F_i. The space required by each fringe indicator is bounded by $\lceil (k'n)/8 \rceil$ bytes.

This concludes the description of the data structures in use. We use these data structures to represent the forests as well as the sets of fringe vertices throughout the execution of the StreamForest algorithm.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$p_1(v)$									9						
$p_2(v)$		2									11				
$p_3(v)$	1		3									12			15
$p_4(v)$	1	2			5	6		8		10			13		15

E								
$(1,2)$	$(2,3)$	$(2,4)$	$(2,7)$	$(2,8)$	...	$(13,15)$	$(14,10)$	$(15,11)$

(a) Initial stage

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$p_1(v)$									9						
$p_2(v)$		2	2	2			2	2			11				
$p_3(v)$	1	1	3									12			15
$p_4(v)$	1	2	2	2	5	6	2	8		10			13		15

$(s,t) \longrightarrow$

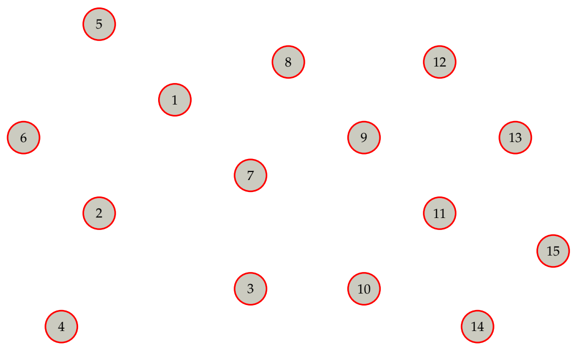
E								
$(1,2)$	$(2,3)$	$(2,4)$	$(2,7)$	$(2,8)$	...	$(13,15)$	$(14,10)$	$(15,11)$

(b) Data structures after reading edge $(2,8)$

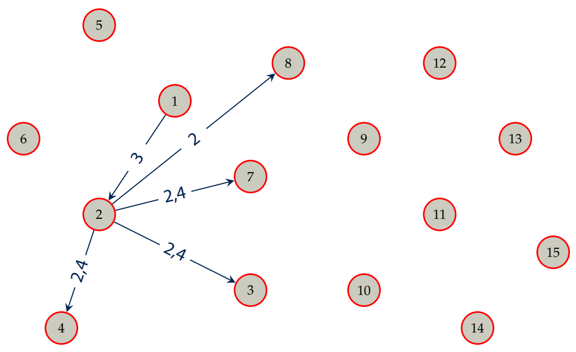
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$p_1(v)$	5	5	9	3	6	4	9	2	9	3	9	8	9	11	13
$p_2(v)$	8	2	2	2	6	4	2	2	8	11	11	8	9	11	15
$p_3(v)$	1	1	3	3	6	4	2	2	12	3	15	12	12	11	15
$p_4(v)$	1	2	2	2	5	6	2	8	8	10	15	8	13	11	15

(c) Data structures after completion

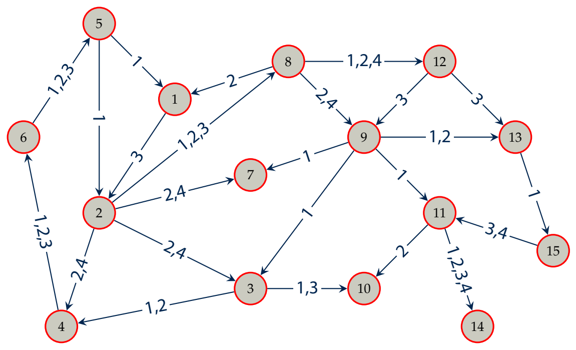
Figure 5.5: Example of the StreamForest algorithm execution – data structures



(a) Initial edge selection



(b) Edge selection after reading edge $(2,8)$



(c) Edge selection after completion

Figure 5.6: Example of the StreamForest algorithm execution – edge selection

Algorithm 8: StreamForest$(G, \{S_1, S_2, \ldots, S_{k'}\})$

```
Data: $G$: input graph, $S_1, \ldots, S_{k'}$: collection of seed sets
Result: collection of $k'$ forests, each rooted at the vertices contained in the
        respective seed set
begin
    $(\Phi_1, \Phi_2, \ldots, \Phi_{k'}) \leftarrow (S_1, S_2, \ldots, S_{k'})$          ▷ current fringe
    $(\Phi'_1, \Phi'_2, \ldots, \Phi'_{k'}) \leftarrow (\emptyset, \emptyset, \ldots, \emptyset)$          ▷ fringe for next round

    foreach $v \in V$ do
        $p_i(v) \leftarrow \texttt{nil} \quad \forall (1 \leq i \leq k')$          ▷ initialize parent pointers
    foreach $i = 1$ to $k'$ do
        foreach $s \in S_i$ do
            $p_i(s) \leftarrow s$          ▷ seeds point to themselves

    while $\Phi_i \neq \emptyset$ for some $1 \leq i \leq k'$ do
        foreach $(s, t) \in E$ do          ▷ stream edges
            for $i = 1$ to $k'$ do
                if $s \in \Phi_i$ and $p_i(t) = \texttt{nil}$ then
                    $p_i(t) \leftarrow s$          ▷ add edge to forest
                    $\Phi'_i \leftarrow \Phi'_i \cup \{t\}$          ▷ add target vertex to fringe

        $\Phi_i \leftarrow \Phi'_i \quad \forall (1 \leq i \leq k')$
        $\Phi \leftarrow \emptyset \quad \forall (1 \leq i \leq k')$
    ▷ return forests, represented by collection of parent pointers
    return $\{p_i \mid 1 \leq i \leq k'\}$
```

EXAMPLE. Consider again the example graph from Figure 2.1. We explain the different steps of the StreamForest algorithm in detail in Figure 5.5 and 5.6. The initial stage is displayed in Fig. 5.5(a), for the data structures and Fig. 5.6 for the selected edges. We depict the arrays containing the parent pointers. The vertices comprising the current fringe are highlighted in black in the respective forest. Figure 5.5(b) and 5.6(b) show the updated data-structures and selected edges after scanning through the edge list up to edge $(2, 8)$. The edge has been added to forest F_2 because 2 is a current fringe vertex (show in black color) in this forest and vertex 8 was not yet assigned a parent. After all edges have been examined, the vertices shown with gray background become the new fringe and the procedure is repeated. The final stage after termination of the algorithm is depicted in Figure 5.5(c) and 5.6(c).

Due to memory restrictions, in general, a single execution of the StreamForest algorithm does not suffice to cover all seed sets $S_1, S_2, \ldots, S_k$ and conclude the precomputation in one round. In the next section we give an overview of the complete index construction algorithm in these cases, incorporating the forest construction as a major building block.

Algorithm 9: ExtractTree$(v, \{p_1, p_2, \ldots, p_{k'}\})$

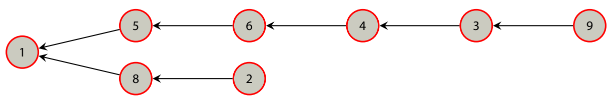
```
Data: $v$: vertex, $p_1, \ldots, p_{k'}$: mappings from vertex to id of parent vertex
Result: $E_T$: collection of edges (of a tree rooted at $v$)
begin
    $E_T \leftarrow \emptyset$
    foreach $i = 1$ to $k'$ do
        $x \leftarrow v$
        while $p_i(x) \neq x$ do
            $E_T \leftarrow E_T \cup \{(p_i(x), x)\}$
            $x \leftarrow p_i(x)$
return $E_T$
```

Complete Procedure

As in the previous section, let k' denote the number of forests that can be kept in memory in at the same time. Further, let k denote the number of seed sets, corresponding to the total number of forests that we have to compute in order to build the Path Sketch index. The index construction algorithm proceeds by repeatedly computing chunks of k' forests in memory until all k seed sets have been processed. After each execution of the StreamForest algorithm, we extract for *each vertex in the graph* a tree structure from the parent-pointer arrays. More specifically, for each of the forests $F_i \in \{F_1, F_2, \ldots, F_{k'}\}$, we extract for vertex v the path $p_{r(t_i(v)) \to v}$ from the root of the tree $t_i(v) \in F_i$, that contains v, up to vertex v itself. The k' extracted paths, starting at the different root vertices $r(t_i(v)), 1 \leq i \leq k'$ and ending at vertex v are then merged into a new (reversed) tree structure. This tree structure is then serialized into an intermediate file.

The procedure to extract this tree structure from the parent pointers is displayed in Algorithm 9.

Example. Consider the complete forest data structure returned by the first execution of StreamForest on the example graph, depicted in Figure 5.5(c). We extract for each vertex its corresponding tree structure using Algorithm ExtractTree, starting with vertex 1. For vertex 1, by following the parent pointers of the 4 arrays $p_1, \ldots, p_4$ up to the root vertices and adding the respective edges, we obtain the structure depicted below:



Whenever it holds $k' < k$, it is necessary to conduct multiple executions of the StreamForest algorithm in order to compute the required number of k forests. Each of the $I = \lceil k/k' \rceil$ iterations will produce one output file, $f_i, 1 \leq i \leq I$, containing the serialized tree structure for each vertex, in ascending order of vertex id. To

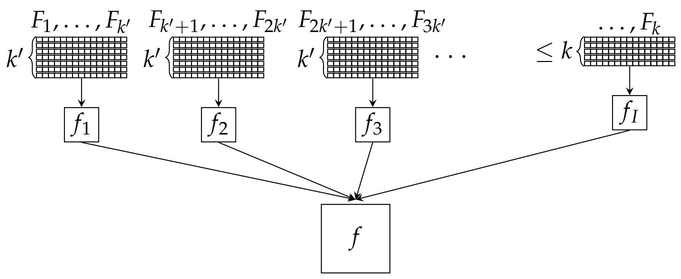


Figure 5.7: Merging Index Files

extract the full index entries for each vertex, we thus have to merge the (sorted) files into one large result file. A schematic overview of this procedure is given in Figure 5.7.

The algorithm for the computation of the outgoing path sketches works accordingly (in fact, the same algorithm is used with the only difference that the currently read edge (u, v) is reversed).

Computational Complexity

For each round of forest computation, we need to perform at most $\text{diam}(G)$ passes over the edge list. Thus, the total number of passes is bounded by

$$O(\text{diam}(G) \cdot I) = O\left(\frac{k \cdot \text{diam}(G)}{\left\lfloor \frac{M}{\text{size}_{\text{id}} \cdot n} \right\rfloor}\right).$$

The per-item processing time for each edge (s, t) is bounded by $O(k')$, since we have to determine whether the current source s is contained in the fringe for each of the k' forests, followed by recording s as the parent of vertex t in at most k' forests. The time complexity of extracting the partial path sketch of a vertex from the k' forests given by the arrays of parent pointers is given by $O(k \cdot \text{diam}(G))$.

The algorithm for the computation of the outgoing path sketches works accordingly (in fact, the same algorithm is used with the only difference that the currently read edge (u, v) is reversed).

5.6.3 Hybrid Algorithm

The semi-streaming approach presented in this section is particularly well-suited for small-world graphs, such as social networks. However, the number of streaming rounds, and thus the total running time of the construction algorithm, directly corresponds to the maximum eccentricity (i. e. distance to other vertices) of the seed sets under consideration. Even in graphs with a small effective diameter (i. e. short paths between a large fraction of all possible vertex pairs), it can happen in the worst-case that many streaming rounds are required to compute the few long paths originating in the selected seed set, resulting in many superfluous reads of

the edge list for already computed paths. In order to alleviate this problem, our implementation employs a *hybrid* algorithm that carefully chooses to either conduct a streaming pass over the edge list or use random accesses in order to retrieve the adjacency lists of the current set of fringe vertices, $\Phi = \Phi_1 \cup \ldots \cup \Phi_k$. Regarding this problem, Beamer et al. (2012) as well as Shun and Blelloch (2013) propose a thresholding heuristic based on the number of vertices and the number of their outgoing edges compared to the total number of edges.
Our own hybrid construction algorithm is also based on a threshold parameter on the *fringe fill-rate*, however, in our case, the threshold is determined by benchmarking the hard disk prior to index construction. For this purpose, in the initialization phase, we measure the average time required to retrieve an adjacency list from disk, t_{RA}, and the time required to perform one sequential pass over the edge list, t_S. Then, we conduct a sequential scan if for the current round it holds

$$|\Phi| \leq \frac{t_S}{t_{\mathrm{RA}}}. \tag{5.18}$$

This way, we use the semi-streaming approach while there are sufficiently many vertices in the expansion fringe (the major part of the work, typically in the middle of the indexing process), and rely on the random-access strategy if only few vertices have to be expanded in the next round (typically in the beginning and towards the end of forest construction), depending on the benchmarked graph access operations. In the experimental evaluation of this chapter, we show the performance difference of the traditional, streaming-only, and hybrid algorithm for index construction in detail. The benchmark-based cost model we propose for determining the right strategy to use is easy to implement and works well in practice, as highlighted in our experimental evaluation.

5.7 Physical Index Layout

Materializing two trees for each vertex in the graph inevitably implies a space requirement in the order of several multiples of the size of the underlying graph. However, by using a sophisticated tree-structure encoding technique in conjunction with byte-level compression of the resulting index entries, the disk space required by the Path Sketch index can be kept small enough to allow index construction over massive graphs comprising billions of edges, requiring only standard disk space of a few hundred GB.

In this section, we provide an in-depth description of the physical layout of the Path Sketch index together with the algorithms involved in – and designed for – the serialization of individual index entries on disk. We first discuss several approaches for serializing tree structures, followed by the detailed description of the low-level representation of our index structure in terms of individual bytes.

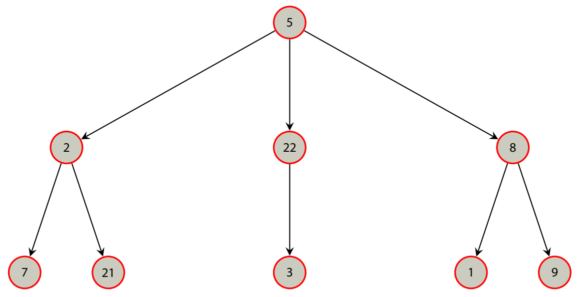


Figure 5.8: Example tree $T = (V_T, E_T)$

5.7.1 Tree Serialization

Since the directed rooted trees, that we assign as index entries, are special instances of general directed graphs, all standard serialization approaches for these graph structures are applicable, most notably edge and adjacency list representations.

Edge List

Consider the directed tree $T = (V_T, E_T)$ depicted in Figure 5.8, which we will use as running example throughout the remainder of this section. The easiest way to serialize a directed graph structure such as T is to simply record all edges in the graph in one contiguous sequence. In this setting, each edge is represented by the identifier of the source followed by the identifier of the target vertex. In this description, as well as in our implementation, we represent vertices by integer ids. Note that there exist $m!$ possible serializations of T with $|E_T| = m$. One possible representation is given by the sequence depicted in Figure 5.9(a).

Note that this sequence contains the edges of the tree together with a header entry indicating the number of edges in the tree. In practical applications, vertex identifiers will be either 32- or 64-bit integers (for $|V| > 2^{32}$). Thus, the space requirement of the edge list representation is given by $(2m + 1)\text{size}_{\text{id}}$. In the example above, the space requirement (using 32-bit ids) of the edge list amounts to 65 bytes. We can further reduce the amount of memory by using a variable byte-length encoding. Note that due to the requirement of extremely fast decompression at query time, we refrain from more aggressive and thus CPU-intensive compression techniques (e. g. bit-level compression).

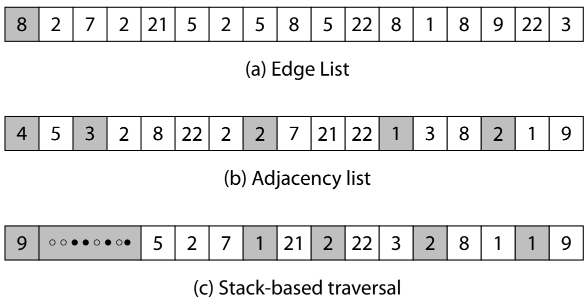


Figure 5.9: Comparison of Serialization Approaches

Adjacency Lists

A serialization technique closely related to the edge list representation is to encode the tree structure in terms of the adjacency lists of the individual vertices. In contrast to edge list encoding, the space requirement of this approach depends on the structure of the tree rather than on the absolute number of edges. In this setting, we simply group edges based on common source identifiers. Then, the edges are treated by recording the source identifier together with the sequence of the respective target identifiers. In order to extract the individual adjacency lists at query time, it is necessary to record the number of entries in each list. As a side note, storing the target ids of each adjacency list in ascending order opens up the possibility of applying further optimizations, most importantly delta-encoding of the gaps between subsequent entries instead of listing the identifiers themselves. This technique, traditionally used for posting list compression of inverted indices, has proven beneficial when used in conjunction with variable byte-length compression schemes. The rationale is that the magnitude of gap sizes and thus their space requirement is substantially smaller than the values of vertex identifiers.
We depict the adjacency list representation of the example tree T (without delta-encoding) in Figure 5.9(b).

Note that, due to the restricted number of k seed sets $S_1, S_2, \ldots, S_k$ (following the seed selection scheme of Das Sarma et al. (2010) we have $k \in O(\log(n))$, the maximum degree of each vertex contained in the index entries (trees) is restricted by $\delta(v) \leq k$. In most practical scenarios, we can thus assume that a single byte is sufficient to record the number of entries per adjacency list. As in the previous example, we highlight the meta-information contained in the serialized tree with gray color. This involves, apart from the list lengths, the number of adjacency lists contained in the index entry, stored as the header. In our example, the space requirement (using one byte to signify list lengths as well as the number of lists in the header) of this representation amounts to 53 bytes.

Stack-Based Traversal

The third serialization technique we consider is more involved than the straightforward approaches discussed above, yet allows for extremely fast (de-)serialization algorithms. This approach results in the best overall compression ratio, as indicated by our experimental evaluation. The key idea of the (de-)serialization procedure, based on a text book algorithm (Shaffer, 2011), is to traverse the tree structure in a depth-first manner and emit identifiers of encountered vertices together with certain *instructions* to the deserialization algorithm, whenever appropriate. We make a distinction between inner vertices of the tree and leaves. Then, during traversal of the tree, we maintain a stack that holds the identifiers of the vertices on the path from the root to the current vertex. Whereas intermediate vertices are simply pushed onto the stack (and written to the output) upon discovery, leaf vertices require a certain number of elements be popped from the stack in order to proceed with the traversal. This particular number of pop-operations before the next vertex in the Dfs sequence can be pushed, is recorded in the serialized tree output, immediately after the id of the respective leaf vertex. Consider again the tree T from our running example. The depth-first traversal proceeds discovering (and emitting) the ids of vertices 5,2 and 7. The leaf vertex 7 requires one pop-operation before the next id, 21, can be pushed onto the stack. Likewise, 2 pop-operations are necessary after the id of vertex 21 has been emitted.

For the decompression algorithm in order to reconstruct the tree structure from the serialized representation, we need to mark for each vertex whether it is a leaf (then the subsequent number denotes the required pop-operations) or an inner vertex, in which case the following entry is the next vertex identifier in the Dfs traversal. For this purpose, we add a bitmap to the header, indicating for each vertex whether the vertex corresponds to a leaf of the tree. For performance reasons, we strive to keep the bytes in the index entry aligned, and thus, the space requirement for the bitmap amounts to $\lceil \frac{n-1}{8} \rceil$ bytes, where $n = |V_T|$ (note that we do not record the leaf indicator of the last vertex since this vertex is anyways a leaf). The number of pop-operations is bounded by the height of the tree, which in turn cannot exceed the diameter of the graph. Since in this work we only consider small-world graphs, a single byte is sufficient to record the number after each leaf. Finally, the serialized representation of the example tree, T is given by the sequence depicted in Figure 5.9(c). Again, meta-information (header with number of vertices and leaf bitmap as well as the number of pops after each leaf) is highlighted. The required space amounts to 42 bytes.

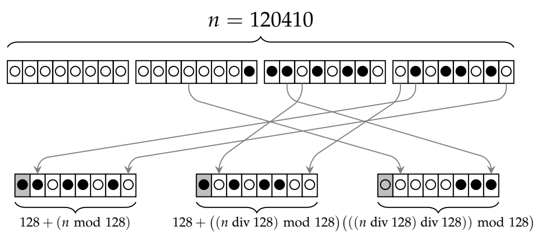

$n = 120410$
$128 + (n \bmod 128)$
$128 + ((n \operatorname{div} 128) \bmod 128)$
$(((n \operatorname{div} 128) \operatorname{div} 128)) \bmod 128)$

Figure 5.10: Example: Variable Byte-Length Encoding

5.7.2 Byte-Level Compression

As mentioned in the preceding paragraphs, in addition to serializing the tree structure in a space-preserving manner, we also experiment with generic compression mechanisms. More precisely, we investigate byte-level compression of the integers emitted by the serialization algorithm. To this end, we employ the basic form of variable byte-length encoding where the integers are recorded as a sequence of bytes where every byte consists of one header bit (continuation bit) and seven payload bits. This approach is particularly attractive for adjacency list serialization in conjunction with delta-encoding.

We briefly explain this integer encoding technique in the following example: Consider the bit-representation of the 32-bit integer 120410, depicted in Figure 5.10 (little endian notation):

While the number is stored in a 4-byte integer, only the lower three bytes actually contain set bits. The variable byte-length encoding technique exploits this fact by recording only the 7-bit sequences from the LSB up to the highest set bit (inclusive). The continuation bit in the header indicates whether the following byte is part of the number representation as well. Note that in the resulting sequence, the payload is arranged in the order of increasing significance. This encoding technique can be efficiently implemented by means of bitwise operations. For further details we direct the reader to the classic literature surveying the field (Manning et al., 2008).

5.7.3 Indexed Flat-File Implementation

Recall that in the Path Sketch index structure, we have to store two records for every vertex in the graph: the serialized incoming and outgoing trees $T^-(v)$ and $T^+(v)$, respectively. In general, the index entries will have variable lengths.

Thus, we choose to store the records in a collection of flat files, where individual index entries are indexed by recording their offset in the file. Since index entries typically consist of tens to hundreds of individual identifiers for the vertices contained in the trees, the space required for storing offsets induces a tolerable size overhead. We store the offsets (which are 4-byte values indicating the byte offset in the file) in dedicated offset files, one for outgoing and one for incoming path sketch trees, respectively. For the case of outgoing trees $T^+(v), v \in V$, the index entries are stored in a number of F_O size-restricted files. The number of files depends on the disk space required by the serialized trees, that is, whenever the size of an index file extends past 4 GB, the 4 bytes allotted to store the entry offset are insufficient, and thus an additional index file is created. The incoming trees are stored in the same way in F_I index files.

Whenever the (physical or user-specified) constraints on the available computational resources permit, it is possible to load the offsets into main memory. This way, access to an index entry at query time only requires a single random seek rather than two (to retrieve offset and index entry, respectively). Finally, in order to determine for a given vertex the respective index file, we need to record for each index file the id of the first vertex in the file. Since the number $F_O + F_I$ of index files is typically very small, we can easily hold the $F_O + F_I$ start ids per file in main memory and then identify the appropriate file for a given vertex id.

5.8 Related Work

5.8.1 Distance Querying

Distance indices can be regarded as an intermediary between reachability and shortest path index structures, over both undirected and directed graph structures. An active area of research, numerous indexing approaches have been proposed in the past, both for exact as well as approximate distance query processing.

Exact Query Processing. In this setting, the graph is preprocessed and index structures are built such that queries for the distance (i. e. length of the shortest path) of a pair of vertices $(s, t) \in V^2$ can be answered significantly faster than the naïve approach of online search (Bfs). A common characteristic of most indexing approaches is their distributed nature, in the sense that the overall index structure consists of *node labels* assigned to the individual vertices. The distance between a pair of vertices can then be computed by inspecting the labels of the query vertices. Classical work in this direction includes the notion of proximity-preserving labeling schemes (Peleg, 2000) and the study of Gavoille et al. (2004) on minimum label-length for different kinds of graph structures. Cohen et al. (2002) propose the notion of *2-hop covers* for reachability and distance labeling. A 2-hop distance labeling is defined as the assignment of a label $L(v)$ to each vertex $v \in V$ of the graph $G = (V, E)$. The label consists of two sets, $L(v) = \big(L_{\text{in}}(v), L_{\text{out}}(v)\big)$, that are collections of vertex-distance pairs $(u, d(u, v))$ and $(w, d(v, w))$, respectively. The distance from vertex s to vertex t can be computed as

$$d(s,t) = \min_{v \in L_{\text{out}}(s) \cap L_{\text{in}}(t)} d(s,v) + d(v,t).$$

It has been shown that the problem of identifying 2-hop covers of minimum size is NP-hard, however, an almost optimal labeling can be achieved with the approximation algorithm proposed by (Cohen et al., 2002). While this algorithm is efficient, large problem instances render this approach infeasible, as demonstrated by (Schenkel et al., 2006). Numerous approaches addressing the efficiency of 2-hop

cover construction have been proposed subsequently. Schenkel et al. (2004) propose a divide-and-conquer approach with restricted memory usage in order to process large XML data collections. Several enhancements to this algorithm are proposed by (Schenkel et al., 2006), including a recursive approach for joining covers computed for different partitions. Cheng and Yu (2009) propose an algorithm for faster computation of distance-aware 2-hop covers by first computing the condensed graph, obtained after collapsing all strongly connected components into supervertices, followed by the computation of a 2-hop cover for this typically much smaller graph in conjunction with a partitioning scheme.
An important characterization of graphs, the notion of *highway dimension*, was recently introduced by Abraham et al. (2010). A graph exhibits low highway dimension if for every value d there exists a sparse set of vertices S_r, such that every shortest path exceeding length d contains a vertex from the set S_r. It is shown that important classes of graphs, most notably road networks, exhibit a low (polylogarithmic) highway dimension, which in turn allows distance labels of polylogarithmic size. In a follow-up work, Abraham et al. (2012) propose *hierarchical hub-labeling*, where the vertex inclusion in a distance label exhibits a partial ordering of the vertices. Jin et al. (2012b) combine the highway structure with a bipartite set cover approach for distance querying over large, sparse graphs. It is shown that the resulting labeling is superior to 2-hop labeling, both empirically and theoretically, regarding indexing time, index size and query processing time. Recently, Akiba et al. (2013) proposed *pruned landmark labeling* based on the precomputation of distance labels obtained by executing a Breadth-First expansion from every vertex of the graph. This at first sight computationally infeasible approach is facilitated by a clever pruning scheme, that terminates expansions early based on the information already contained in the partially constructed index. This approach enables substantial performance benefits especially during the later stages of the indexing process. Fu et al. (2013) propose IS-Label, a vertex labeling approach based on the organization of the input graph hierarchically into layers, based on the notion of *independent sets*, sets of pairwise non-adjacent vertices with respect to a specified (sub-)graph. The resulting labels are stored on disk and combined with bidirectional Dijkstra search for query processing.
Due to the wide applicability of the problem, further problem variants have been studied, including single-source distance computation (Cheng et al., 2012).

Approximate Query Processing. Another important stream of research in distance query processing can be classified as distance estimation techniques, where only an estimate on the true distance for a pair of query vertices is returned. Dropping the requirement to compute exact solutions leads to substantial benefits in space and time consumption. In this setting, distances between the vertices of the graph and a predefined set of seed vertices (often called *landmarks* are precomputed. Then, for a pair of query vertices (s, t), the distance from vertex s to vertex t is computed as the minimum sum of distances from s to a seed vertex and from the same seed vertex to t. Given that the edge weights satisfy the triangle inequality (as is the case for unweighted graphs), this method provides an upper bound on the true distance. Thorup and Zwick (2005) show in their classical work that for undirected, weighted graphs, it is possible to construct an index structure for distance estimation of size $O(kn^{1+1/k})$ in $O(kmn^{1/k})$ expected time such that distance queries can be answered in $O(k)$ time. The returned distance estimate exhibits a multiplicative stretch of at most $2k - 1$. Thus, for the estimated distance $\tilde{d}(s,t)$ it holds

$$1 \leq \tilde{d}(s,t)/d(s,t) \leq 2k - 1.$$

Potamias et al. (2009) study the effects of different seed selection strategies on the estimation accuracy. In order to improve accuracy especially for close-by query vertices (a case that often occurs in social networks and other small-world graphs), Das Sarma et al. (2010) advocate the use of randomly sampled seed sets of exponentially increasing size. For undirected graphs, Qiao et al. (2012) use a least-common-ancestor approach to identify a seed close to the query vertices. Qi et al. (2013) study seed-based indexing for distance query processing in a distributed setting.

5.8.2 Shortest Path Querying

The classical textbook algorithm for pairwise shortest path query processing are (bidirectional) Bfs for unweighted, and Dijkstra's algorithm for weighted graphs. Significant improvements in limiting the search space can be achieved by the A^* algorithm (Hart et al., 1968), which maintains the vertices in the expansion queue by their *potential* to provide a connection to the target. This is achieved by combining the distance from the source with a lower bound on the distance (so-called admissible heuristic) to the target. For graphs like road networks, where the coordinates of vertices are known, the straight-line distance is a common heuristic. For general graphs, Goldberg and Harrelson (2004) propose ALT, a combination of A^* search using landmarks as a heuristic to compute lower bounds on the distance to the target in conjunction with the triangle inequality.

Recent results on *index support* for shortest path query processing include previously discussed frameworks for distance querying, which can be extended to support path queries (Akiba et al., 2012, Fu et al., 2013). These modifications involve additional bookkeeping (e. g. maintaining the first edge on the shortest path to the target), with detrimental effect on both precomputation time as well as index size. Furthermore, maintaining only the first edge on the path to the target requires additional effort, proportional to the length of the resulting path, by requiring additional I/O operations that in turn negatively impact the query processing performance. Other approaches, like the prototypical implementation of Path Sketch (Gubichev et al., 2010) are designed specifically for path queries rather than offering this functionality in an extension, leading to different design decisions for the involved algorithms and data structures.

Road Networks. Shortest paths play a crucial role in many applications involving (near-)-planar graphs such as route planning over road networks. This property enables provable efficiency of methods like contraction and highway hierarchies (Geisberger et al., 2008, Sanders and Schultes, 2005) and transit node routing (Bast et al., 2007). Recently, Abraham et al. (2011) have applied results from learning theory, in particular related to the Vapnik-Chervonenkis dimension, in order to derive better bounds on query time complexity for the case of graphs with low highway dimension. Rice and Tsotras (2010) propose an index structure based on contraction hierarchies to support path query processing in the presence of restrictions on the labels of edges contained in the solution.

Complex Networks. In addition to the earlier prototype of our Path Sketch index structure (Gubichev et al., 2010), we mention the following index structures for shortest path query processing: Xiao et al. (2009) propose the materialization of Bfs-trees and exploit symmetric substructures within to reduce index size. Wei (2010) applies a tree decomposition approach with a tunable parameter controlling the the time/space-tradeoff. Along similar lines, Akiba et al. (2012) propose an index for both graphs with small tree-width and complex networks. The approach is also combined with the seed-based technique for a hybrid algorithm. Tretyakov et al. (2011) show how the indexing and query processing strategy of a path-based index structure such as Path Sketch can be adapted by restricting the index entries to contain only the first edge of each path to the seed. This way, the index structure can easily incorporate updates to the underlying graph structure.

Shortest Path Variants. Gao et al. (2011) discuss the implementation of shortest path algorithms inside a relational database system. Tao et al. (2011) study a variant of shortest path computation where a solution specifies only at least one of every k consecutive vertices in the resulting path. This scenario has applications in the context of spatial network databases.

For a more comprehensive overview, the reader is directed to the survey of Sommer (2012).

5.9 Experimental Evaluation

5.9.1 Datasets and Setup

We evaluate our algorithms over a variety of real-world datasets, focusing on social network and web graphs. The datasets we consider are:

- **Wikipedia** Link graph of the English Wikipedia (article namespace) from February 2013[2].
- **Google** Web graph from the 2002 Google Programming Contest[3] (Leskovec et al., 2009).

[2]http://law.di.unimi.it/datasets.php

Dataset	Type	Disk Size	$\|V\|$	$\|E\|$	Avg. Degree	Diameter
Wikipedia	ER graph	400 MB	4,206,289	101,311,614	24.09	53
Google	web graph	21 MB	875,713	5,105,039	5.83	47
BerkStan	web graph	22 MB	685,230	7,600,595	11.09	676
UK	web graph	8.4 GB	105,896,435	3,717,169,969	35.10	1,096
Slashdot	social network	4 MB	82,168	870,161	10.59	12
DBLP	social network	27 MB	986,207	6,707,236	6.80	20
Twitter	social network	6 GB	41,652,230	1,468,364,884	35.25	18

Table 5.1: Dataset Characteristics

- **BerkStan** Web graph from pages of the `berkeley.edu` and `stanford.edu` domains crawled in 2002[3] (Leskovec et al., 2009).
- **UK** Web graph of the `.uk` domain from May 2007[2] (Boldi et al., 2008).
- **Slashdot** Social network among users of the technology website Slashdot[3] (Leskovec et al., 2009).
- **DBLP** Social network of scientists cooperating on articles, extracted in July 2011 from the DBLP bibliography server[2].
- **Twitter** Directed social network of users of the microblogging social network[2] (Kwak et al., 2010).

We summarize basic statistics of the graphs in Table 5.1. While some of the graphs above are undirected, in this work we treat all inputs as directed graphs. That is, we convert undirected graphs to a directed representation by including both possible directions for each edge in the graph. The numbers in Table 5.1 reflect the sizes of the directed graphs after this conversion.[4]

All experiments were conducted on a server equipped with 2 Intel Xeon X5650 6-core CPUs at 2.66 Ghz, 64 GB of main memory and four local disks configured as RAID-10, i. e. using mirroring without parity and block-level striping. The operating system in use was Debian 7.0 using kernel 3.10.45.1.amd64-smp. Our algorithms are implemented in C++ and compiled using GCC 4.7 at the highest available optimization level.

[3] `https://snap.stanford.edu/data/`

[4] In principle we need only one direction of the sketches for query processing over undirected graphs. Thus, the index construction times and index sizes for undirected (symmetric) graphs would amount to roughly half of the values reported in this section if a distinction is made between directed and undirected inputs.

5.9.2 Baseline Strategies

As baseline strategies for processing shortest path and distance queries, we consider BFS (breadth-first-search) as well as bidirectional BFS over the memory-mapped, disk-resident graph and also compare with two state-of-the art approaches for point-to-point (exact) distance computation:

- IS-Label (Fu et al., 2013), is a recently proposed indexing scheme for exact distance querying of large graphs based on independent sets – that is, sets of pairwise non-adjacent vertices – for efficient index construction.

- Pruned Landmark Labeling (PLL) (Akiba et al., 2013), is an index structure for exact distance queries based on the precomputation of vertex distances by conducting breadth-first-expansions from all vertices. The key idea in this approach is the use of a clever pruning technique, that drastically reduces both index size as well as precomputation time when compared with a naïve approach.

For both PLL and IS-Label, we obtained the original source code from the authors. While the implementation of IS-Label allows for a disk-based index construction out of the box, we modified the source code for PLL to construct disk-based index entries, using offset-based index entry retrieval, similar to the Path Sketch index. Furthermore, we adapted PLL to work on directed graphs, since we treat all input datasets as directed graphs for all considered algorithms. All approaches are implemented in C++ and compiled at highest available optimization level. The approaches considered in this work are single-thread implementations.

5.9.3 Disk Access

As the storage layer, we materialize the incoming and outgoing edges of the vertices on disk in two separate, clustered B$^+$-trees, adapted from the RDF-3X database engine (Neumann and Weikum, 2010). As in the original RDF-3X implementation, these B$^+$-trees are backed by memory-mapped files using demand-paging. Thus, during program execution memory equal to the size of the database is reserved, and pages are read into memory from disk as they are needed, and stay memory-resident until program termination. If the graph size is larger than the available main memory, virtual memory is allocated. Both (bidirectional and regular) BFS, as well as the Path Sketch query processor access the graph structure in this manner, whereas the actual Path Sketch index entries are stored in binary files that are accessed via seek-operations. IS-Label and PLL use their own methods to access the graph, which they need to access only during indexing. PLL loads the graph to index entirely into main memory, and constructs the full index in main memory as well. For PLL, we use 64 bit-parallel labels for all datasets and order vertices by degree (sum of in- and out-degrees). We remark, that other strategies are possible for PLL, e. g. ordering vertices by product of in- and out-degrees as Path Sketch does for seed selection. IS-Label accesses the graph on disk, satisfying a user-specified budget on the amount of memory to use, which we keep at the default setting (4

GB). Both Path Sketch as well as our adapted version of PLL use offset-based retrieval of index entries. The offsets of index entries in the files are kept in main memory during query processing for both approaches.

5.9.4 Index Construction

The first evaluation metric we consider is **indexing time**, the total time required to create the disk-resident index structures from a given (as well disk-resident) graph, i. e. including serialization and compression.

Indexing Strategies

We consider the following three seed selection strategies:

Closeness Centrality (CC) Selection of the top-k vertices with highest closeness centrality, given by

$$\mathrm{cc}(v) = \sum_{w \in V \setminus \{v\}} 2^{d(v,w)}. \tag{5.19}$$

Since exact computation the centrality scores entails an all-pairs shortest distance computation, we approximate the centrality scores by randomly sampling $1,000$ vertices Σ:

$$\tilde{\mathrm{cc}}(v) = \sum_{s \in \Sigma} 2^{d(s,v)}. \tag{5.20}$$

We then use the $k \in \{10, 25, 50, 100\}$ vertices with highest centrality score as singleton seed sets for constructing the Path Sketch index:

$$S_i := \left\{ \operatorname*{arg\,max}_{v \in V \setminus \bigcup_{j=1}^{i-1} S_j} \tilde{\mathrm{cc}}(v) \right\}, 1 \leq i \leq k \tag{5.21}$$

Degree Product (DP) Selection of the top-k vertices with highest product of in- and out-degrees:

$$\mathrm{dp}(v) = |\mathcal{N}^{+}(v)||\mathcal{N}^{-}(v)|. \tag{5.22}$$

We use the $k \in \{10, 25, 50, 100\}$ vertices with highest product of in- and outdegrees as singleton seed sets:

$$S_i := \left\{ \operatorname*{arg\,max}_{v \in V \setminus \bigcup_{j=1}^{i-1} S_j} \mathrm{dp}(v) \right\}, 1 \leq i \leq k. \tag{5.23}$$

Random Exponential (RE) Selection of $K \cdot \log(n)$ sets of randomly selected vertices $S_i, 1 \leq i \leq K \log(n)$ of exponentially increasing size, $|S_i| = 2^{i-1}$. We vary the parameter $K \in \{1, 2, 5, 10\}$.

Dataset	*Closeness (CC)*				*Degree Product (DP)*				*Random Exponential (RE)*			
	$k = 10$	$k = 25$	$k = 50$	$k = 100$	$k = 10$	$k = 25$	$k = 50$	$k = 100$	$K = 1$	$K = 2$	$K = 5$	$K = 10$
Wikipedia	329	592	1,055	2,046	334	591	1,081	2,062	588	1,048	2,472	4,850
Google	51	115	218	409	57	116	223	441	68	137	337	654
BerkStan	91	187	347	699	94	195	370	737	196	418	1,023	2,002
UK	32,066	58,697	–	–	28,792	63,985	106,844	–	138,973	–	–	–
Slashdot	3	7	11	21	4	8	12	22	5	9	20	38
DBLP	41	81	151	299	41	84	148	311	68	112	290	570
Twitter	4,324	7,189	13,009	22,359	5,307	9,180	13,048	24,336	9,070	17,144	35,489	67,789

Table 5.2: Indexing Time [s]: Comparison of Seed Strategies, PathSketch (Hybrid Algorithm)

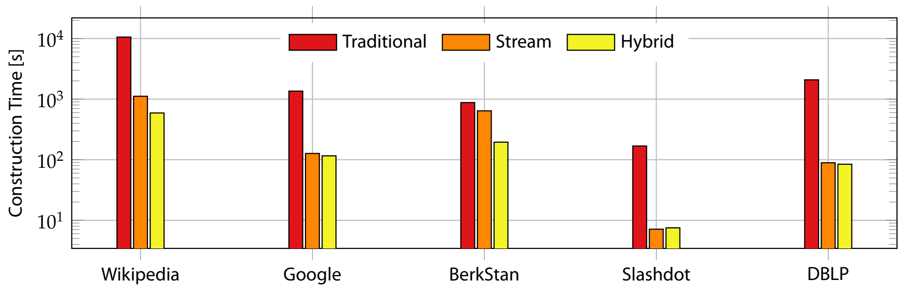


Figure 5.11: Indexing Time [s], Comparison of Construction Algorithms, Path Sketch (DP, $k = 25$)

The index construction times (just construction, excluding determining the seed vertices[5]) for the different settings of seed selection strategies and cardinalities are presented in Table 5.2 using the default settings (adjacency list representation of path sketch trees with delta encoding, use of variable byte-length encoding, and inclusion of the adjacency list of the root vertex in the tree). In this setting, we use the hybrid algorithm and construct exactly 10 shortest path forests in main memory per iteration. It is evident that the integration of the streaming BFS technique in conjunction with the construction of multiple trees into Path Sketch allows for a very efficient index construction. As a result, Path Sketch can easily handle even massive graphs comprising billions of edges. As an example, for the strategy $DP_{k=10}$, i. e. using the top-10 vertices with highest degree product, we are able to construct the index structure in less than 90 minutes on the Twitter social network comprising roughly 1.5 billion edges and in around 8 hours on the UK web graph comprising 3.7 billion edges. Small datasets, like the Slashdot social network with a little less than 1 million edges, can be processed within just a few seconds.

For the choice of closeness centrality to determine seeds as well as random exponential seed sets, the index can also be constructed very efficiently. It is worth noting that, for the random exponential seed selection, the number of seed sets depends on the size of the graph (number of vertices), whereas for the CC and DP strategies the number of seed sets is fixed a-priori using parameter k. In this experiment, we use a timeout of 48 hours after which the index construction is aborted (this was the case only for the UK webgraph for settings $CC_{k=100}$, $DP_{k=100}$, and $RE_{K>1}$.

[5]Which is cheap for degree product and expensive (since requiring computation of several shortest path trees) for approximated closeness centrality

Dataset	IS-Label	PLL	PS (CC-25)	PS (DP-25)	PS (RE-2)
Wikipedia	8,218	–	592	591	1,048
Google	56	181	115	116	137
BerkStan	164	98	187	195	418
UK	–	–	58,697	63,985	
Slashdot	10	5	7	8	9
DBLP	52	971	81	84	112
Twitter	–	–	7,189	9,180	17,144

Table 5.3: Indexing Time [s], Comparison to state-of-the-art distance computation frameworks

Streaming Index Construction

The hybrid algorithm used for index construction offers substantial benefits over the traditional approach of Bfs with random accesses used in Gubichev et al. (2010). As shown in Figure 5.11 (note the logarithmic scale of the y-axis), the hybrid algorithm offers a speedup of an order of magnitude over the traditional algorithm in the vast majority of considered cases. Compared to a streaming-only approach (shown as orange bars in Figure 5.11), the hybrid algorithm is significantly faster, up to a factor of 3.3 for the case of the BerkStan webgraph. On average, for the considered datasets (all graphs except for the massive graphs Twitter and UK, which turned out to be too demanding for the traditional algorithm) and setting $\mathrm{DP}_{k=25}$, the hybrid algorithm is faster than the pure streaming approach by a factor of 1.7, and faster than the traditional approach by a factor of 16, and is thus used as the standard index construction algorithm in our implementation. In this experiment we compute 10 shortest path forests at a time for the streaming-only as well as the hybrid algorithm. The speedup obtained by the streaming- and hybrid algorithm over the traditional index construction is quite high even though the size of the considered graphs is smaller than the available main memory and the graph is memory-mapped. This is due to the fact that the overhead associated with random accesses to the graph, i. e. retrieving the correct page, uncompressing all page contents, and extracting the relevant parts, is still high, even when the graphs can be held in main memory. In combination with the benefits obtained from caching contiguous pages in the memory hierarchy, sequential access to the edge list outperforms random accesses even though the latter do not (always) induce disk seek operations.

Comparison with Baseline Strategies

In Table 5.3 we summarize the indexing times for the various baseline algorithms (in seconds), compared to the Path Sketch index for different seed selection strategies with cardinality 25, and random exponential seed set selection with $K = 2$ rounds. While both Path Sketch and IS-Label provide competitive index construction performance, the results vary greatly on individual datasets. As an example, while the Path Sketch index requires 116 seconds (DP-25) to index the Google dataset, IS-Label finishes index construction already after 56 seconds. On the other hand, the Path Sketch index requires 591 seconds to create the index over the Wikipedia dataset, while IS-Label needs 8,218 seconds for indexing.

Regarding scalability to massive graphs, as expected for an exact approach, we found that IS-Label could not index the Twitter and UK datasets within the allotted time limit of 2 days. The PLL implementation for directed graphs was only able to index 4 of the datasets within the allotted time limit (1 day for the small datasets, 2 days for Twitter and UK). For the UK dataset, loading the graph itself entirely into main memory in uncompressed form led to memory exhaustion even before starting index construction of PLL. While for the DBLP dataset, PLL required much more time for indexing than both Path Sketch and IS-Label, it exhibited the best performance for both Slashdot and BerkStan.

5.9.5 Index Size

We now consider **index size**, i. e. the total size of the resulting disk-resident index structure that is used in the query processing stage. The Path Sketch index comprises of files containing the path sketch trees and the offset files. In our experiments, as described in Section 5.5.2, the direct successor/predecessors of each vertex are included in the index entries, and thus the whole input graph is implicitly contained in the index entries. We first compare the index sizes for the different seed selection strategies, and then discuss the space requirements of the different proposed tree serialization approaches.

Indexing Strategies

The (on-disk) index size, including the offset and index entry files, is summarized in Table 5.4, for adjacency list encoding using variable byte-length compression. While the selection strategies CC (*Closeness*) and DP (*DegreeProduct*) can be compared easily, the number of seed sets used in the *random exponential* selection depends on the graph size rather than being specified as an input parameter. For $k = 10$ seeds, the computed index size ranges from 10 MB (CC, DP) for the Slashdot social network (graph size 4 MB) to 75 GB (CC) and 106 GB (DP) for the largest considered graph (UK web graph, size 8.4 GB). We observe that for the strategies CC and DP the index size exhibits a linear dependence on the number of selected seeds for the increase from 10 to 25 for most of the graphs. This effect levels off for a larger number of seeds, resulting in sublinear growth of index size in the number of selected seeds. Overall, it becomes clear that the Path Sketch index structure easily

Dataset	*Closeness (CC)*				*Degree Product (DP)*				*Random Exponential (RE)*			
	$k = 10$	$k = 25$	$k = 50$	$k = 100$	$k = 10$	$k = 25$	$k = 50$	$k = 100$	$K = 1$	$K = 2$	$K = 5$	$K = 10$
Wikipedia	1,202	2,065	3,392	5,598	1,178	2,143	3,528	6,017	1,867	3,033	6,081	10,580
Google	281	622	1,049	1,692	304	585	980	1,762	365	651	1,339	2,271
BerkStan	120	152	183	324	148	195	302	535	292	488	945	1,871
UK	75,193	118,420	64,998	–	106,015	198,450	329,477	–	116,138	–	–	–
Slashdot	10	17	25	41	10	18	27	44	15	25	44	69
DBLP	222	410	676	1,120	223	424	689	1,213	391	573	1,304	2,272
Twitter	10,521	13,755	18,714	26,359	12,413	20,095	27,304	37,662	20,241	33,229	58,406	102,155

Table 5.4: Index Size [MB]: Comparison of Seed Strategies, PathSketch (AL+Delta, compressed)

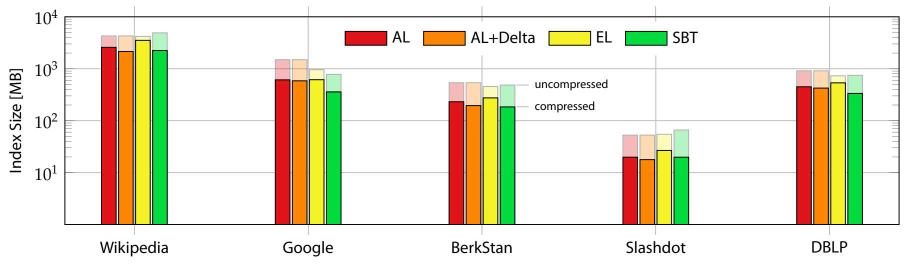


Figure 5.12: Index Size [MB]: Comparison of Serialization Approaches, Path Sketch (DP, $k = 25$) – (lower bars correspond to variant using variable byte-length encoding)

fits on standard hard drive sizes, with a maximum observed index size of 330 GB for the UK web graph with 3 billion edges for 50 seeds ordered by degree product.

Serialization Approaches

In Figure 5.12 we give an overview of the effect of the serialization techniques on the overall index size for the setting (DP-25) on the 5 smaller datasets. As described in Section 5.7.1, we consider adjacency list (AL), sorted adjacency list with delta encoding for gaps between subsequent entries (AL+Delta), edge list (EL) and stack-based traversal (SBT). Further, we consider plain as well as variable byte-length (VBE) encoding of the integers. We observe that plain edge-list encoding results in larger index size, whereas stack-based traversal encoding results in the smallest observed indices. In comparison with original graph size, for the choice of seeds by strategy DP-25, the index constructed by edge-list encoding requires on average 16 times more space than the graph, whereas for stack-based traversal this factor reduces to 10. The two adjacency list variants require 14 (AL) to 13 (AL+Delta) times the size of the input graph. Variable-byte length encoding has a major impact on index size, resulting on average in a compression ratio of 2:1. Note that the achieved compression does not incur a penalty on indexing time, rather the indexing time is slightly improved by the smaller amount of data that has to be written to disk. Interestingly, stack-based traversal encoding not only leads to the smallest index sizes on average, but allows fast index construction as well. On average, SBT encoded indexes can be constructed as fast than adjacency list encoded index entries, and slightly faster than edge-list encoded path sketch trees.

Dataset	IS-Label	PLL	PS (CC-25)	PS (DP-25)	PS (RE-2)
Wikipedia	1,941	–	2,065	2,143	3,033
Google	159	2,958	622	585	651
BerkStan	168	1,989	152	195	488
UK	–	–	118,420	198,450	
Slashdot	18	196	17	18	25
DBLP	123	5,742	410	424	573
Twitter	–	–	13,755	20,095	33,229

Table 5.5: Index Size [MB], Comparison to state-of-the-art distance computation frameworks

Comparison with Baseline Strategies

In Table 5.5 we show the index sizes for IS-LABEL, PLL, and the Path Sketch variants with cardinality $k = 25$ (for DP,CC) as well as $K = 2$ indexing rounds (RE). The index sizes of IS-LABEL and Path Sketch exhibit the same order of magnitude. In general, given that IS-LABEL is an exact distance computation framework, its index sizes are remarkably compact. The index sizes for the computed PLL indexes are significantly larger than both the IS-LABEL and Path Sketch indexes.

5.9.6 QUERY PROCESSING

We now evaluate the query processing performance in several dimensions: query processing time, accuracy of the estimated distance (and corresponding path), and the number of paths generated. For benchmarking, we use a set of 1000 reachable pairs of vertices, that are executed back-to-back. The metric we use to evaluate the accuracy of distance estimates is a measure of (micro-averaged) relative estimation error.

Processing Time

The reported query processing times correspond to the average time per individual query over the 1000 input queries. As described above, for both PLL as well as Path Sketch, we use offset-based indexing of the individual index entries. The offsets are loaded into main memory prior to query execution. In the experiments, we use warm filesystem caches for all strategies, that is, we execute the experiments five times back-to-back (including index initialization) and report the best run. In Table 5.6 we compare the query processing performance of IS-LABEL, PLL, the Path Sketch index (for different seed selection strategies and budgets of 0 and 10 additional vertex expansions, respectively, using budgeted expansion by vertex level), and the baseline strategies BFS (breadth-first search) and BBFS (bidirectional breadth-first search). Regarding query processing time, the Path Sketch variants without expansion budget ($\beta = 0$) are extremely fast while maintaining a good estimation accuracy (which is discussed in detail in the subsequent section). We set the timeout for processing the 1000 benchmark queries to 10 hours (corresponding to 36 seconds per query), which led to the abortion of the plain BFS algorithm over

Dataset	IS-Label	PLL	PS (CC-25)				PS (DP-25)				PS (RE-2)				BFS	BBFS
			$\beta = 0$		$\beta = 10$		$\beta = 0$		$\beta = 10$		$\beta = 0$		$\beta = 10$			
			QT	ϵ	QT	ϵ	QT	ϵ	QT	ϵ	QT	ϵ	QT	ϵ		
Wikipedia	25.32	–	0.12	0.07	0.86	0.06	0.12	0.07	0.86	0.05	0.11	0.43	0.83	0.37	–	26.54
Google	0.37	0.03	0.15	0.05	0.59	0.04	0.14	0.04	0.60	0.04	0.13	0.18	0.54	0.18	7,459.57	88.28
BerkStan	2.29	0.02	0.08	0.01	0.52	0.01	0.09	0.01	0.53	0.01	0.15	0.01	0.56	0.01	5,286.53	202.38
UK	–	–	0.21	0.10	1.62	0.10	0.29	0.18	1.66	0.18	–	–	–	–	–	4,026.59
Slashdot	0.28	0.03	0.09	0.07	0.43	0.02	0.10	0.07	0.43	0.02	0.07	0.19	0.41	0.04	460.23	1.70
DBLP	0.84	0.06	0.10	0.12	0.57	0.11	0.10	0.12	0.57	0.11	0.09	0.23	0.55	0.22	6,451.58	10.36
Twitter	–	–	0.12	0.09	11.80	0.07	0.13	0.05	9.61	0.04	0.13	0.19	9.32	0.14	–	61.08

Table 5.6: Query Processing: Time [ms]/Relative Error, PathSketch (AL+Delta, compressed), budgeted expansion by vertex level (VL)) and Baselines

3 datasets (Wikipedia, UK, Twitter). The execution time of Path Sketch increases steeply as the expansion budget is increased, which however leads to better estimation accuracy and a larger number of synthesized paths. IS-Label offers compelling query execution time for the datasets it was able to index. In concordance with the previous section, when regarding the seed selection strategy DP-25 (top-25 vertices by degree product), the best query processing performance of IS-Label (Slashdot dataset) corresponds to twice the time consumed by Path Sketch (with no additional vertex expansions), and returns the correct distance, while Path Sketch overestimates the correct distance between 1% to 18% relative error. In the worst-case for IS-Label (Wikipedia dataset), Path Sketch is able to answer the queries on average 200 times faster, at an approximation error of 7%. As expected, a higher number of vertex expansion leads in most cases to a significant improvement of estimation accuracy, however at a considerable impact on the query processing time. For the datasets that could be indexed by PLL, it exhibits exceptionally good query processing times, answering queries several times faster than the best Path Sketch and IS-Label variants. However, as mentioned before, only 4 out of 7 datasets could be indexed using this strategy. Bidirectional breadth-first search provides huge improvement over a plain BFS expansion, as can be seen from the query processing times, typically in the range of tens of milliseconds per query. For the case of the UK webgraph dataset, BBFS query processing amounts to roughly 4 seconds per query, while the plain Path Sketch procedure answers the queries with just two index lookups as fast as 0.29 ms per query (at 18% error). Again, as the expansion budget is increased, the query execution time of Path Sketch becomes slower. The reason lies both in the time spent on random accesses to the graph but of course also on the increased work done in online processing, where we have to check the containment of the neighbors of expanded vertices in the respective other tree. In addition the generation of a large number of paths has an impact on query processing times as well, because of the effort to extract paths from the trees as well as making sure that there are no duplicates among the generated paths.

Accuracy

In this section we discuss the estimation accuracy of the Path Sketch index, since it provides *approximate shortest paths*, whereas IS-Label and PLL compute the *exact distance* between the query vertices. In order to assess the relative approximation error $\epsilon(Q)$, we define

$$\epsilon(Q) = \frac{\sum_{(s,t)\in Q} \tilde{d}(s,t) - d(s,t)}{\sum_{(s,t)\in Q} d(s,t)}, \tag{5.24}$$

where Q denotes the set of queries. Table 5.6 gives an overview of the relative error obtained by Path Sketch for different seed strategies and budgets, using expansion by vertex level. Starting with query processing without additional vertex expansions (budget $\beta = 0$), we find that already in this fastest query processing variant, the obtained approximate paths have a very high quality, ranging from relative error between 1% (BerkStan webgraph) to at most 18% (UK) for DP-25. In general, the obtained path accuracy does not differ vastly for the seed selection strategies

Dataset	$k = 10$				$k = 25$				$k = 50$			
	QT	ϵ	P	A	QT	ϵ	P	A	QT	ϵ	P	A
Wikipedia	0.06	0.11	10	1.00	0.12	0.07	25	1.00	0.23	0.05	50	1.00
Google	0.07	0.10	9	0.98	0.14	0.04	21	0.98	0.27	0.02	39	0.98
BerkStan	0.05	0.01	7	0.86	0.09	0.01	17	0.86	0.16	0.01	27	0.87
UK	0.18	0.32	16	0.94	0.29	0.18	26	0.94	0.44	0.11	41	0.94
Slashdot	0.05	0.11	10	1.00	0.10	0.07	25	1.00	0.19	0.05	50	1.00
DBLP	0.05	0.16	9	1.00	0.10	0.12	22	1.00	0.19	0.09	43	1.00
Twitter	0.08	0.07	10	1.00	0.13	0.05	25	1.00	0.22	0.03	50	1.00

Table 5.7: Query Processing: Time [ms] (QT), Error (ϵ), Number of Paths (P), Fraction of Answered Queries (A), PathSketch (DP-25, AL+Delta, compressed), budget $\beta = 0$)

Dataset	$k = 10$				$k = 25$				$k = 50$			
	QT	ϵ	P	A	QT	ϵ	P	A	QT	ϵ	P	A
Wikipedia	1.80	0.04	11	1.00	1.97	0.02	26	1.00	2.14	0.01	51	1.00
Google	2.55	0.06	9	0.99	3.05	0.03	21	0.99	3.59	0.01	39	0.99
BerkStan	4.87	0.01	7	0.92	5.29	0.01	17	0.92	5.19	0.01	28	0.92
UK	72.82	0.17	16	0.97	67.88	0.08	26	0.97	65.41	0.05	41	0.97
Slashdot	0.38	0.03	11	1.00	0.43	0.02	26	1.00	0.52	0.02	50	1.00
DBLP	1.28	0.08	9	1.00	1.49	0.05	22	1.00	1.68	0.03	44	1.00
Twitter	18.35	0.03	11	1.00	15.70	0.02	26	1.00	17.60	0.02	50	1.00

Table 5.8: Query Processing: Time [ms] (QT), Error (ϵ), Number of Paths (P), Fraction of Answered Queries (A), PathSketch (DP-25, AL+Delta, compressed), budget $\beta = \infty$)

by closeness centrality (CC) and degree product (DP), especially as the expansion budget is increased. Since seed selection by degree product can be obtained much easier and faster, we advocate the use of the selection by vertex degree product as the default strategy of Path Sketch. In contrast to these singleton seed strategies, the number of seed sets for the random exponential (RE) strategy depends logarithmically on the size of the graph measured by the number of vertices. Overall, the approximation quality of this alternative seed selection strategy turns out to be inferior when compared to the other methods, while requiring a larger index size.

In Tables 5.7 and 5.8 we compare the extreme settings of budgeted expansion by vertex level, that is for budget $\beta = 0$ (no vertex expansions) and $\beta = \infty$ (continued expansions until no better path can be found). It becomes clear, that the approximation quality increases for larger number of seeds, as expected. However, larger index entries also incur an increase in query processing time. Using 25 seed vertices, the index entries alone are sufficient to achieve approximation errors below 10% for all datasets. For an unlimited expansion budget, we achieve a very high approximation quality with relative error of at most 8% for all datasets for 25 and at most 5% for 50 seeds (DP).

Number of Paths

In the last part of our experimental evaluation, we discuss the number of computed paths in more detail. As we have seen in the last section (cf. Tables 5.7,5.8), for query processing without vertex expansions (budget $\beta = 0$), the number of generated paths typically corresponds to at most (and often exactly to) the number of used seed vertices. The number of paths discovered increases significantly when vertices in the Path Sketch trees are expanded, culminating in as many as $120,575$ (RE-2) and $2,694$ (DP-25) paths using a budget of $\beta = 5$ expansions by vertex degree for the Twitter dataset. We further find that for social networks we can typically generate a much larger number of connecting paths than for other datasets such as web graphs. As shown in Table 5.9, expansion by degree leads to a much larger number of generated paths than expansion by distance to the query vertices. We thus propose the use of expansion by vertex level for the generation of few, high quality paths, and the use of expansion by vertex degree for the scenario where a large number of connecting paths is desired. As mentioned before, in theory it may happen that for a pair of vertices that is connected in the input graph, no path is found by the Path sketch index. We report the fraction of such cases in Table 5.9. In the vast majority of cases a path is found by the Path Sketch index. Using 25 seeds by degree product, all queries can be answered for the case of Wikipedia, Slashdot, DBLP and Twitter. For the web graphs, we can answer 92% of the queries for BerkStan, 99% of the queries over the Google graph, and 97% of the queries over the UK web graph. The fraction of missed paths further decreases if more seeds are used, as can be seen from Table 5.9 where we report the numbers for the (larger number of) random exponential seeds. In this setting, using expansion by vertex degree, all queries can be answered in all datasets except for UK. Finally, it should be remarked that for the Path Sketch variant employing budgeted expansion by degree, we compute the vertex degrees on the fly prior to query execution. Thus, the adjacency lists will already be cached before the queries are executed, giving some advantage to this variant over the level-wise expansion strategy.

Summary and Discussion

In summary, the Path Sketch index structure provides very fast query processing at high accuracy while satisfying a given vertex expansion budget. Major strengths of the Path Sketch index structure are its robustness and scalability, which allows to index graphs comprising billions of edges, which could not be processed using recently proposed exact distance computation methods. Further, Path Sketch can be adapted to a large number of scenarios, most notably, the computation of many connecting paths. For smaller and medium sized graphs, both PLL as well as IS-Label provide compelling precomputation and query processing performance, however only compute the distances rather than paths. While, in principal, both methods can be extended to compute actual paths, the additional overhead will have negative impact on indexing time, index size, and query processing performance.

Dataset	PS (CC-25)								PS (DP-25)								PS (RE-2)							
	VL, $\beta = 5$				VD, $\beta = 5$				VL, $\beta = 5$				VD, $\beta = 5$				VL, $\beta = 5$				VD, $\beta = 5$			
	QT	ϵ	P	A	QT	ϵ	P	A	QT	ϵ	P	A	QT	ϵ	P	A	QT	ϵ	P	A	QT	ϵ	P	A
Wikipedia	0.49	0.07	25	1.00	1.10	0.03	68	1.00	0.49	0.07	25	1.00	1.10	0.03	72	1.00	0.48	0.42	6	1.00	2.68	0.08	207	1.00
Google	0.36	0.04	20	0.99	0.42	0.03	21	0.99	0.36	0.04	21	0.99	0.41	0.03	22	0.99	0.34	0.18	5	1.00	0.43	0.08	12	1.00
BerkStan	0.30	0.01	18	0.91	0.39	0.01	19	0.92	0.31	0.01	17	0.91	0.43	0.01	22	0.92	0.37	0.01	10	1.00	0.41	0.01	12	1.00
UK	0.47	0.10	24	0.94	1.81	0.09	25	0.97	0.55	0.18	26	0.94	2.14	0.15	29	0.97	–	–	–	–	–	–	–	–
Slashdot	0.39	0.03	25	1.00	0.57	0.03	90	1.00	0.39	0.03	25	1.00	0.60	0.04	99	1.00	0.37	0.08	12	1.00	1.25	0.05	328	1.00
DBLP	0.33	0.11	22	1.00	0.39	0.09	34	1.00	0.34	0.11	22	1.00	0.40	0.09	35	1.00	0.33	0.23	8	1.00	0.40	0.11	27	1.00
Twitter	2.98	0.08	25	1.00	14.24	0.03	1,916	1.00	2.96	0.05	25	1.00	12.23	0.02	2,694	1.00	2.97	0.18	9	1.00	715.04	0.04	120,575	1.00

Table 5.9: Query Processing: Time [ms] (QT), Error (ϵ), Number of Paths (P), Fraction of Answered Queries (A), PathSketch (AL+Delta, compressed), Comparison of budgeted expansion strategies (VL: vertex level , VD: vertex degree)

5.10 Summary

The Path Sketch index structure provides a budget-aware framework for approximately answering shortest path queries, that scales to massive graphs with billions of edges. The index consists of collections of paths from and to designated seed vertices that are stored at the individual vertices. At query time, the paths assigned to the query source and target are combined to generate multiple short paths as candidate answers. The query processor optionally expands a limited number of selected vertices in order to improve the obtained short paths by detecting potential shortcuts.

The distance estimates and corresponding paths can be computed very fast, with query times of less than a millisecond at small relative error when compared to the true distance. Our new indexing algorithms incorporate fast traversal methods that avoid costly random accesses. Regarding query processing, the approximation quality of our algorithms can be further improved by allowing a small budget of random accesses to the graph, providing direct control over the tradeoff between approximation quality and query processing time. The index can be computed efficiently, requiring less than 3 hours to index a large social network with 1.5 billion edges, and consuming only 20 GB of disk space.

6
Relatedness Cores

Q4: How can the relationship be characterized?

«Which European politicians are related to politicians in the United States and how?»

«How can one summarize the relationship between China and countries from the Middle East over the last five years?»

6.1 Problem Definition

In the previous two chapters, we have discussed in detail how the important algorithmic primitives of reachability and distance as well as path approximation can be computed quickly, in order to support relationship analysis over large-scale graphs. In this chapter, we shift our focus from a purely efficiency-based point of view towards facilitating semantically more meaningful analysis of relationships.
For this purpose, we focus on knowledge graphs, wherein (semantically typed) entities are connected via semantic relationships. Such graphs have become a major asset for search, recommendation, and analytics. Prominent examples are the Google Knowledge Graph[1], the Facebook Graph, and the Web of Linked Open Data[2], centered around public knowledge bases like DBpedia (Auer et al., 2007), Freebase (Bollacker et al., 2008), and YAGO (Hoffart et al., 2013, Suchanek et al., 2007).

In this chapter, we specifically address the problem of *characterizing the relationship between two sets of entities* in a knowledge graph:

Problem: Relationship Characterization

Given	Knowledge graph K, consisting of vertices corresponding to entities, that are connected via semantically typed edges, together with two subsets of entities Q_1, Q_2.
Goal	«Characterize» the relationship between Q_1 and Q_2.

As an example, the goal of a relationship characterization algorithm over a knowledge graph containing facts about real-world entities – such as people, countries, organizations, etc. – is to answer questions of the form «*Which European politicians are related to politicians in the United States, and how?*» or «*How can one summarize the relationship between China and countries from the Middle East over the last few years?*»

In the remainder of this chapter, we describe an algorithm – coined Espresso – to compute semantically meaningful substructures (so-called relatedness cores) from the knowledge graph as answers to above questions. In our setting, a question is specified by means of two sets of query entities. These sets of entities (e. g. *European politicians* or *United States politicians*) can be determined by an initial query, e. g., expressed in the SPARQL language for RDF data, the Cypher query language for property graphs (see Chapter 3, Sections 3.1.2 and 3.1.3), or simply enumerated. As a first step, we analyze the (indirect) relationships that connect entities from both sets (e. g., membership in organizations, statements made on TV, etc.), generate an informative and concise result, and finally provide a user-friendly explanation of

[1] `http://www.google.com/insidesearch/features/search/knowledge.html`
[2] `http://lod-cloud.net`

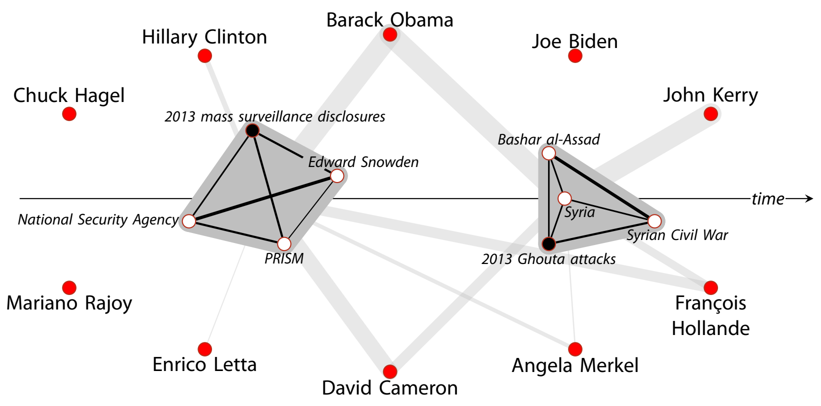


Figure 6.1: Relatedness Cores for Politicians from US and Europe

the answer. As output we aim to return concise subgraphs that connect entities from the two sets and explain their relationships.

As an example, consider the graph depicted in Figure 6.1, which demonstrates the desired output of a relationship characterization system. Here, we are interested in characterizing the political relationship between European countries and the United States. For this purpose, compact and coherent subgraphs from the proximity of important politicians from the specified countries are displayed. Each such subgraph corresponds to a *key event* that is highly relevant to at least one entity from each of the two input sets. These subgraphs form the core of the relationship explanation; we thus call them *relatedness cores*. The full answer to the user query can be derived by connecting relatedness cores with the entities in the two input sets.

6.1.1 Problem Characteristics

Previous work on relationship explanation over graphs has mainly focused on computing the "best" subgraph providing a connection among the entities in a single query set. Typically, in this setting, algorithms operate in two stages. In the first stage, vertices and/or edges are assigned a score signifying their informativeness or relatedness to the query entities. In the second stage, a solution (subgraph connecting the query vertices) is constructed based on the previously assigned scores. We briefly review the proposed approaches:

- Faloutsos et al. (2004) address the problem of computing a *connection subgraph* between two vertices in a large social network. A connection subgraph is defined as a small (i. e. satisfying a budget on the size) subgraph containing the query vertices that maximizes a goodness measure. This measure is based on a notion of "flow" from source vertex to target vertex.

- Tong and Faloutsos (2006) generalize the notion of connection subgraphs to the case of $k \geq 2$ query vertices. The goal is to extract *centerpiece subgraphs*

(CePS), size constrained-graphs that connect all or $k' \leq k$ of the query vertices and maximize a goodness score. This approach is based on the identification of *centerpiece vertices*, based on aggregated scores from random walks with restart (RWR) from each query vertex, followed by a dynamic programming approach to extract the best paths.

- In Ming, Kasneci et al. (2009a) study the extraction of informative subgraphs connecting k user-specified query entities, focusing on entity-relationship graphs. Their approach relies on deriving a notion of informativeness, based on the assignments of edge weights computed from co-occurrence statistics of entity pairs in a Web corpus, followed by the computation of a connecting Steiner tree. The approach is complemented by RWR processes over the edge-weighted graph to establish informative connections.
- Fang et al. (2011) specifically address the problem of *explaining* the relationship between a pair of entities. The proposed Rex algorithm is applied in the context of Web search, where entities that are related to a keyword query entity are displayed. Relationship explanations are used to provide information why each of the displayed related entities was deemed important. A relationship explanation corresponds to an instance of a graph pattern that was identified as most interesting and connects the entity pair. The interestingness of a pattern is determined based on aggregate measures (such as number of instances of a pattern) as well as distributional measures (e. g. rarity of a pattern for the entity pair).

In this chapter we deviate from previous approaches for relationship explanation, since we are interested in the relationship between *two sets of query entities*, Q_1, Q_2, rather than among a set of entities. Similar to the prior work discussed above, we view the problem of explaining the relationship between entities as an issue of computing an informative subgraph. Our output is not focused on a single coherent subgraph, but aims to find multiple sub-structures in the knowledge graph that are highly informative.

To this end, we adapt and extend the above-mentioned CePS framework for computing good subgraphs connecting entities contained in a single query set to the case of two query sets in knowledge graphs. As a building block along these lines, we define a notion of *relationship centers*, intermediate vertices that play an important role in the relationship between entities from either set. Relationship centers play a role similar to centerpiece vertices in CePS, with the difference that relationship centers are *informative* vertices (by entity type restriction) for the relationship *between two sets* rather than among a single set. To answer a query, we identify a set of relationship centers, connect them to (subsets) of the query entities, and subsequently expand the center vertices into *relatedness cores*: concise subgraphs that represent key events connected to both input sets of entities. This outlined method can be seen as the approximation of a combinatorial optimization problem specified over the underlying graph and input entity sets (to be explained in Section 6.3), extended to incorporate considerations regarding the semantics of the generated solution, which cannot be easily compressed merely in terms of graph-theoretic properties.

6.1.2 Contribution

The salient contributions we make in this chapter are as follows.

1. We introduce a model for the novel problem of explaining relationships between two intensionally specified sets of entities, in contrast to the previously studied problem of explanations for a pair of single entities or among a single set of query entities,

2. adapt and extend the CePS framework into a heuristic algorithm for computing explanations of the relationship between two sets (based on the notions of relationship centers and relatedness cores) in a scalable manner,

3. discuss additional scenarios such as the integration of user-specified constraints on the computed solutions and the integration of temporal information,

4. and present an experimental evaluation based on user assessments on the usefulness of the generated explanations, as well as experiments highlighting the efficiency of the heuristic algorithm.

We compute the solutions over an enriched entity-relationship graph derived from existing knowledge bases and integrating additional data sources.

The remainder of the chapter is organized as follows. In Section 6.2 we discuss the knowledge graph we use to compute relationship explanations. Section 6.3 presents our computational model.

In Sections 6.4, 6.5, and 6.6 we present our heuristic algorithm for scalable computation of relationship explanations, introducing the two major building blocks of our solution, relationship centers and relatedness cores. Section 6.7 extends our setting to considering temporal aspects in the relatedness of entities. Section 6.8 discusses related work. Section 6.9 presents experimental results, followed by a summary of the chapter.

6.2 The Espresso Knowledge Graph

The knowledge graph we use as input to our relationship characterization algorithm – in this chapter referred to as *Espresso Knowledge Graph* – is derived from the YAGO2 (Hoffart et al., 2010) and Freebase (Bollacker et al., 2008) knowledge bases. More specifically, the set of entities we model as vertices are exactly the entities present in the intersection of YAGO2 and Freebase. This way, we can discard many concepts present in YAGO2 that do not correspond to actual entities, for example overview pages from Wikipedia such as discographies, summaries (*2013 in film*), etc. Two entities (e_1, e_2) in the Espresso Knowledge Graph are connected via an undirected edge, if the corresponding Wikipedia article pages describing the respective entities contain an intra-wiki link in either direction. Every edge is assigned the relationship label *relatedTo* as well as additional labels, corresponding to the relation names for each fact contained in YAGO2 between the respective entities.

The Espresso Knowledge Graph contains a total of 3,674,915 vertices (corresponding to entities), connected via 57,703,180 relationships labeled with one of 30 possible relationship names.

We enrich this knowledge graph by integrating several additional data sources, including

- edge weights signifying the relatedness between entities, derived from structural properties (inlink overlap, Milne and Witten (2008)) , textual descriptions of the entities based on the Wikipedia article text (Hoffart et al., 2012), and co-occurrence in the ClueWeb12 corpus (Gabrilovich et al., 2013),
- semantic types associated with the entities, derived from YAGO2 (Wikipedia categories, WordNet classes) and Freebase (types). Every entity is assigned one or more of the 508,356 YAGO types and one or more of the 7,513 Freebase types. On average, an entity is associated with 15.8 YAGO types and 3.3 Freebase types, translating to a total of 58,023,593 entity-YAGO type and 12,278,102 entity-Freebase type relationships.
- the popularity of individual entities over time, extracted from the page view statistics of the respective Wikipedia pages [3], a total of 2,827,668,135 triples of the form (entity,day,views) reflecting the page view counts of each entity in daily granularity in the time frame 01/01/2012 to 07/31/2014.

The Espresso algorithm described later in this chapter relies on identifying query-relevant entities of informative types, e. g. – as used in the experimental evaluation of this chapter – entities of type *event*. In principle, events can be identified by considering all entities typed Event (WordNet class) in YAGO or `/time/event` in Freebase. However, in order to increase both precision as well as recall, we employ a machine learning approach to identify entities of the event type. For this purpose, we have trained a linear SVM classifier by manually identifying 1,786 entities corresponding to real-world events from a pool of 28,000 training examples. The features used to classify an entity are the entity name and short snippets describing the entities (extracted from Freebase). We use the model to classify each of the 3.6 million entities as either *event* or *not event*, resulting in a total of 89,321 entities marked as event.

This rich, integrated data collection is made publicly available for further studies on relationship analysis[4].

[3] `http://dumps.wikimedia.org/other/pagecounts-ez/`
[4] `http://espresso.mpi-inf.mpg.de/`

6.3 Computational Model

This chapter is based on the notion of a knowledge graph, introduced in Chapter 2, that we now define formally:

Definition 6.1 (Knowledge Graph). *A knowledge graph, denoted by*

$$K = (V, E, \ell_V, \ell_E, T, R),$$

is a labeled graph with a set V of entities as vertices, a set E of relationships as edges, and two labeling functions $\ell_V : V \rightarrow 2^T$ and $\ell_E : E \rightarrow 2^R$ that assign to each vertex a set of type names (from a set T of possible types) and to each edge a set of relation names (from a set R of possible relations) as labels. Each edge e and each vertex v can optionally have weights $\omega(e)$ and $\omega(v)$.

Example. As an example, the vertex *Hillary Clinton* in Figure 6.1 may have type labels *female politician*, *US Democratic Party member*, *US Secretary of State*, *First Lady of the United States*, and more, and the edge between *Hillary Clinton* and *Barack Obama* could have relation labels *competed with* (in US presidential primaries) and *served under* (in the US government).

Starting with two sets Q_1 and Q_2 of entities, such as *United States politicians* and *European politicians*, the explanation of their relationships aims to identify one or more key events that connect the two sets. Key events are themselves groups of strongly interrelated entities. We call the corresponding subgraphs S_i *relatedness cores*, or *cores* for short. As a general framework, we place the following desiderata to exemplify what constitutes a "good" core:

- **Informativeness**
  1. Critical: The core S_i is important, i. e. it describes an important event or topic.
  2. Comprehensive: The core S_i is self-explanatory, i. e. gives insight into the topic without requiring further explanation.
  3. Compact: The core S_i does not overload the user with information by satisfying an upper bound on the size, i. e. the number of contained vertices.
  4. Coherent: The core S_i describes a single event or topic rather than a mixture of topics. For this purpose, the entities contained in the core should be strongly interrelated, corresponding to a subgraph with high pair-wise edge weights or high degrees within S_i.
- **Relevance**
  1. The core is relevant to some or all query entities, i. e. S_i should be highly related to both entity sets Q_1 and Q_2, for example in terms of edges or paths between S_i and each of the two query sets, total or average edge weight of paths connecting them, or another way that captures relatedness between vertices $v \in S_i$ and query vertices $q \in Q_1 \cup Q_2$.

2. Temporal relevance: The topic was important/the event happened during a user-specified time interval.

Further desiderata can be introduced, for example controversiality of the topic, following the intuition that points of disagreement (conflicts) give more insight into the dynamics of a relationship than general events, as well as the diversity of computed cores.

One way of formalizing these desiderata into a computational model is to aim for identifying one or more subgraphs as relatedness cores with high edge weights (i) within each subgraph and (ii) regarding the connections to each of the two input sets. When considering paths between some core S_i and a query set Q_1 or Q_2, we need to combine edge weights by an aggregation function like average, maximum, or minimum applied to the shortest path (or K shortest paths) between vertices in S_i and all or some of the vertices in Q_1 or Q_2. We denote the path score between vertices x and y as $\omega(x \overset{\star}{\leftrightarrow} y)$. Formally, we cast these ideas into the following *edge-weighted relatedness cores* problem.

Definition 6.2 (Edge-Weighted Cores Problem).
Given an edge-weighted knowledge graph $K = (V, E, \ell_V, \ell_E, T, R)$, two query sets $Q_1 \subset V$ and $Q_2 \subset V$ and a budget B, compute up to k connected subgraphs $S_1 = (V_1, E_1), \ldots, S_k = (V_k, E_k)$ such that

$$\sum_{i=1}^{k} \left(\sum_{e \in E_i} \omega(e) + \sum_{\substack{x \overset{\star}{\leftrightarrow} y, \\ x \in Q_1, y \in V_i}} \omega(x \overset{\star}{\leftrightarrow} y) + \sum_{\substack{x \overset{\star}{\leftrightarrow} y, \\ x \in Q_2, y \in V_i}} \omega(x \overset{\star}{\leftrightarrow} y) \right) = max!$$

with $|V_i| \leq B$ and each $V_i \cap Q_1 \neq \emptyset \neq V_i \cap Q_2$.

Intuitively, in this setting we aim to find up to k coherent cores that explain the relationship between Q_1 and Q_2, while observing a size budget for each core. The density of each S_i is measured by the total weight of edges in S_i, and the relatedness to Q_1 and Q_2 is measured by the total score of paths that connect vertices in S_i with vertices in Q_1 and Q_2, respectively. This problem formulation is well-suited for the scenario where we seek an explanation involving all entities from both query sets. Regarding complexity, the NP-hardness of the edge-weighted cores problem can be derived as follows. Consider the special case of the problem where we set the number of cores, k, to 1 and choose singleton query sets $Q_1 = \{v_1\}$ and $Q_2 = \{v_2\}$. Suppose an efficient algorithm exists for computing the densest subgraph of fixed cardinality B that contains two designated vertices v_1, v_2. Then, we could derive an efficient algorithm for the more general problem of computing a dense subgraph with fixed cardinality B, by running above algorithm for each possible pair of vertices in $V \times V$ and keeping the densest solution. However, the problem of computing the densest subgraph with fixed cardinality has been shown to be NP-hard (Feige et al., 2001).

Above problem formulation considers paths (or, more precisely, distances) between the query and core vertices to assess the relevance. In many cases, it makes more sense to assess relevance by more involved measures (such as proximity derived from random walks). We now consider the case where the individual core vertices are assigned a score signifying their relevance. This leads to the following, combined problem with a tunable coefficient β to balance between relevance and coherence.

Definition 6.3 (Edge- and Vertex-weighted Cores Problem).
Given an edge-weighted knowledge graph $K = (V, E, \ell_V, \ell_E, T, R)$, two query sets $Q_1 \subset V$ and $Q_2 \subset V$ and a budget B, compute up to k connected subgraphs $S_1 = (V_1, E_1), \ldots, S_k = (V_k, E_k)$ such that

$$\sum_{i=1}^{k} \left(\beta \sum_{e \in E_i} \omega(e) + (1 - \beta) \sum_{v \in V_i} \omega(v) \right) = \mathit{max!}$$

with $|V_i| \leq B$ and each $V_i \cap Q_1 \neq \emptyset \neq V_i \cap Q_2$.

By encoding the relevance of core vertices to the query entities in the form of vertex weights, we can support different scenarios, including the case where only subsets of the query sets have to be related to a core.

The special case with $\beta = 0$ (i. e., vertex weights only) is also known to be NP-hard, following from the intuition given for the hardness of the edge-weighted core problem and the fact that the general problem of computing the heaviest vertex-weighted tree spanning k vertices has been shown to be NP-hard (Fischetti et al., 1994).

As vertices and edges have type and relation labels, respectively, we can further extend this family of problems by imposing constraints on the vertices and edges that are eligible for the desired cores. Two important kinds of constraints are the following:

Definition 6.4 (Type covering constraint). *In the edge-weighted, or edge- and vertex-weighted problem, enforce the constraint that the vertices V_i of each core $S_i = (V_i, E_i)$ contain all types of a specified subset of types $T' \subseteq T$, that is:*

$$\bigcup_{v \in V_i} \ell_V(v) \supseteq T'.$$

Definition 6.5 (Relation filtering constraint). *In the edge-weighted, or edge- and vertex-weighted problem, enforce the constraint that the edges E_i of each core $S_i = (V_i, E_i)$ have relation labels that are included in a specified subset of relations $R' \subseteq R$, that is:*

$$\bigcup_{e \in E_i} \ell_E(e) \subseteq R'.$$

More constraints along similar lines can be added to the framework. In particular, when the knowledge graph associates entities and/or relational edges with textual descriptions (e. g., by connecting entities with Wikipedia articles or other high-quality texts), we can also apply *text filter predicates* by requiring that each core contains certain keywords. One variant of this extension could require that all vertices of each core are associated with at least one keyword from a specified set of words. Another variant could require that each core contains at least one vertex that matches at least one of the given keywords. Again, more variations are conceivable, and could easily be added.

6.4 Relationship Centers

The idea of the Espresso algorithm is to identify relatedness cores using a two-stage algorithm. In the first stage, we identify a set of *relationship centers*, intermediate vertices that play an important role in the relationship – similar to centerpiece vertices in CePS (Tong and Faloutsos, 2006). These centers are subsequently connected to the query vertices. In the second stage, the neighborhoods of the best relationship centers are expanded, in order to obtain the *relatedness cores*, concise subgraphs that represent key events connected to both input sets of entities. In this section, we discuss how the relationship centers – i. e. vertices that exhibit a high potential for explaining the relationship between the two input sets of entities – can be computed.

Intuitively, the goal is to identify vertices in the knowledge graph that have strong relations with vertices in both input sets. We adopt the idea of identifying central vertices by performing random walks over the graph (Tong and Faloutsos, 2006). For this purpose, we operate on an entity-relationship graph that we derive from the given knowledge graph and additional statistics. The importance of a vertex for explaining the relationship between the query sets is quantified by assigning each vertex in the graph a numeric score based on the random walk processes. For the example of explaining the connections between European and US politicians, the *PRISM surveillance program*, *G8 summits*, etc. represent entities for which we would desire a high vertex score. In the following, we first define the relatedness graph, and then present our method for random-walk based scoring of vertices.

Entity-Entity Relatedness Graph Our extraction algorithm operates over the edge-weighted Espresso Knowledge Graph described in Section 6.2.
For edge weights, we consider the following similarity measures, based on structural properties (inlink overlap) as well as textual descriptions of the entities (partial keyphrase overlap):

Inlink Overlap Milne and Witten (2008) propose a relatedness measure based solely on structural properties of the entity-relationship graph. In this setting, given entities u, v, the MW-similarity is given by

$$\mathrm{sim}_{\mathrm{MW}}(u,v) = 1 - \frac{\log\left(\max\{|I_u|,|I_v|\}\right) - \log(|I_u \cap I_v|)}{\log(n) - \log\left(\min\{|I_u|,|I_v|\}\right)}, \qquad (6.1)$$

where I_u and I_v denote the set of entities linking to u and v, respectively and n corresponds to the total number of entities in the knowledge base.

Keyphrase Overlap For entities that are prominently covered in Wikipedia with many incoming links, above measure works very well. However, it is less suitable for long-tail entities with few links, and it does not easily extend to entities outside Wikipedia (e. g., entities in social media or specialized domains such as the medical domain). This shortcoming is addressed in the work of Hoffart et al. (2012), that proposes the idea of computing the semantic relatedness of two entities by considering the (partial) overlap of associated keyphrases. Here, every entity u is associated with a set of keyphrases, $P_u = \{p_1, p_2, \ldots\}$, each consisting of one or more terms. Every term is associated with a weight (inverse document frequency) and every keyphrase is weighted with respect to the respective entity, by normalized mutual information. The KORE similarity for entities u, v is then defined as

$$\mathsf{sim}_{\mathsf{KORE}} = \frac{\sum_{p \in P_u, q \in P_v} \mathrm{PO}(u,v)^2 \cdot \min(\varphi_u(p), \varphi_f(q))}{\sum_{p \in P_u} \varphi_u(p) + \sum_{q \in P_v} \varphi_v(q)} \tag{6.2}$$

where $\mathrm{PO}(p,q)$ denotes the weighted Jaccard similarity of a pair of keyphrases:

$$\mathrm{PO}(p,q) = \frac{\sum_{w \in p \cap q} \min(\gamma(w), \gamma(w))}{\sum_{w \in p \cup q} \max(\gamma(w), \gamma(w))}. \tag{6.3}$$

The values $\gamma(w)$ and $\varphi_u(p)$ denote the term weight of w and the weight of the keyphrase p with respect to entity u, respectively. The salient keyphrases can be automatically mined from Wikipedia and/or other sources that provide textual descriptions of entities, such as online communities with user comments on new songs.

The above two measures are just two of many possibilities for entity relatedness. In a recent work, Ceccarelli et al. (2013) cast the problem of determining entity relatedness as a learning-to-rank problem and give an overview over effective features for determining the degree of relatedness between entities.

Given the relatedness measure, we obtain the weighted adjacency matrix of the graph, given by

$$M \in \mathbb{R}^{n \times n}, \quad M[i,j] = \begin{cases} \mathrm{sim}(v_i, v_j) & \text{if } (v_i, v_j) \in E, \\ 0 & \text{otherwise.} \end{cases} \tag{6.4}$$

Following the intuition given by Tong and Faloutsos (2006), we construct the transition matrix for the random walks by introducing

$$\hat{M} = D^{-1/2} M D^{-1/2}, \text{ i. e. } \hat{M}[i,j] = \frac{M[i,j]}{\sqrt{\sum_{k=1}^{n} M[i,k]} \sqrt{\sum_{k=1}^{n} M[j,k]}}, \tag{6.5}$$

where D is the diagonal matrix with entries $D[i,i] = \sum_{k=1}^{n} M[i,k]$. Note that with this use of the normalized graph Laplacian for constructing $\hat{M}$, the column vectors are no longer stochastic (Tong et al., 2006), so we need to add a normalization step after executing a random walk.

In the next subsections we discuss how the transition matrix is used to conduct random walks with restart (RWR) in order to identify vertices relevant to the query. We first describe how CePS employs RWRs to identify centerpiece vertices that are related to query vertices belonging to a single set Q.

The CePS approach The CePS algorithm executes one random walk with restart (RWR) from each query vertex $q \in Q$. As a result, each vertex $v \in V$ in the graph is assigned $|Q|$ scores, $r_q(v), q \in Q$. The random walk computation from $q \in Q$ works as follows:

- We set the starting probability vector $\mathbf{s} \in \mathbb{R}^n$ to $\mathbf{s}[q] = 1$ and $\mathbf{s}[v] = 0, v \in V \setminus \{q\}$.
- At vertex v, with probability $\alpha\hat{M}[v,w]$, we walk to a neighboring vertex w, $\hat{M}[x,y]$ denotes the transition probability from vertex v to vertex w. With probability $1-\alpha$, we jump back to start vertex q.
- To obtain the steady-state probabilities, we iterate the equation

$$\mathbf{x_{i+1}} = (1-\alpha)\mathbf{s} + \alpha\hat{M}\mathbf{x_i}, \quad \mathbf{x}_0 = s. \tag{6.6}$$

until convergence or for a predetermined number of iterations.

The score $r_q(v)$ then corresponds to the steady-state probability $\mathbf{x}[v]$. The individual scores are aggregated into an overall score of the vertex v, depending on the desired scenario. Tong and Faloutsos (2006) distinguish between the scenarios *AND* (all query vertices should be highly related v), *OR* (at least one query vertex should be highly related to v) and *Soft-AND* (at least k of the $|Q|$ query vertices should be highly related to v). Since the score $r_q(v)$ is interpreted as the probability for a random walk particle starting at q to be located at v, in the *AND*-scenario the scores (probabilities) are simply multiplied to aggregate a total score:[5]

$$\text{score}_{\text{CePS-AND}}(v) = \prod_{q\in Q} r_q(v). \tag{6.7}$$

Conversely, for the *OR*-scenario, the aggregated score corresponds to the probability that one or more random walk particles from the individual random walks are located at v:

$$\text{score}_{\text{CePS-OR}}(v) = 1 - \prod_{q\in Q}(1 - r_q(v)). \tag{6.8}$$

With this approach, CePS can identify vertices with high proximity to all or at least one (at least k) of the query vertices. Again, to compute the corresponding scores, it requires to perform $|Q|$ random walks with restart over the input graph.

We now discuss how this framework can be extended in order to identify vertices relevant to two sets of query vertices.

[5] If we use the transition matrix as defined in Equation 6.5, the resulting scores $r_q(v)$ are strictly speaking not probabilities anymore, see also Tong et al. (2006). For this reason, we normalize the scores to sum to 1 by dividing each individual score $r_q(v)$ by $\sum_{v\in V} r_q(v)$ prior to the score aggregation step. This is also the default configuration used in the original implementation of CePS.

Relationship Centrality We present an extension of the CePS method to find vertices highly related to the two sets $Q_1, Q_2 \subseteq V$. Suppose all query entities from either set should be related to the center. Then, an appropriate score of a vertex with respect to the query sets is given by

$$\begin{aligned} \text{score}_{\text{CePS2-AND}}(v) &= \left(\prod_{q \in Q_1} r_q(v)\right)\left(\prod_{q \in Q_2} r_q(v)\right) \\ &= \prod_{q \in Q} r_q(v) = \text{score}_{\text{CePS-AND}}(v), \end{aligned} \tag{6.9}$$

i. e., in this case we can directly apply the previous score aggregation for the case of a single query set. This scenario requires however, that all query entities should be related to the central vertices. In many practical settings, including our envisioned applications, it is a far more common assumption that only certain subsets of the query entities are related. To this end, we propose the following generalization of the CEPS scoring mechanism to deal with two sets of query entities. Continuing the description above, in this scenario we quantify the relatedness of a vertex to the query sets Q_1, Q_2 as the probability that *at least one random walk particle from either set* is located at vertex v. This is expressed in the formula

$$\text{score}_{\text{CePS2-OR}}(v) = \left(1 - \prod_{q \in Q_1} (1 - r_q(v))\right)\left(1 - \prod_{q \in Q_2} (1 - r_q(v))\right). \tag{6.10}$$

We can expect this approach to capture vertices that are in close proximity to subsets of the two query sets. However, while Equation 6.10 is an appropriate generalization of the original CEPS approach to sets, we are still facing the problem of having to conduct $|Q|$ random walks. To this end, Tong and Faloutsos (2006) and Tong et al. (2006) propose substantial improvements over straightforward implementations of RWRs (like partitioning and low-rank matrix approximation), however, very large graphs in combination with many query vertices can remain challenging, especially in our envisioned scenario of an interactive, user-facing system. To overcome this problem, we propose a faster approach to identify candidates for relationship centers, i. e. vertices in close proximity to subsets from both query sets. This approach requires only two random walks with restart, one from each of the query sets. The score derived from the set-based RWR from each of the query sets directly captures the proximity of a vertex $v \in V$ to the set. More precisely, we modify above procedure for computing the scores as follows, where, without loss of generalization, we compute relatedness scores with respect to query set Q_1:

- For each $q \in Q_1$, we set the starting probability to $s[q] = 1/|Q_1|$ and to $s[v] = 0$ for $v \in V \setminus Q_1$.
- As before, with probability $\alpha\hat{M}[v, w]$, we walk from v to w. With probability $1 - \alpha$, we jump back to a vertex in Q_1, *chosen uniformly at random*.
- To obtain the steady-state probabilities, we likewise iterate Equation 6.6 until convergence or for a predetermined number of iterations.

This procedure is performed for both Q_1 and Q_2, resulting in scores $\mathbf{x}_{Q_1}[v]$ and $\mathbf{x}_{Q_2}[v]$ for each vertex v. The relationship center score (RC) of $v \in V$ is the product of the two scores multiplied by a prior $\mathsf{pr}(v)$, derived from the overall importance or prominence of v (e. g., its PageRank score in $\hat{M}$, popularity determined from external sources, etc):

$$\mathrm{RC}(v) = \mathbf{x}_{Q_1}[v] \cdot \mathbf{x}_{Q_2}[v] \cdot \mathsf{pr}(v). \tag{6.11}$$

With this approach for vertex scoring, we can restrict the computational effort to two RWR processes in order to compute the scores. This makes the computation time independent of the input query sizes. In the experimental evaluation of this chapter, we show that for the considered queries the quality of results is comparable to the *CePS2-OR* (Eq. 6.10) approach, while leading to a substantial speed-up of score computation.

As a final remark, recall that among the best vertices according to above score, we aim to select vertices that help explain the relationship. The type(s) associated with the individual vertices can be very helpful to distinguish informative, high-scoring vertices from other, more generic high-scoring vertices. For knowledge graphs containing facts about real-world entities (such as the Espresso Knowledge Graph) we found that limiting relationship center candidates to entities that are (i) relatively prominent and (ii) of type *event* is an effective strategy that works well over a diverse range of queries. However, entity types of informative relationship centers could also be derived in a more principled way, i. e. by measures like mutual information between query entity types and possible candidate types.

6.5 Connection to Query Entities

As a next step, we consolidate relatedness cores by connecting the selected relationship centers with the query entities, and expand the centers into tightly knit sub-graphs. The entities in such a core should be highly related with both Q_1 and Q_2 and also within the core itself. In this section, we discuss the first step, i. e. establishing the connections between the query sets and the center. We remark that in contrast to the original work of CePS, which was motivated by a social network application, in our envisioned querying scenario we expect the case of central entities directly connected to the query sets as far more common. We could even go so far as to say that entity sets that are not immediately connected via one intermediate hop can be deemed as too unrelated, which in many settings constitutes an appropriate answer (retrieval setting). Nevertheless, for the sake of completeness, we discuss how central vertices can be connected to the query sets if there is no direct connection.

In general, establishing indirect connections is a challenging problem for the following reasons: (i) we need to determine what makes for a "good" connection between the center and the query entities, and (ii), given the budget on the number of vertices we include in our solution, we might have to decide for only a subset of the query entities that will be connected to the center. Regarding this problem, the original CePS algorithm proceeds by connecting the identified central vertices with 1 to $|Q|$ (depending on the scenario) of the query entities, employing a dynamic programming algorithm to identify good paths. The score of a path is evaluated as the sum of vertex weights divided by the number of vertices that have to be added to the partially constructed solution. CePS connects query entities in descending order of relatedness to the central vertex, and continues until either a sufficient number of query vertices according to the scenario (1 for *OR*, *k* for *SOFT-AND*, all for *AND*) have been connected to the center, or the available budget has been spent. For the set-based relatedness measure from Equation 6.10, the original CePS subgraph extraction algorithm would connect one vertex from either query set to the next center until the budget is exhausted.

In this section, we describe alternative ways to connect the best central vertices to the query sets, for the following reasons. First, the extraction used by CePS requires knowledge of the scores assigned by individual random walks from the query vertices, which we do not have in Espresso where we just perform two RWRs from the query sets. Second, rather than connecting the central vertex to 1 or k query entities (1 might be too less, while a fixed number k is unrealistic for the user to specify), it might be desirable to fix a certain budget we can spend to connect as many query vertices as possible.

One possibility along these lines is to compute a shortest path tree from the center and connect it to the closest query entities until the budget has been spent. This is the default strategy used by Espresso.

An alternative algorithm we propose is based on (i) computing the relatedness between the relationship center c and the other vertices in the graph, and (ii) computing a cheap (i. e. satisfying the budget) subgraph connecting the relationship center with as many highly related query entities as possible. For the former, we rely on the computation of an additional random walk with restart, this time from the relationship center, c. This results in a score assigned to each vertex v – denoted by $\mathbf{x}_c[v]$. If the relationship center in a real-world knowledge graph is an actual *event*, the score $\mathbf{x}_c[q]$ of a query entity q can be interpreted as the *degree of involvement* of the entity in the event. Regarding step (ii), we solve certain instances of the prize-collecting Steiner tree (PCST) problem (Segev, 1987), where we regard the relatedness score $\mathbf{x}_c[q]$ of a *query entity* q as a "penalty" we have to pay whenever entity q is not included in the computed solution:

Problem: Prize-Collecting Steiner Tree (PCST)

Given Undirected graph $G = (V, E, c, \pi)$ with a non-negative cost function defined on the edges, $c : E \to \mathbb{R}_{\geq 0}$, and a non-negative penalty function defined on the vertices: $\pi : V \to \mathbb{R}_{\geq 0}$.

Goal Identify a subtree $T = (V_T, E_T)$ of G, minimizing the sum of costs of the included edges and the penalties of the vertices not included:

$$T := \operatorname*{arg\,min}_{T \in \mathcal{T}(G)} c(E_T) + \pi(V \setminus V_T). \tag{6.12}$$

where $\mathcal{T}(G)$ denote the set of subtrees of G.

Here, we are interested in the rooted variant, where a designated vertex – in our case the relationship center – has to be spanned by the resulting tree. As penalties, w. l. o. g. we assign – as stated above – to each query entity $q \in Q_1$ the value $\mathbf{x}_c[q]$, and zero penalty to the other (non-query-)vertices in the graph. The PCST algorithm trades off the cost of the edges included in the output tree against the sum of penalties for vertices that are not included in the tree. Thus, the number of query entities that are connected to the relationship center (and thus the size of the resulting tree) is governed by the relationship between edge weights and vertex penalties. In order to include as many query entities as possible while satisfying the size constraint, we introduce a scale factor $\lambda \in [0,1]$ in order to directly control the tradeoff between solution size and number of included query entities. As a result, given one set of query entities – w. l. o. g. the set Q_1 – we try to identify (approximate) a tree containing the relationship center and satisfying

$$T = (V_T, E_T) := \underset{T \in \mathcal{T}(G), c \in V_T}{\arg\min} \; (1-\lambda) c(E_T) + \lambda \left(\sum_{v \in Q_1 \setminus V_T} \mathbf{x}_c[v] \right), \tag{6.13}$$

The last remaining issue then is to identify an appropriate value $\lambda \in [0,1]$ such that the budget on the solution size is satisfied. We use a binary search over the parameter range, as shown in the complete Algorithm 10. The value d used in the termination criterion of the binary search in line 7 can be regarded as a discretization of the interval for λ, trading off speed for solution quality.

Algorithm 10: ConnectQueryEntities-PCST($G, v_c, Q, b, \mathbf{x}_c$)

```
Data: Graph $G = (V, E)$, relationship center $v_c \in V$, query entity set $Q \subseteq V$,
      budget $b$, relatedness scores $\mathbf{x}_c$ w. r. t. center $v_c$
1  begin
       ▷ Are some query entities connected directly to the relationship center?
2      $Q' \leftarrow \{q \in Q \mid (v_c, q) \in E\}$
3      if $Q' \neq \emptyset$ then
4          return $(\{v_c\} \cup Q', \{(v_c, q) \mid q \in Q'\})$
5      $(\lambda_1, \lambda_2) \leftarrow (0, 1)$
6      $(V_T, E_T) \leftarrow (\emptyset, \emptyset)$
7      while $\lambda_2 - \lambda_1 > 1/d$ do
8          $\lambda \leftarrow \lambda_1 + \frac{\lambda_2 - \lambda_1}{2}$
9          $(V_T, E_T) \leftarrow$ RootedPCST$(G, v_c, \mathbf{x}_c, \lambda)$    ▷ compute solution for PCST rooted at $v_c$
10
11         if $|V_T| > b$ then
12             $(\lambda_1, \lambda_2) \leftarrow (\lambda_1, \lambda)$
13         else
14             $(\lambda_1, \lambda_2) \leftarrow (\lambda, \lambda_2)$
15     return $(V_T, E_T)$
```

The procedure RootedPCST solves the rooted variant of the Prize-Collecting Steiner Tree problem, e. g. using the algorithm by Goemans and Williamson (1992), over the input graph G with scaled edge costs, as shown in Equation 6.13. As discussed previously, the original algorithm incurs running time quadratic in the number of vertices with non-zero penalties, i. e. the number of entities in a query set. Faster variants of the algorithm have been proposed by Cole et al. (2001) and, very recently, by Hegde et al. (2014). Both variants solve the unrooted version of the problem and thus require to add a very high penalty to the desired root vertex, the relationship center c. A nice property of above formulation is its ability to deal with skewness in the involvement scores. Whereas in CePS, query entities are connected to the center in descending order of their score, this might lead to suboptimal solutions if the query entity scores are rather uniform. The PCST solution takes the skewness of involvement scores into account in a natural way.

6.6 Relatedness Cores

In the previous sections we have discussed the identification of relationship centers and the ensuing connection to the query entities. Now, we discuss how the relationship centers can be expanded further to provide better relationship explanations.

To this end, we use the following three-phase heuristic algorithm to construct a coherent, yet compact subgraph around a relationship center c.

(i) **Phase 1: Key Entity Discovery.** We compile a set of entities from the neighborhood of c, based on the relationship strength with c, e. g. as measured by the underlying entity-entity relatedness measure (see Section 6.4). For example, when starting with the event *2013 mass surveillance disclosures* as center entity c, we would like to add entities like *Edward Snowden*, *Glenn Greenwald*, etc.

(ii) **Phase 2: Center Context Generation.** The set of entities compiled in Phase 1 is further extended with entities that fill in the context necessary for explanation. Entities that should added in this step typically have a broader scope and less specific relatedness, in the sense that they are also involved in many other events, e. g. *United States*, *The Guardian*, etc.

(iii) **Phase 3: Query Context Generation.** In addition, we finally add entities that are highly related both to the relationship center as well as to a part of the query entities. Continuing above example, if we assume that *New Zealand* is a query entity, an appropriate query context entity is given by *Five Eyes*, the alliance of intelligence agencies involving the United States, United Kingdom, Australia, Canada and New Zealand, since this entity is highly related both to the central entity as well as to the query entity *New Zealand*.

6.6.1 Key Entity Discovery

Starting with the initial set $V_c = \{c\}$ comprising just the center vertex, the algorithm works by iteratively adding the adjacent entity that is most related to c, based on the entity-entity similarity score measure introduced in Section 6.4, or on the degree-of-involvement scores, $\mathbf{x}_c$. This procedure is repeated until the set V_c is sufficiently large, say $\lceil \gamma B'/2 \rceil$ vertices, where B' is the bound on the size of a relatedness core (the user-specified size constraint per core minus the number of vertices used to establish connections to the query sets) and the parameter $\gamma \in [0, 1]$ controls the fraction of key and context entities versus query context entities in a core.

6.6.2 Center Context Entities

Given the current set of key entities, V_c, the goal of the second phase is the generation of additional context by adding generally important entities that are related to many of the key entities in V_c but are not necessarily specific to c. Our algorithm to this end computes a dense subgraph around V_c, using the following greedy procedure: First, we mark each vertex $v \in V_c$ currently contained in the solution after key entity discovery as a *terminal* that must remain in the resulting solution. Second, we add all entities adjacent to the center $c \in V_c$ to an initial graph structure $G_c = (C_c, E_c)$, with

$$C_c = V_c \cup \{v \in V \mid (c, v) \in E\}, \tag{6.14}$$

$$\text{and} \quad E_c = \{(v, w) \in E \mid v, w \in C_c\}. \tag{6.15}$$

Based on the idea of computing dense subgraphs by iterative removal of low-weight vertices (Charikar, 2000, Sozio and Gionis, 2010), we expand the relatedness core as follows. We identify the non-terminal vertex $\hat{v} \in C_c$ with the smallest weighted degree with respect to the key entities:

$$\hat{v} = \operatorname*{arg\,min}_{v \in C_c \setminus V_c} \sum_{w \in V_c : (v,w) \in E_c} \mathrm{sim}(v, w) \cdot \mathbf{x}_c[w]. \tag{6.16}$$

The vertex $\hat{v}$ is deleted from the graph G_c, and the procedure is repeated until the size constraint of $\gamma B'$ vertices is satisfied. The resulting subgraph thus consists of $\lceil \frac{\gamma B'}{2} \rceil$ key and $\lfloor \frac{\gamma B'}{2} \rfloor$ context entities.

6.6.3 Query Context Entities

Given the expanded relatedness core, the goal of the third phase is the addition of query context entities should give further insight into the relationship of the center entity to the query vertices by highlighting certain aspects of the relationship center that explain the involvement of the query entities. This is especially important for the frequently encountered case where some query entities are already directly connected to the relationship center, and no indirect connections have been added. For this purpose, we add the best $B' - \lceil \gamma B' \rceil$ vertices according to the score $\mathbf{x}_{Q_1}[v]\mathbf{x}_{Q_2}[v]\mathbf{x}_c[v] \cdot \mathrm{pr}(v)$, i. e. additional, prominent vertices that are highly related both to the query entities as well as to the relationship center.

The complete Espresso algorithm is outlined in Algorithm 11. In the algorithm we refer to the procedure that connects the center with the query entities as ConnectQueryEntities, which can be the ConnectQueryEntities-PCST approach discussed previously, or another method.

6.6.4 Handling very large query sets

In some cases, the query sets can be very large. In these cases, we can potentially improve query context entity discovery by employing the following modification. After the relatedness center scores $RC(v), v \in V$ have been computed, and the best center events $c_1, c_2, \dots$ have been selected, we compute the relatedness score of the vertices in the graph with respect to relationship center c_i (corresponding to vector $\mathbf{x}_{c_i}$ discussed in the previous section). As outlined above, these scores indicate the degree of involvement of the query entities in the relationship center. Once these scores are computed, we can recompute the relationship center scores $\mathbf{x}_{Q_1}, \mathbf{x}_{Q_2}$ by biasing the required random walks with restarts such that, w. l. o. g. we have for Q_1 the starting probability for $q \in Q_1$ corresponding to $\mathbf{x}_{c_i}[q]$, i. e. the degree of involvement of q in the relationship center. We then rerun the computation of relatedness scores in order to bias the scores towards the query entities that exhibit a high relevance to the center. This approach can help remove the noise due to a large number of query entities affecting the relatedness scores, leading to a potentially better selection of query context entities that are relevant to both the center as well as the related query entities.

6.6.5 Considering Type Constraints

Recall from Section 6.3 that our framework can be enhanced with additional constraints regarding vertex and edge labels. In the following, we discuss how we handle covering constraints on type labels for entity vertices. For a broad domain of interest, such as sports, we assume a set of types that should be covered by the entities in a relatedness core. For example, for a center entity of type *teamsportsMatch*, we want to ensure that we capture related entities of types *Team*, *Stadium*, *Tournament* (or *League*), etc. These types can also be determined automatically by mutual information with respect to the center entity type, potentially in a hierarchical manner.

For a given set of entity types and the constraint that a core must contain entities that together cover all of these types, we extend the Espresso algorithm as follows.

In the first phase, we assign to every edge in the graph a weight corresponding to the reciprocal similarity between the corresponding entities, turning similarity into a distance measure. Then, starting with the center vertex c, we compute a tree containing c and at least one vertex for every required type t, such that the sum of the weights of the included edges is minimized. This problem is known as *Group Steiner Tree* (Reich and Widmayer, 1990), which generalizes the classical Steiner tree problem and is thus NP-hard. However, there are quite a few polynomial-time algorithms with good approximation ratios (Garg et al., 2000, Helvig et al., 2001). In our implementation of Espresso we include a modified version of the algorithm

Algorithm 11: Espresso$(G, Q_1, Q_2, B, \mathrm{pr}, k, \gamma)$

```
Input: graph $G = (V, E)$, query sets $Q_1, Q_2 \subseteq V$, size constraint $B \geq 1$ per core, entity
       prior $\mathrm{pr} : V \to \mathbb{R}_{\geq 0}$, $k$ number of relationship centers to use, $\gamma \in [0, 1]$ fraction
       of key vs. context entities
Result: relatedness cores explaining the relationship between $Q_1$ and $Q_2$
begin
    ▷ Compute relationship center scores according to Equation (6.11)
    $\mathbf{x}_{Q_1} \leftarrow \mathsf{RWR}(G, Q_1)$
    $\mathbf{x}_{Q_2} \leftarrow \mathsf{RWR}(G, Q_2)$
    $(V_{rc}, E_{rc}) \leftarrow (\emptyset, \emptyset)$
    while $|V_c| < kB$ do
        ▷ identify relationship center (restricted to events)
        $c \leftarrow \arg\max_{v \in V, v \notin V_{rc}} \mathbf{x}_{Q_1}[v] \cdot \mathbf{x}_{Q_2}[v] \cdot \mathrm{pr}(v)$
        $V_c \leftarrow \{c\}$
        ▷ compute relatedness of vertices to the relationship center
        $\mathbf{x}_c \leftarrow \mathsf{RWR}(G, \{c\})$
        ▷ find a connection to the query entities, budget for connection if half of the core budget
        $(V', E') \leftarrow \mathsf{ConnectQueryEntities}(G, c, Q_1, (B-1)/4)$
        $(V'', E'') \leftarrow \mathsf{ConnectQueryEntities}(G, c, Q_2, (B-1)/4)$
        ▷ add connections to solution
        $V_c \leftarrow V_c \cup V' \cup V''$
        ▷ size of partially constructed core
        $B' \leftarrow B - |V_c|$
        ▷ add key entities
        $i \leftarrow 1$
        $V_{\mathrm{key}} \leftarrow \emptyset$
        while $i < \lceil \gamma B'/2 \rceil$ do
            $v^* = \arg\max_{v \in V, (c,v) \in E} \mathbf{x}_c[v] \cdot \mathrm{pr}(v)$
            $V_{\mathrm{key}} \leftarrow V_{\mathrm{key}} \cup \{v^*\}$
            $i \leftarrow i + 1$
        $V_c \leftarrow V_c \cup V_{\mathrm{key}}$
        ▷ add context entities
        $C_c \leftarrow \{v \in V \setminus V_c \mid (c, v) \in E\}$
        while $|C_c| > \gamma B'$ do
            $\hat{v} \leftarrow \arg\min_{v \in C_c \setminus V_{\mathrm{key}}} \sum_{w \in C_c : (v,w) \in E} \mathrm{sim}(v, w) \cdot \mathbf{x}_c[w]$
            $C_c \leftarrow C_c \setminus \{\hat{v}\}$
        $V_c \leftarrow V_c \cup C_c$
        ▷ add query context
        while $|V_c| < B$ do
            $v^* = \arg\max_{v \in V, (c,v) \in E} \mathbf{x}_{Q_1}[v] \cdot \mathbf{x}_{Q_2}[v] \cdot \mathbf{x}_c[v] \cdot \mathrm{pr}(v)$
            $V_c \leftarrow V_c \cup \{v^*\}$
        $V_{rc} \leftarrow V_{rc} \cup V_c$
    return $(V_{rc}, (u, v) \in E \mid u, v \in V_{rc})$
```

by Reich and Widmayer (1990), based on expansion of the graph around the center c until all requirements are met, followed by a cleanup phase to prune redundant vertices.

6.7 Integration of Temporal Information

In our model so far, entity-entity relatedness was time-invariant. Here we extend our model by considering the temporal dimension as an additional asset for ensuring the coherence of relatedness cores. The entities in a core are often centered around a key event that connects the two input sets (e. g., the *2013 mass surveillance disclosures*, connecting US politicians and European politicians). We want to ensure that the core is *temporally coherent* in the sense that many participating entities are relevant as of the time of the key event.

With each vertex $v \in V$ in the knowledge graph, we associate an *activity function* that captures the importance of (or public interest in) an entity as a function of time:

$$\alpha_v : T \rightarrow \mathbb{R}$$

One way of estimating the values of this function is to analyze the edit history of Wikipedia: the more edits take place for an article in a certain time interval, the higher the value of the activity function. Other kinds of estimators may tap into longitudinal corpora that spans decades or centuries such as news archives or books. Our implementation is based on Wikipedia page view statistics. This entails that the estimated activity function is a discrete time-series, but we use the notation of continuous functions for convenience.

To make different entities comparable, we normalize the activity function of an entity as follows:

Definition 6.6 (Normalized Activity Function). *Let $v \in V$ denote an entity with activity function $\alpha_v : T \rightarrow \mathbb{R}$. The* normalized activity function *of v is defined as*

$$A_v(t) = \frac{\alpha_v(t) - \mu_{\alpha_v}}{\sigma_{\alpha_v}}, \tag{6.17}$$

with $\mu_{\alpha_v} = \mathbb{E}[\alpha_v]$ and $\sigma_{\alpha_v} = \sqrt{\mathbb{E}\left[(\alpha_v - \mu_{\alpha_v})^2\right]}$.

Thus A_v captures, for every time point, the number of standard deviations from the mean activity of v. A similar technique is proposed by Das Sarma et al. (2011) based on the time-dependent connection discovery for co-buzzing entities.

To assess whether two entities are *temporally coherent*, we compare their activity functions. It turns out that many entities exhibit very pronounced peaks of activity at certain points. These peaks are highly characteristic for an entity. Therefore, we specifically devise a form of temporal peak coherence for comparing entities.

Definition 6.7 (Temporal Peak Coherence). *Let $x, y \in V$ denote two entities and $T = [t_s, t_e]$ a time interval of interest. The T*-peak coherence *of x and y is given by*

$$\operatorname{tpc}(x, y, T) = \int_{t_s}^{t_e} \max\left(\min\left(A_x(t), A_y(t)\right) - \theta, 0\right)\, dt \tag{6.18}$$

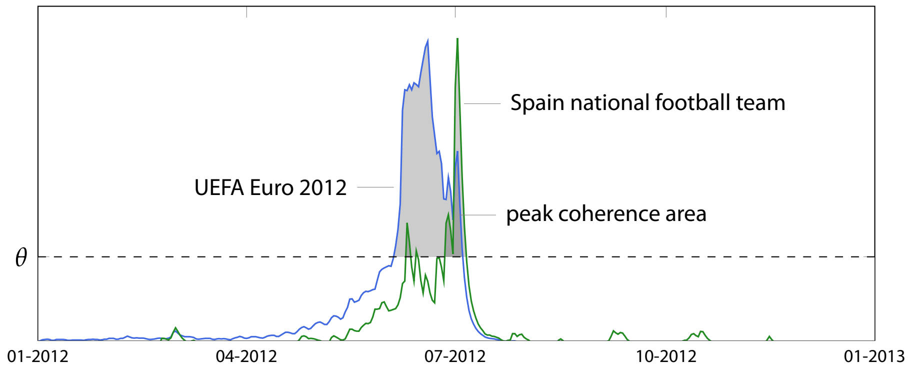


Figure 6.2: Peak Coherence (example)

where θ is a thresholding parameter to avoid over-interpreting low and noisy values.

In this definition, the time interval T can be set in various ways: the overall time-span covered by the knowledge graph, the overlap of the lifespans of the two entities (relevant for people or organizations), or the temporal focus specified by the user in the query that determines the two input sets Q_1 and Q_2.

EXAMPLE. As an example, for the entity *UEFA Euro 2012*, we want to identify entities that exhibit high activity during the tournament, as opposed to famous ones who missed the tournament or did not advance into later stages. High scores will be assigned to successful teams (e. g. *Spain national football team*), notable players (e. g. *Andrés Iniesta*), and important locations (*Olimpiyskiy National Sports Complex*).

Figure 6.2 provides an example for the concepts introduced in this section.

Temporal Peak Coherence for Relatedness Cores We harness the notion of temporal peak coherence in the algorithm for expanding relationship centers into relatedness cores, see Algorithm 11 in Section 6.6. Specifically, when computing key entities surrounding the relationship center c (line 15), we can enforce that only entities v are chosen with temporal coherence $\text{tpc}_T(c, v)$ above a (tunable) threshold τ.

6.8 RELATED WORK

EXPLAINING RELATIONSHIPS BETWEEN ENTITIES

Knowledge discovery in large graphs has been studied along several dimensions. Most relevant to this work is the extraction of subgraphs that explain the relationship between two or more input entities. Faloutsos et al. (2004) proposed the notion of *connection subgraphs*. Given an edge-weighted graph, a pair of query vertices, (s, t), and a budget b, the goal is to extract a connected subgraph containing s and

t and at most b other vertices with a maximum value of a specified goodness function.

Tong and Faloutsos (2006) later generalized this model to the case of a query vertex set Q with $|Q| \geq 2$. In this approach, coined *centerpiece subgraphs* (CePS), the vertices in the graph are assigned a score based on random walks with restart from the query vertices. Ramakrishnan et al. (2005) proposed a connection subgraph discovery algorithm over RDF graphs, based on an edge-weighting scheme taking into account the edge-semantics of the respective RDF schema. Kasneci et al. (2009a) addressed the extraction of *informative subgraphs* with respect to a set of query vertices in the context of entity-relationship graphs. This approach is based on first computing a Steiner tree and then expanding it within a given budget. Cheng et al. (2009) partitioned graphs into communities that define the context of a vertex. Then, a subgraph is computed that connects all query vertices, aiming for strong connections at the intra-community and the inter-community levels. The methods outlined above have been used for improved graph visualization (Chau et al., 2008) and interactive graph mining (Rodrigues et al., 2006). Recently, Akoglu et al. (2013) considered a similar problem, where, given a set of vertices, simple pathways connecting these are found, which are used to partition the input vertices into clusters of related vertices.

Fang et al. (2011) considered entity-relationship graphs in order to explain the relationship between *pairs of individual entities* that co-occur in queries. The algorithm first enumerates subgraph patterns by combining paths between the two query entities, and then ranks patterns based on interestingness measure derived from pattern instances.

Keyword Search over Graphs

Another area of related work is keyword search over (semi-)structured and relational data. In this setting, each vertex of a graph (e. g. derived from foreign-key relationships in a database) is associated with a textual description. The result of a query $Q = \{t_1, t_2, \ldots\}$, consisting of several terms (keywords) t_i, is a cost-minimal tree T (e. g., a Steiner tree based on edge weights) connecting a set of vertices such that for every term t in the query at least one of the vertices in T is a match for term t. This field has been studied extensively (Agrawal et al., 2002, Bhalotia et al., 2002, He et al., 2007, Hristidis and Papakonstantinou, 2002, Hristidis et al., 2003, Kacholia et al., 2005, Kargar and An, 2011), most prominent results being the methods/systems DISCOVER (Hristidis and Papakonstantinou, 2002, Hristidis et al., 2003), DBXplorer (Agrawal et al., 2002), BANKS (Bhalotia et al., 2002, Kacholia et al., 2005), and BLINKS (He et al., 2007). The combination of keyword search over graphs and relationship analysis has been addressed by Kasneci et al. (2009b). Coffman and Weaver (2013) give an overview and experimental comparisons for this research area.

Dense Subgraph Mining

Here the goal is to identify a set of vertices $S \subseteq V$ maximizing a measure of density. A widely used notion of density is the *average-degree*, given by $2|E(S)|/|S|$, where $E(S)$ the set of edges contained in the spanning subgraph of S. In the unconstrained case, computing the densest subgraph is polynomially solvable using a max-flow technique (Goldberg, 1984). With cardinality constraints, the problem becomes NP-hard, though. Lee et al. (2010) gives an overview over this area.

A greedy approximation algorithm computing the densest subgraph with a given number of vertices was proposed by Asahiro et al. (2000). Charikar (2000) developed an $(1/2)$-approximation algorithm. Andersen and Chellapilla (2009) studied further variants of the problem. Sozio and Gionis (2010) addressed the *community-search* problem with the goal of extracting dense components including a set of query vertices. Bahmani et al. (2012) studied the dense subgraph problem in the streaming as well as in the MapReduce model. Tsourakakis et al. (2013) investigate an alternative notion of density based on subgraph diameter.

Event Identification

In some sense, the extraction of relatedness cores resembles event detection over graph data. In this direction, Das Sarma et al. (2011) studied the discovery of events on the time axis (e. g., over query logs or news streams), but did not consider relationships between entities. Instead, the major asset for detecting events is the intensity of entity co-occurrence during certain time windows. In contrast to Espresso, this approach is not driven by given user query but aims to find all events in a real-time manner.

With a similar focus on real-time discovery, Angel et al. (2012) combined dense subgraph mining with a temporal analysis. Underlying is a graph of entities where edge weights are dynamically updated based on co-occurrence patterns in news or social-media streams. In this setting, events (or "stories") correspond to dense subgraphs – continuously maintained as content items – that refer to multiple entities. More remotely related, Hossain et al. (2012) studied the problem of storytelling in entity-relationship graphs, where directed chains between target entities are extracted.

6.9 Experimental Evaluation

This section presents experimental comparisons of Espresso with different baseline methods at two levels: (i) computing relationship centers, and (ii) end-to-end evaluations comparing the full explanation for the relationship between two entity sets. We discuss both the informativeness of the outputs, as judged in user studies, and the efficiency of the computation, as determined in run-time measurements.

6.9.1 Setup

All experiments were conducted on a Dell PowerEdge M610 server, equipped with two Intel Xeon E5530 CPUs, 48 GB of main memory, and running Debian Linux (kernel 3.10.40.1.amd64-smp) as operating system. All algorithms have been implemented in C++11 (GCC 4.7.2).

Data As previously described in Section 6.2, for the evaluation of our approaches we have extracted a large entity-relationship graph from the intersection of the YAGO2 and FreeBase knowledge bases, comprising roughly 3.7 million distinct entities, corresponding to Wikipedia articles. Using the links between Wikipedia pages, we have extracted an undirected graph by mapping each link from page u to page v to an undirected edge, (u, v) for the corresponding entities. The resulting graph structure comprises almost 60 million edges. Wherever YAGO2 knows specific relationship labels between two entities, these are taken as edge labels for the knowledge graph, in addition to the generic label *relatedTo* for all edges. The complete dataset is available for download for further studies[6].

Queries We have manually created a set of 15 queries, each consisting of two sets of entities, Q_1, Q_2. There are 5 queries from each of the three categories *Politicians, Organizations*, and *Countries*. An overview of the structure and contents of all queries is given in Table 6.1.

Evaluation Metrics As all methods for identification of relationship centers yield ranked results, we evaluate the informativeness of the competitors by two metrics from information retrieval: *precision* and *normalized discounted cumulative gain (NDCG)*. For this purpose, we have conducted a user study, in which a total of six judges assessed the relevance of the top-5 outputs (best entities of type *event* (manually identified) as relationship centers) for each of the 15 different test queries. The task of each judge was to evaluate the relevance of the output event with respect to the two input sets Q_1, Q_2, using a graded relevance scheme with values 0 (not relevant) to 3 (very relevant). The grades of the six judges were averaged for each test case. Each item was evaluated by all six evaluators. To compute precision, we mapped average grades from $[0, 1.5)$ onto 0 (irrelevant) and from $[1.5, 3]$ to 1 (relevant). The precision at rank k then is the fraction of relevant items among the first k results in the ranked list $R = (r_1, r_2, \ldots, r_k)$:

$$\text{Prec@k}(R) = \frac{\left|\{r_i \in R \mid i \leq k, \text{rel}(r_i) \geq 1.5\}\right|}{k}. \tag{6.19}$$

The evaluation by NDCG uses the average grades assigned by the judges on the full scale from 0 to 3. To compute the normalization for NDCG, we used a pooling approach: for each query, the results returned by all considered algorithms and settings are combined into a single list and ranked with respect to the relevance grades assigned by the human annotators. This list is denoted by C. We compare

[6] `http://espresso.mpi-inf.mpg.de/`

Query	Area	Q_1	Q_2	$\lvert Q_1\rvert / \lvert Q_2\rvert$
Q1	Politics	Heads of State (North America) *Barack Obama, Stephen Harper*	Heads of State (5 largest EU member states) *Angela Merkel, François Hollande, David Cameron, Giorgio Napolitano, Mariano Rajoy*	2/5
Q2	Politics	Heads of State (North America) *Barack Obama, Stephen Harper*	Heads of State (Asian G20 member states) *Xi Jinping, Li Keqiang, Pranab Mukherjee, Narendra Modi, . . .*	2/8
Q3	Politics	Heads of State (Pakistan) *Mamnoon Hussain, Nawaz Sharif, Asif Ali Zardari, . . .*	Heads of State (India) *Pranab Mukherjee, Narendra Modi, Prathiba Patil, . . .*	9/5
Q4	Politics	Russian Prime Ministers *Vladimir Putin, Dmitriy Medvedev, Viktor Zubkov*	Presidents of the United States *Barack Obama, George W. Bush, Bill Clinton*	6/3
Q5	Politics	Heads of State (Japan) *Naoto Kan, Yoshihiko Noda, Shinzō Abe, . . .*	Heads of State (China) *Hu Jintao, Xi Jinping, Li Keqiang, Wen Jiabao*	5/4
Q6	Organizations	Terrorist Organizations *Kurdistan Workers' Party, Liberation Tigers of Tamil Eelam, al Qaeda, ETA, . . .*	North American and Western European Countries *United States, Canada, Portugal, France, Spain, Germany, . . .*	11/18
Q7	Organizations	Peace Organizations *Amnesty International, Grameen Bank, Intergovernmental Panel on Climate Change*	Countries in Asia *China, Japan, India, South Korea, Indonesia, . . .*	14/26
Q8	Organizations	Environmentalism Organizations *Earth First, Friends of the Earth, Green Cross, Greenpeace, . . .*	Oil Companies *BP, Chevron Corporation, Exxon Mobil, . . .*	7/ 6
Q9	Organizations	Health and Human Rights Organizations *World Health Organization, UNICEF, Oxfam, . . .*	Food Industry *Nestlé, Groupe Danone, PepsiCo, Arla Foods, . . .*	7/16
Q10	Organizations	Privacy Advocate Organizations *Privacy International, Electronic Privacy Information Center, American Civil Liberties Union, . . .*	Internet Companies *Apple Inc., Google, Yahoo!, Microsoft Corporation, . . .*	8/6
Q11	Countries	Eurozone Countries *Austria, Belgium, Cyprus, . . .*	Investment Banks *JPMorgan Chase, Merrill Lynch, Goldman Sachs, . . .*	18/10
Q12	Countries	China	United Kingdom	1/1
Q13	Countries	Russia	Countries in Asia *China, Japan, India, South Korea, Indonesia, . . .*	1/26
Q14	Countries	Western European Countries *Portugal, France, Spain, Germany, Netherlands, . . .*	North African Countries *Algeria, Egypt, Libya, Morocco, . . .*	15/7
Q15	Countries	United States	Countries from South and Middle America *Argentina, Brazil, Mexico, Colombia,*	1/23

Table 6.1: Overview of the 15 test queries

the top-k result lists from the competing algorithms against the respective ideal ranking of the combined results. Here, ideal ranking means that results are sorted in descending order of the judges' average grades. For ranking $R = (r_1, r_2, \ldots, r_k)$ we compute

$$\mathsf{DCG}_k(R) = \sum_{i=1}^{k} \frac{2^{\mathrm{rel}_i} - 1}{\log_2(i+1)}, \tag{6.20}$$

where rel_i denotes the average grade of the i-th item in R, obtained in the user study. We finally normalize by the ideal ranking C:

$$\mathsf{nDCG@k}(R) = \frac{\mathsf{DCG}_k(R)}{\mathsf{DCG}_k(C)} \tag{6.21}$$

6.9.2 Results: Extracting Relationship Centers

Baseline: CePS

We compare our method for extracting relationship centers against the extension of the centerpiece sub-graph algorithm by Tong and Faloutsos (2006) for sets, considering both the *CePS-AND* (cf. Equation 6.9) as well as the *CePS2-OR* (cf. Equation 6.10) variant. Recall that this baseline performs random walks with restarts from *all* query nodes $q_i \in Q$, given a single node set. These approaches are abbreviated as CePS-AND and CePS2-OR, respectively.

Informativeness

We evaluate the relevance of computed relationship centers by Espresso and CePS in the following scenarios. First, we use two different entity-entity relatedness measures, MW and KORE. Second, we use the plain top-5 ranking (based only on random walk scores) and the top-5 after reranking. In this setting, the reranking procedure multiplies the relatedness score with the entity prior (as described in Equation 6.11), which, in this experiment, corresponds to the square root of the relative amount of Wikipedia pageviews for the corresponding article over the time period from January 1, 2013 to July 31, 2014.

A total number of six judges responded to the study, which comprises 249 distinct (question, event) pairs. The results of comparing the relationship centers of our set-based random walk algorithm described in Section 6.4 – denoted by RC – against those by the extension of CePS for query sets are shown in Table 6.2, for the top-5 results.

The explicit distinction between the two input sets of entities, Q_1, Q_2 benefits Espresso and CePS2-OR, whereas CePS-AND requires central vertices relevant to all query entities. For 6 of the 12 settings (area, relatedness measure, reranking), CePS2-OR is slightly better than Espresso (RC), for 3 queries both achieve the same precision, and Espresso even produces the best result for 3 queries. Over all queries, Espresso achieves average precision of 0.75, whereas CePS2-OR and CePS-AND achieve average precision of 0.77 and 0.58, respectively. Thus, Espresso manages to attain 97% of the precision of CePS2-OR, while only requiring two

RWRs from the query sets.
The results obtained by using Milne-Witten inlink overlap (MW) as relatedness compare favorably with the results obtained from employing the KORE measure. Reranking based on entity popularity improves the result quality significantly for the MW measure. Thus, the combination of MW and popularity-based reranking is used as the default strategy for ESPRESSO and CEPS.

The NDCG results for the two algorithms, *RC* and the CEPS variants are shown in Table 6.3. Overall, ESPRESSO achieved average NDCG@5 of 0.67, while CePS2-OR and CePS-AND achieved scores of 0.68 and 0.55, respectively. ESPRESSO thus retains 98% of the score.

Table 6.4 shows sample results based on the RC scores for three queries, together with the averaged relevance grades obtained from the judges in the user study.

Run-Time Efficiency

From an asymptotic complexity perspective, the analysis is straightforward:

- The CEPS variants require a random walk process from each of the $|Q_1| + |Q_2|$ query vertices. Each RWR iteration has a time complexity of $O(|E|)$. In our implementation we use a fixed number of I iterations. Thus, the time complexity for this method is $\Theta\big((|Q_1| + |Q_2|)I|E|\big)$.

- For our random walk approach *RC* described in Section 6.4, we conduct one random walk with restart from each query set. With a fixed number of I iterations, this procedure results in a time complexity of $\Theta(2I|E|)$.

It is evident that the time for computing RC scores is independent of the input set sizes and only depends on the size of the overall graph, whereas the CEPS method has linear dependence on the size of the input entity set(s).

For the 15 test queries, the running-times of ESPRESSO and CEPS are shown in Table 6.5. In our implementation we use $I = 25$ iterations for the random walk and a restart probability of 0.5. As expected, the time required to compute RC scores remains constant ($\sim$ 25 seconds for MW, $\sim$ 58 seconds for KORE) across all queries, while the time required to compute CEPS scores varies between 25 (MW) and 56 (KORE) seconds for Q12 and 505 (MW) as well as 1,141 (KORE) seconds for Q7. The algorithms are significantly faster when the MW measure is used, simply because for this measure many of the edges receive zero weight (for the case of no inlink overlap). These edges are not included in the graph, resulting in faster running times.

Algorithm	Relatedness	Rerank	Q1	Q2	Q3	Q4	Q5	Avg
RC	KORE	✓	**1.00**	0.40	0.60	**1.00**	0.60	0.72
RC	KORE	✗	**1.00**	0.80	**0.80**	**1.00**	**1.00**	**0.92**
RC	MW	✓	**1.00**	**1.00**	**0.80**	0.80	**1.00**	**0.92**
RC	MW	✗	**1.00**	**1.00**	0.40	0.60	**1.00**	0.80
CePS SETS-OR	KORE	✓	**1.00**	0.40	0.60	**1.00**	0.80	0.76
CePS SETS-OR	KORE	✗	**1.00**	0.80	**0.80**	**1.00**	**1.00**	**0.92**
CePS SETS-OR	MW	✓	**1.00**	**1.00**	**0.80**	0.80	**1.00**	**0.92**
CePS SETS-OR	MW	✗	**1.00**	**1.00**	0.60	0.60	**1.00**	0.84
CePS SETS-AND	KORE	✓	0.80	0.80	0.40	0.80	0.60	0.68
CePS SETS-AND	KORE	✗	**1.00**	0.80	0.40	0.80	0.60	0.72
CePS SETS-AND	MW	✓	**1.00**	**1.00**	0.60	0.80	0.60	0.80
CePS SETS-AND	MW	✗	**1.00**	**1.00**	0.60	0.80	0.60	0.80

(a) Q1-5: Politicians

Algorithm	Relatedness	Rerank	Q6	Q7	Q8	Q9	Q10	Avg
RC	KORE	✓	**1.00**	0.60	**1.00**	0.60	0.80	0.80
RC	KORE	✗	**1.00**	0.60	**1.00**	**1.00**	**1.00**	**0.92**
RC	MW	✓	0.80	**1.00**	**1.00**	0.80	**1.00**	**0.92**
RC	MW	✗	0.60	**1.00**	0.80	0.60	**1.00**	0.80
CePS SETS-OR	KORE	✓	**1.00**	0.60	**1.00**	0.80	0.80	0.84
CePS SETS-OR	KORE	✗	0.80	0.80	**1.00**	**1.00**	0.80	0.88
CePS SETS-OR	MW	✓	0.80	**1.00**	**1.00**	0.60	**1.00**	0.88
CePS SETS-OR	MW	✗	0.80	0.80	0.80	0.60	**1.00**	0.80
CePS SETS-AND	KORE	✓	0.40	0.80	**1.00**	0.60	0.80	0.72
CePS SETS-AND	KORE	✗	0.40	0.80	**1.00**	0.60	0.80	0.72
CePS SETS-AND	MW	✓	0.60	0.60	**1.00**	0.60	**1.00**	0.76
CePS SETS-AND	MW	✗	0.60	0.40	**1.00**	0.60	**1.00**	0.72

(b) Q6-10: Organizations

Algorithm	Relatedness	Rerank	Q11	Q12	Q13	Q14	Q15	Avg
RC	KORE	✓	0.60	0.40	**0.40**	0.40	0.40	0.44
RC	KORE	✗	**1.00**	0.40	0.20	0.40	**0.80**	0.56
RC	MW	✓	**1.00**	**1.00**	0.20	0.60	0.60	0.68
RC	MW	✗	0.80	**1.00**	0.00	0.00	**0.80**	0.52
CePS SETS-OR	KORE	✓	0.60	0.40	**0.40**	0.60	0.40	0.48
CePS SETS-OR	KORE	✗	**1.00**	0.40	0.20	0.40	0.60	0.52
CePS SETS-OR	MW	✓	**1.00**	**1.00**	**0.40**	**0.80**	**0.80**	**0.80**
CePS SETS-OR	MW	✗	0.80	**1.00**	**0.40**	0.20	**0.80**	0.64
CePS SETS-AND	KORE	✓	0.20	0.40	**0.40**	0.40	0.00	0.28
CePS SETS-AND	KORE	✗	0.20	0.40	0.20	0.40	0.00	0.24
CePS SETS-AND	MW	✓	0.20	**1.00**	0.00	0.20	0.00	0.28
CePS SETS-AND	MW	✗	0.20	**1.00**	0.00	0.20	0.00	0.28

(c) Q11-15: Countries

Table 6.2: Prec@5 over the set of 15 test queries

Algorithm	Relatedness	Rerank	Q1	Q2	Q3	Q4	Q5	Avg
RC	KORE	✓	0.88	0.31	**0.68**	0.39	0.70	0.59
RC	KORE	✗	0.92	0.79	0.63	0.55	0.79	**0.73**
RC	MW	✓	**0.96**	0.88	0.54	0.49	0.66	0.71
RC	MW	✗	0.95	0.87	0.30	0.52	0.72	0.67
CePS SETS-OR	KORE	✓	0.95	0.40	**0.68**	0.39	**0.80**	0.64
CePS SETS-OR	KORE	✗	0.92	0.73	0.65	0.56	0.79	0.73
CePS SETS-OR	MW	✓	0.95	0.88	0.55	0.51	0.67	0.71
CePS SETS-OR	MW	✗	0.95	**0.89**	0.41	0.52	0.72	0.70
CePS SETS-AND	KORE	✓	0.75	0.70	0.50	0.48	0.74	0.63
CePS SETS-AND	KORE	✗	0.86	0.73	0.50	0.58	0.74	0.68
CePS SETS-AND	MW	✓	0.92	0.82	0.40	**0.76**	0.67	0.71
CePS SETS-AND	MW	✗	0.92	0.82	0.40	0.76	0.66	0.71

(a) Q1-5: Politicians

Algorithm	Relatedness	Rerank	Q6	Q7	Q8	Q9	Q10	Avg
RC	KORE	✓	0.90	0.53	0.91	0.69	0.78	0.76
RC	KORE	✗	0.90	0.66	0.89	**0.91**	0.84	**0.84**
RC	MW	✓	0.83	0.69	0.90	0.83	0.82	0.81
RC	MW	✗	0.64	**0.78**	0.75	0.70	**0.86**	0.74
CePS SETS-OR	KORE	✓	**0.90**	0.53	0.91	0.80	0.76	0.78
CePS SETS-OR	KORE	✗	0.83	0.68	**0.92**	**0.91**	0.77	0.82
CePS SETS-OR	MW	✓	0.83	0.65	0.81	0.71	0.83	0.76
CePS SETS-OR	MW	✗	0.69	0.62	0.75	0.67	0.85	0.71
CePS SETS-AND	KORE	✓	0.53	0.63	0.88	0.54	0.76	0.67
CePS SETS-AND	KORE	✗	0.53	0.64	0.89	0.54	0.76	0.67
CePS SETS-AND	MW	✓	0.62	0.45	0.88	0.59	0.81	0.67
CePS SETS-AND	MW	✗	0.64	0.34	0.88	0.60	0.81	0.65

(b) Q6-10: Organizations

Algorithm	Relatedness	Rerank	Q11	Q12	Q13	Q14	Q15	Avg
RC	KORE	✓	0.36	0.40	0.56	0.53	0.40	0.45
RC	KORE	✗	**0.98**	0.39	0.50	0.40	0.48	0.55
RC	MW	✓	0.96	**0.80**	0.26	0.76	0.57	0.67
RC	MW	✗	0.81	0.79	0.26	0.10	0.63	0.52
CePS SETS-OR	KORE	✓	0.36	0.40	**0.56**	0.59	0.40	0.46
CePS SETS-OR	KORE	✗	0.98	0.39	0.48	0.48	0.42	0.55
CePS SETS-OR	MW	✓	0.96	**0.80**	0.37	**0.86**	**0.68**	**0.73**
CePS SETS-OR	MW	✗	0.83	0.79	0.46	0.20	0.63	0.58
CePS SETS-AND	KORE	✓	0.18	0.40	0.44	0.51	0.11	0.33
CePS SETS-AND	KORE	✗	0.18	0.39	0.33	0.51	0.11	0.30
CePS SETS-AND	MW	✓	0.18	**0.80**	0.11	0.27	0.05	0.28
CePS SETS-AND	MW	✗	0.22	0.79	0.11	0.28	0.05	0.29

(c) Q11-15: Countries

Table 6.3: nDCG@5 over the set of 15 test queries

Rank	Event	Relevance
1	Canada–South Korea Free Trade Agreement	2.66
2	2013 G-20 Saint Petersburg summit	2.42
3	39th G8 summit	2.58
4	40th G7 summit	2.66
5	APEC Indonesia 2013	2.00

(a) Q2: heads of State (North America) and heads of state (Asian G-20 members)

Rank	Event	Relevance
1	Stop Esso campaign	3.00
2	Exxon Valdez oil spill	3.00
3	Global warming conspiracy theory	2.17
4	Global warming controversy	2.66
5	Deepwater Horizon oil spill	3.00

(b) Q8: environmentalism organizations and oil companies

Rank	Event	Relevance
1	World War II	1.33
2	Second Opium War	3.00
3	Panda diplomacy	1.83
4	Boxer Rebellion	2.83
5	Operation Minden	2.50

(c) Q12: China and United Kingdom

Table 6.4: Example results: top-5 extracted relationship centers (MW+reranking)

6.9.3 Results: Explaining Relationships between Entity Sets

We also conducted end-to-end experiments for generating relationship explanations for each of our 15 test queries, comparing Espresso to an extension of the recently proposed Rex approach of Fang et al. (2011) to sets of input entities rather than individual pairs, as well as the CePS algorithm.

Baseline: REX The Rex algorithm in its original form computes relationship explanations in the form of instances of informative graph patterns (edge-labeled templates) that connect two nodes. Due to its restriction to a single pair of query nodes we extend Rex in the following way: Given input sets Q_1, Q_2 of entities, we choose K pairs of query entities $(q_1^i, q_2^i) \in Q_1 \times Q_2, 1 \leq i \leq K$ and aggregate the individual results into a combined result. Specifically, this involves three steps: (1) select query entity pairs, (2) for each (q_1, q_2) pair compute the best p patterns $P_1, \ldots, P_p$ (for pattern ranking we use a combination of the size and monocount measures described in the original paper (Fang et al., 2011), and (3) add the heaviest (in terms of sum of edge weights) i instances of each of the p best patterns to the candidate subgraph $C = (V_c, E_c)$. Subsequently, in order to satisfy the size constraint on the output, while maintaining a coherent relationship explanation, we prune the candidate subgraph as follows. We arrange the vertices in C that do not correspond

Algorithm	Relatedness	Q1	Q2	Q3	Q4	Q5	Avg
RC	KORE	57.61	57.20	56.63	56.69	56.66	56.96
RC	MW	25.38	25.39	25.32	25.35	25.36	25.36
CePS	KORE	200.57	316.16	403.98	172.72	259.22	270.53
CePS	MW	88.38	138.81	176.71	75.78	113.61	118.66

(a) Q1-5: Politicians

Algorithm	Relatedness	Q6	Q7	Q8	Q9	Q10	Avg
RC	KORE	57.03	57.70	57.61	57.54	57.72	57.52
RC	MW	25.41	25.40	25.39	25.33	25.35	25.37
CePS	KORE	833.65	1141.20	371.81	628.13	368.90	668.74
CePS	MW	366.09	504.89	164.11	277.71	164.08	295.38

(b) Q6-10: Organizations

Algorithm	Relatedness	Q11	Q12	Q13	Q14	Q15	Avg
RC	KORE	57.55	57.55	57.82	57.77	57.19	57.58
RC	MW	25.37	25.39	25.41	25.40	25.34	25.38
CePS	KORE	794.49	56.44	770.82	657.75	659.62	587.82
CePS	MW	353.35	25.33	340.93	290.45	290.35	260.08

(c) 11-15: Countries

Table 6.5: Running times [s] over the set of 15 test queries for center computation (Espresso (RC) vs. CePS-AND)

to query entities in a min-heap based on the weighted degree in C. As long as the cardinality constraint is violated, we repeatedly identify a vertex for pruning. For this purpose we select the vertex from the heap with lowest weighted degree that does not disconnect C, in the sense that each inner vertex $v_i \in V_c \setminus (Q_1 \cup Q_2)$ remains connected to at least one vertex in both Q_1 and Q_2. If no such vertex exists, we simply prune the inner vertex v_i with lowest weighted degree, potentially having to prune dangling vertices (i. e. vertices with degree 1 after deletion of v_i) afterwards. This way, we aim to obtain a coherent, dense graph that maintains as much informativeness as possible.

We used the following parameters for the extended Rex algorithm. The pattern size was restricted to 4. For each query we use up to 20 pairs of query entities $(q_1, q_2) \in Q_1 \times Q_2$ to compute pairwise explanations. If it holds $|Q_1||Q_2| > 20$, we randomly select 20 pairs. Our implementation uses bidirectional expansion from the pair of query vertices to generate the path patterns. In order to cope with the combinatorial explosion, we restricted the number of generated paths per pair to 20,000. We have implemented the *PathUnionBasic* algorithm to generate the patterns, as described by Fang et al. (2011). For each pair of query entities we compute the 5 best patterns as measured by size (smaller better) and monocount (larger better). Finally, for each of the patterns, we add the heaviest two instances to the candidate subgraph, followed by the pruning phase to consolidate a small and dense connecting subgraph. The algorithm was implemented in C++ and compiled at the highest available optimization level. As input graph we used the Espresso Knowledge Graph with MW-weighted edges, and add additional directed, labeled edges for YAGO2 facts between the entities.

Baseline: CePS As second baseline, we implemented the entire algorithm for centerpiece subgraph (CePS) extraction, as described by Tong and Faloutsos (2006), and compute results in the *CePS2-OR* setting, i. e. random walks with restarts from all query entities with the aggregation based on Equation 6.10, followed by the *EXTRACT* algorithm to connect the best query entity from either set to the centerpiece vertices (via a so-called key path) in a dynamic programming fashion. Regarding the dynamic programming procedure to extract key paths, in our implementation we will directly connect a query entity to the current center if the latter is already present in the partially built graph and a direct edge exists, rather than extracting a key path. The input graph used is the Espresso Knowledge Graph with MW-weighted edges. Since our previous user study indicated the combination of MW-weights with popularity based reranking as entity prior to be the superior strategy, we incorporate the reranking as follows. After all random walk processes have finished, we multiply the resulting aggregated score for each individual vertex with the entity prior (square-rooted relative popularity). In addition, after the aggregated score has been computed, we also multiply the scores obtained by each individual random walk process with the entity prior. We implemented CePS in C++, compiled using the highest available optimization level.

The cardinality constraint we used in this experiment was 6, for all three algorithms (Rex, CePS, Espresso). For the random walks in CePS and Espresso we use 5 iterations and set the restart probability to 0.5. For Espresso, the parameters we specify are a number of 3 relationship centers to use, fraction of key and context entities vs. query context entities set to $\gamma = 0$. This represents the first stage visualization of the proposed Espresso system, where we rather show more yet compact events, to give a good overview. In a user-facing system, the individual events would then be shown with more key/context and query-context when selected by the user (leading to recomputation with a higher value for γ and cardinality constraint).

For all queries, there was at least one query entity from either query set directly connected to an event. As a result, it was not required to compute the connections to the query entities, rather we only had to connect the event with the appropriate query entities and could add one query context entity to each of the relationship centers. As discussed previously, one approach to ensure informativeness of the central entities is to ensure a certain level of prominence of the considered events. In CePS, where we do not filter by entity types, the prominence of the center entities is ensured by the popularity-based reranking step. In the Espresso algorithm, we discard event entities from consideration with less than 50 daily pageviews on average, and a maximum number of daily pageviews of less than 500, prior to reranking. As a result, out of the 89,321 entities originally marked as events in the Espresso Knowledge Graph, a total number 7,056 events remains as candidates for relationship centers.

Both CePS as well as Espresso include the subgraph induced by the query and center entities, i. e. all edges appearing between any pair of vertices in the computed solution, including edges within the query sets. The labels displayed are the most specific ones associated with each included edge for CEPS and ESPRESSO, and the labels contained in the best patterns for REX. We omit the generic label *relatedTo* in the displayed solutions.

Informativeness The resulting explanations were assessed by human judges. For every query, the output of the three algorithms ESPRESSO, extended REX, as well as CePS2-OR was presented pairwise to the users, who had to choose whether (i) the first graph provides a better explanation, (ii) both graphs are equally good or (iii) the second graph provides a better explanation. For each of the 15 queries the users evaluated all 3 possible pairings (REX vs CePS2-OR, CePS2-OR vs ESPRESSO, ESPRESSO vs REX). We used the same graph layout for all three algorithms. A total number of six judges responded to the study. We show the resulting judgements in Tables 6.6(a)-(c). It becomes clear that the judges clearly favor the graphs computed by ESPRESSO over the two baseline solutions. In 82% of all considered cases, the solution returned by ESPRESSO was favored over the solution returned by REX, while both were rated equally good in 10% of all cases. When compared with CEPS, ESPRESSO returned the better solution in 61% of all cases, and both were evaluated equally good in 23% of all cases. The comparison between CEPS and REX shows that both were evaluated equally good in 22% of the considered cases, while CEPS computed the better solution for 57% of the queries.

Rex better	both equal	CePS2-OR better	total
19 (21%)	20 (22%)	51 (57%)	90

(a) Rex vs. CePS

Rex better	both equal	Espresso better	total
7 (8%)	9 (10%)	74 (82%)	90

(b) Rex vs. Espresso

CePS2-OR better	both equal	Espresso better	total
14 (16%)	21 (23%)	55 (61%)	90

(c) CePS2-OR vs. Espresso

Table 6.6: Results of the end-to-end evaluation of Rex, CePS2-OR, and Espresso

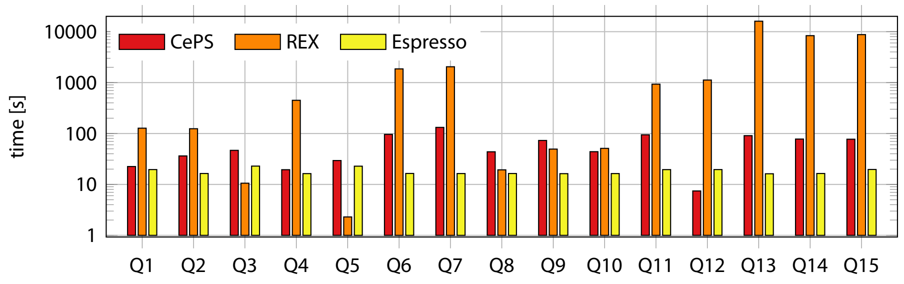


Figure 6.3: End-to-end computation time for CePS, Rex, and Espresso

Efficiency Regarding the efficiency, we compare the end-to-end running times of the considered algorithms in Figure 6.3. Espresso is the fastest of all three algorithms, and exhibits execution times always below 23 seconds over all queries, highlighting its independence from the size of the query sets. In general, the number of random walks executed from the relationship centers differs for the different questions, since sometimes the extracted cores overlap, leaving budget for additional relationship centers. Thus, for 9 of 15 queries the running time of Espresso amounts to $\sim$ 16 seconds, for 4 queries to $\sim$ 19 seconds, and for one query (Q3) to 23 seconds.

On the other hand, the running time of CePS is directly dependent on the size of the query sets, since it executes a random walk from each query entity. The overall running times range from 6 seconds (Q12) to 123 seconds (Q7), while CePS is slower than Espresso for 12 of the 15 queries. The time required to compute the relationship explanation with the extended Rex algorithm ranges from 2 seconds (Q5) to 16,000 seconds (Q13), depending highly on the number of generated paths between the query entities, and the resulting number of path patterns that are subsequently combined in order to identify informative connecting patterns. As a result, Rex is the slowest algorithm for 10 of the 15 considered queries.

6.9.4 Discussion

In this section we summarize the findings from our user studies and efficiency experiments. The Rex algorithm relies on informative patterns, which in turn depend on the available labels annotated with the edges in the knowledge graph. As it turns out, for the case of the Espresso Knowledge Graph, which carries most of the relationship labels present in the YAGO2 knowledge base, this strategy seems ineffective. One reason is the sparsity of labels apart from the generic and overly prominent label *relatedTo*. This label makes up for about 56 million of the 60 million labeled edges. As a result, many of the generated patterns include this generic label, limiting the informativeness of generated patterns. Regarding the computation time, Rex is highly dependent on the local density of the query entities, directly influencing the number of generated paths. It turns out that a moderate pattern size of 4 vertices is already computationally challenging, due to the combinatorial explosion. However, we remark that we implemented the more basic version of the pattern enumeration algorithm described by Fang et al. (2011), so we can expect better running times from an optimized implementation. Finally, the original Rex algorithm is geared towards explaining the relationship between entity *pairs*, rather than sets of entities. In our extended implementation, we compute (a subset of) all possible pairs, resulting in multiple executions of the original pairwise algorithm.

Regarding the comparison with CePS, it is clear that the original algorithm proposed by Tong and Faloutsos (2006) is geared towards computing a good subgraph providing a connection *among some (soft-and query) or all (and query)* query entities rather than between two sets. However, as far as the identification of centerpiece vertices is concerned, we have shown in Equation 6.9 and 6.10 on page 143 that the relatedness measure can be adapted to queries comprising two sets of query entities.

Indeed, the CePS2-OR approach computes slightly better scores when compared to Espresso, albeit at increased running time. Afterwards CePS connects the high-scoring centerpiece vertices with the query sets. As it turns out, for knowledge graphs and the querying scenarios (e. g. of a political analyst researching a story background) envisioned in this work, computing good paths between query and center entities is less important, because usually either a salient connection (event) involving query entities from either set exists, or not, in which case the query entities can be deemed too unrelated. Indeed, for most considered queries, the centerpiece vertices were directly connected to both query sets, so that just the best vertices according to score were added and edges to the query sets included, and only few key paths were extracted by CePS. In this case, popularity-based reranking of results had adversary effect, boosting very generic entities such as countries. This happened because, in contrast to Espresso, there is no notion of entity types in CePS.

We can thus reason, that in the knowledge graph scenarios, it is very important to enforce informativeness of the central entities. In the experimental evaluation of Espresso, this is ensured by focusing on central vertices of type *event*, and extracting *coherent* cores by combining the relatedness center scores w. r. t. to the query entities with relatedness scores w. r. t. the center event, in order to identify good query context entities.

The restriction to entities typed *event* gives an advantage to Espresso, since many non-informative entities are not considered as central vertices (e. g. for the case of countries in the two query sets, we can expect other related/bordering countries to be assigned high scores – however, connections via such vertices are hardly informative and do not give much insight into the relationship). The main insight is thus, that entity type restriction is a crucial step for a good relationship explanation. It should further be mentioned, that for the settings of Espresso with core cardinality $B = 1$, and CePS in the CePs2-SETOR setting employing an event-filtering step similar to Espresso, we would expect very similar results. Espresso can however support further analysis by expanding the identified events into coherent explanations via the addition of key, context, and query context entities.

It remains a challenging problem to automatically identify appropriate entity types, to distinguish between potentially informative and less informative central vertices. For many queries involving real-world entities, the restriction to events works very well. This advantage carries a certain risk however, if good connections are missed because no event exists that involves the query entities (but, rather, subsets of the two query sets are for example members of the same organization, etc.).

One strength of Espresso lies in its flexibility to adapt to a wide range of user specifications, such as the budget to spend for connecting the query sets to the central event (using the PCST algorithm), using temporal coherence to detect key entities, etc.
We show some anecdotal examples for the solutions computed by Espresso in the description of the system demonstration in Appendix A.

6.10 Summary

Entities and their relationships play an increasingly important role as semantic features for numerous kinds of search, recommendation, and analytics tasks – on the Web, for social media, and in enterprises. Explaining relationships between entities and between sets of entities is a key issue within this bigger picture. The Espresso framework presented in this chapter is a novel contribution along these lines. Our experiments, including user studies, demonstrate the practical benefits of Espresso.

Future technical work include the automatic derivation of informative entity types based on the query as well as harnessing our methods for better summarization of online contents, for both individual documents (e. g., articles in news or magazines) and entire corpora (e. g., entire discussion threads, entire forums, or even one day's or one week's entire news). Users are overwhelmed with content; explaining and summarizing content at a cognitively higher level is of great value.

Part III

The Big Picture

7 Summary & Future Directions

Summary of Contributions
In this thesis, we have discussed the problem of analyzing the relationship between entities – represented as vertices in a graph – from two different angles.
First, with a focus on facilitating the interactive application of fundamental analysis tasks, we have proposed:

1. The FERRARI index structure for reachability analysis of large, memory-resident graphs. Reachability queries allow to answer questions of the following form:

   - *Which relationships exist in the data?*
   - *Is there a relationship between a certain pair of entities?*

   In this work, we treat entities as vertices in a graph. Thus, the former question corresponds to the graph-theoretic concept of (efficiently) computing or approximating the transitive closure of the graph and the latter to probing the graph for the existence of a path between the specified vertices. The proposed FERRARI index structure can be regarded as an adaptive compression of the transitive closure, representing for each vertex in the graph the set of reachable vertices by a combination of exact and approximate identifier ranges. While this compression approach trades off scalability for accuracy of representation, the answers returned by the FERRARI index are exact. This is achieved by combining the approximate representation of reachability encoded in the index entries with additional online search during query processing. The compact nature of the index together with restricted online search effort, relying on the combination of several effective heuristics, lead to a space-efficient yet fast index structure for reachability queries. The FERrari index allows processing of this kind of queries over large graphs with query processing times in the order of a few microseconds.

2. The PathSketch index for distance and shortest path queries over massive graphs that may reside on secondary storage (disk or flash). Distance queries enable to answer questions of the form:

    - *How closely related is a pair of entities?*

    The second query type – (approximate shortest) path queries – allow to answer questions of the form:

    - *What is the structure of the relationship between a pair of entities?*

    The former question is answered by a number, indicating the strength of the relationship, whereas an answer to the latter question corresponds to a path – a sequence of intermediate entities that provide an (indirect) connection between the query entities. The PathSketch index allows efficient processing of both query types. The index entries of PathSketch consist of vertex labels, containing for every vertex in the graph, two trees connecting the vertex to a set of carefully selected seeds. At query time, given a pair of query vertices (s, t), multiple short paths connecting s with t are generated by intersecting the respective trees. In order to retrieve a potentially shorter path as well as additional connections, a limited number of additional vertices is expanded by retrieving the adjacency list from the graph. The number of vertices that can be expanded during query processing is upper-bounded by a predefined budget, providing a direct control over the tradeoff between desired accuracy of results and query processing time. We propose algorithms for efficient index construction, together with serialization and compression techniques ensuring a compact size of the index entries. PathSketch provides approximate answers to shortest path and distance queries over massive graphs with up to billions of edges in the order of a few milliseconds.

In the second part of this work, we have switched our focus from the efficiency aspect towards facilitating a more expressive variant of relationship analysis. To this end, we have proposed

3. the Espresso algorithm for explaining the relationship between two sets of entities in a knowledge graph. We have proposed a novel graph-theoretic concept – relatedness cores – to describe a relationship by identifying certain key events that played an important role in the interaction of the involved query entities. This allows us to answer questions of the form

    - *How can we characterize the relationship between the United States and countries from the Middle East over the last few years?*
    - *In what way have American technology companies interacted with countries from Asia?*

    The proposed solution concept of relatedness cores can be regarded as a small, semantically coherent set of intermediate entities that are relevant to both input sets. The Espresso algorithm works by first identifying a central *event* entity, corresponding to a real-world event such as a conflict involving countries from the query sets, high-profile political meetings, sports events, etc. These central entities are determined by combining the results of random walk processes from either query set. In the second algorithmic stage, a coherent subgraph is computed from the surroundings of the central event entity and connected to the query vertices. ESPRESSO combines features from several data sources, including semantic types of entities derived from the YAGO knowledge base, relatedness measures computed both from the structure of the knowledge graphs as well as textual descriptions of the entities, and entity importance over time extracted from page view statistics of the Wikipedia encyclopedia. The usefulness of the solutions computed by ESPRESSO is confirmed by several user studies.

The combination of the techniques proposed in this thesis allows to conduct insightful and interactive forms of relationship analysis over very large graphs.

Future Directions
We see a number of open issues that should be addressed in future work:

- **Integrated Relationship Analysis Platform.** In this thesis, we provide three algorithmic building blocks for relationship analysis. A logical and promising direction for future work is to extend and combine the proposed approaches into an integrated platform. Interesting research problems arise with regard to visualization, novel query languages, integration of the proposed index structures into other, higher-order graph algorithms, and the addition of further algorithmic building blocks.

- **Specialized Index Structure for Knowledge Graphs.** The notion of knowledge graphs has gained significant momentum over recent years, to a large part due to their application in web search and content recommendation. The amount of knowledge encoded in these structures is growing at rapid pace via large-scale harvesting over web sources. This class of graphs offers appealing opportunities for further research regarding efficient processing. Distinguishing properties such as semantically typed vertices and labeled relationships bear potential for dedicated index structures that can speed up analysis tasks over this important class of graphs, including – but not restricted to – label-constrained reachability and path queries.

- **Semantic Graph Mining.** On a similar note, knowledge graphs offer many interesting possibilities for novel data mining and graph mining approaches that compute semantically meaningful results. The ESPRESSO algorithm has focused on extracting explanations of relationships in knowledge graphs, specifically targeting the computation of *(semantically) coherent* and *interesting* results. Mapping these requirements into the space of graph algorithmics is a challenging problem. Examples range from new graph theoretic concepts (such as relatedness cores) to the integration of external data sources (such as disambiguated news articles, textual descriptions, popularity over time, etc.) into an enriched, multi-modal graph structure.

Part IV

Appendix

A

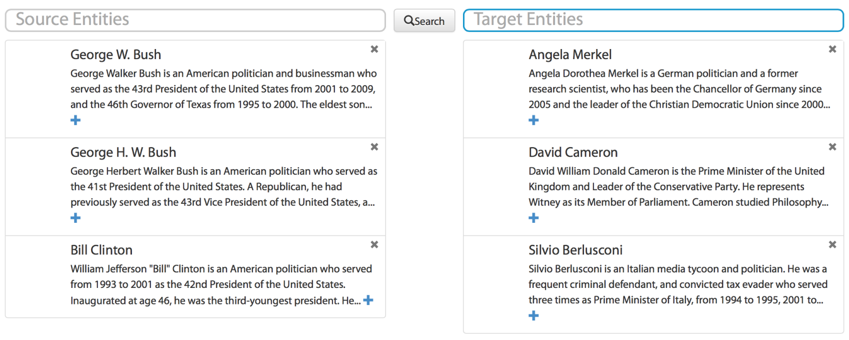
The Instant Espresso System Demonstration

We have created a web-based interface showcasing the Espresso algorithm described in detail in Chapter 6. Using this interface, which is available online at the URL

`http://espresso.mpi-inf.mpg.de`

users can specify two sets of entities from the Espresso Knowledge Graph, and are displayed a graph visualization of the computed relatedness cores. The input interface is shown in Figure A.1.

We show examples for computed graph visualizations in Figures A.2-A.2. The thickness of an individual edge reflects the relationship strength. In addition each



George W. Bush
George Walker Bush is an American politician and businessman who served as the 43rd President of the United States from 2001 to 2009, and the 46th Governor of Texas from 1995 to 2000. The eldest son...

George H. W. Bush
George Herbert Walker Bush is an American politician who served as the 41st President of the United States. A Republican, he had previously served as the 43rd Vice President of the United States, a...

Bill Clinton
William Jefferson "Bill" Clinton is an American politician who served from 1993 to 2001 as the 42nd President of the United States. Inaugurated at age 46, he was the third-youngest president. He...

Angela Merkel
Angela Dorothea Merkel is a German politician and a former research scientist, who has been the Chancellor of Germany since 2005 and the leader of the Christian Democratic Union since 2000...

David Cameron
David William Donald Cameron is the Prime Minister of the United Kingdom and Leader of the Conservative Party. He represents Witney as its Member of Parliament. Cameron studied Philosophy...

Silvio Berlusconi
Silvio Berlusconi is an Italian media tycoon and politician. He was a frequent criminal defendant, and convicted tax evader who served three times as Prime Minister of Italy, from 1994 to 1995, 2001 to...

Figure A.1: User input to select query entity sets

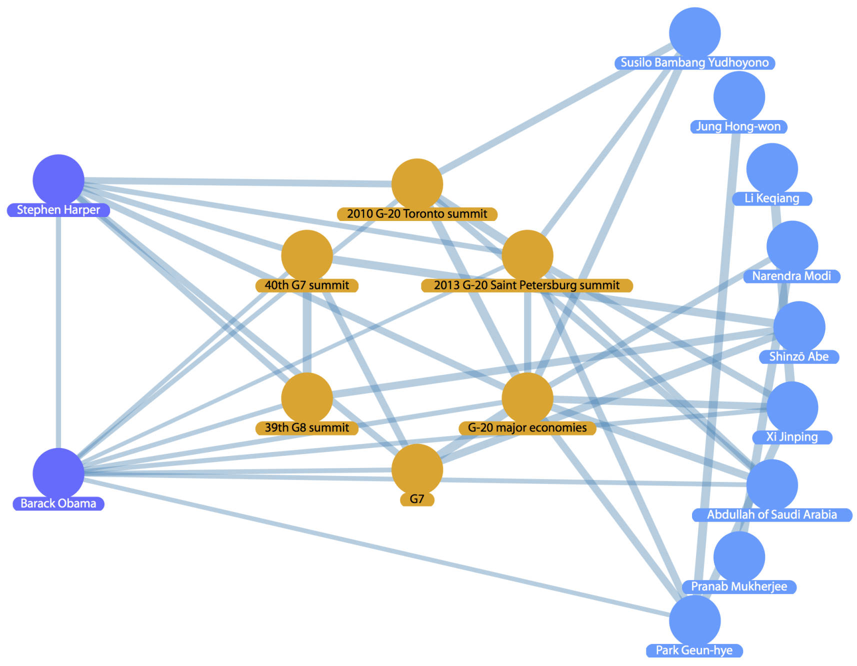


Figure A.2: Anecdotal result: Espresso relatedness cores for Q2

individual entity is shown with a thumbnail image. Upon clicking on an entity vertex, a short description snippet of the entity is shown together with a larger image and a link to the respective Wikipedia article.

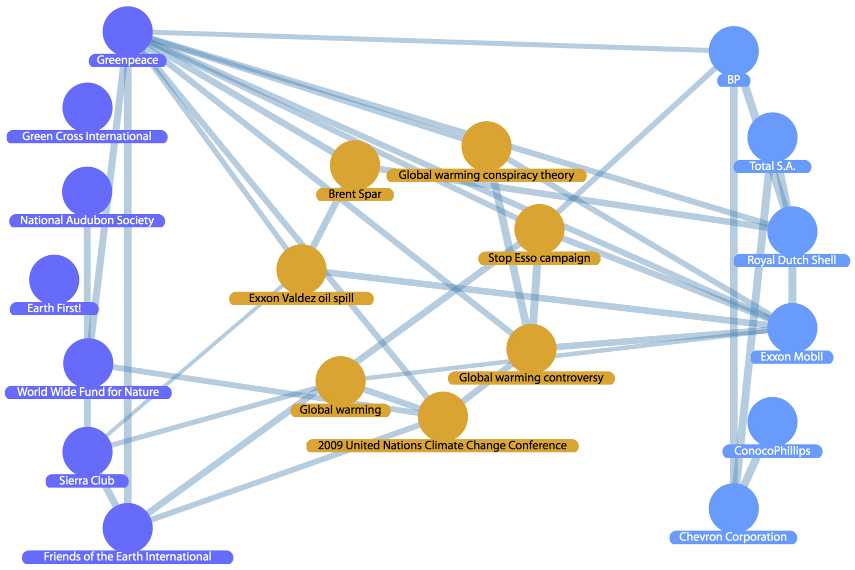


Figure A.3: Anecdotal result: Espresso relatedness cores for Q8

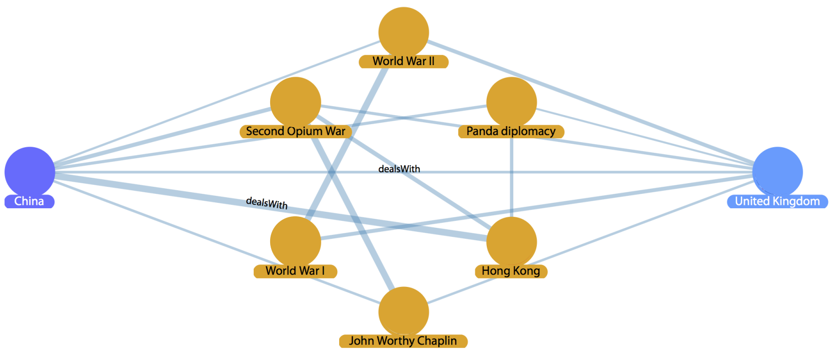


Figure A.4: Anecdotal result: Espresso relatedness cores for Q12

Bibliography

Daniel J. Abadi, Adam Marcus, Samuel Madden, and Kate Hollenbach. SW-Store: A Vertically Partitioned DBMS for Semantic Web Data Management. *The VLDB Journal*, 18(2):385–406, 2009. (cited on page 32)

Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In *SODA'10: Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 782–793. ACM/SIAM, 2010. ISBN 978-0-898716-98-6. (cited on page 109)

Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. VC-Dimension and Shortest Path Algorithms. In *Automata, Languages and Programming*, volume 6755 of *Lecture Notes in Computer Science*, pages 690–699. Springer, 2011. (cited on page 111)

Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Hierarchical Hub Labelings for Shortest Paths. In *ESA'12: Proceedings of the 20th Annual European Conference on Algorithms*, volume 7501 of *Lecture Notes in Computer Science*, pages 24–35. Springer, 2012. (cited on page 109)

Alok Aggarwal and Jeffrey S. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, September 1988. ISSN 0001-0782. (cited on page 34)

Rakesh Agrawal, Alexander Borgida, and H. V. Jagadish. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 253–262. ACM, 1989. (cited on pages 45, 46, 47, 52, 56, 61 and 63)

Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. DBXplorer: A System for Keyword-Based Search over Relational Databases. In *ICDE'02: Proceedings of the 18th IEEE International Conference on Data Engineering*, pages 5–16. IEEE, 2002. (cited on page 153)

Deepak Ajwani, Roman Dementiev, and Ulrich Meyer. A Computational Study of External-Memory BFS Algorithms. In *SODA'06: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 601–610. ACM/SIAM, 2006. (cited on page 35)

Deepak Ajwani, Ulrich Meyer, and Vitaly Osipov. Improved External Memory BFS Implementations. In *ALENEX'07: Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments*. SIAM, 2007. (cited on page 35)

Deepak Ajwani, Ulrich Meyer, and David Veith. I/O-efficien Hierarchical Diameter Approximation. In *ESA'12: Proceedings of the 20th Annual European Conference on Algorithms*, volume 7501 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2012. (cited on page 35)

Takuya Akiba, Christian Sommer, and Ken-ichi Kawarabayashi. Shortest-Path Queries for Complex Networks: Exploiting Low Tree-width Outside the Core. In *EDBT'12: Proceedings of the 15th International Conference on Extending Database Technology*, pages 144–155. ACM, 2012. (cited on pages 110 and 111)

Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast Exact Shortest-Path Distance Queries on Large Networks by Pruned Landmark Labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 349–360. ACM, 2013. (cited on pages 109 and 113)

Leman Akoglu, Duen Horng Chau, Christos Faloutsos, Nikolaj Tatti, Hanghang Tong, and Jilles Vreeken. Mining Connection Pathways for Marked Nodes in Large Graphs. In *SDM'13: Proceedings of the 13th SIAM International Conference on Data Mining*, pages 37–45. SIAM, 2013. (cited on page 153)

Noga Alon, Yossi Matias, and Mario Szegedy. The Space Complexity of Approximating the Frequency of Moments. In *STOC'96: Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pages 20–29. ACM, 1996. (cited on page 35)

Reid Andersen and Kumar Chellapilla. Finding Dense Subgraphs with Size Bounds. In *Algorithms and Models for the Web-Graph*, volume 5427 of *Lecture Notes in Computer Science*, pages 25–37. Springer, 2009. (cited on page 154)

Albert Angel, Nikos Sarkas, Nick Koudas, and Divesh Srivastava. Dense Subgraph Maintenance under Streaming Edge Weight Updates for Real-time Story Identification. *Proceedings of the VLDB Endowment (PVLDB)*, 5(6):574–585, February 2012. (cited on page 154)

Yuichi Asahiro, Kazuo Iwama, Hisao Tamaki, and Takeshi Tokuyama. Greedily Finding a Dense Subgraph. *Journal of Algorithms*, 34(2):203–221, 2000. (cited on page 154)

Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. DBpedia: A Nucleus for a Web of Open Data. In *The Semantic Web*, volume 4825 of *Lecture Notes in Computer Science*, pages 722–735. Springer, 2007. (cited on page 132)

Bahman Bahmani, Ravi Kumar, and Sergej Vassilvitskii. Densest Subgraph in Streaming and MapReduce. In *Proceedings of the VLDB Endowment (PVLDB)*, volume 5, pages 454–465. VLDB Endowment, 2012. (cited on page 154)

Albert-László Barabási and Albert Réka. Emergence of Scaling in Random Networks. *Science*, 286:509–512, 1999. (cited on page 17)

Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In Transit to Constant Time Shortest-Path Queries in Road Networks. In *ALENEX'07: Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments*. SIAM, 2007. (cited on pages 22 and 111)

Scott Beamer, Krste Asanović, and David Patterson. Direction-Optimizing Breadth-First Search. In *SC'12: Proceedings of the 2012 International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012. (cited on pages 95 and 103)

Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and Shashank Sudarshan. Keyword Searching and Browsing in Databases using BANKS. In *ICDE'02: Proceedings of the 18th IEEE International Conference on Data Engineering*, pages 431–440. IEEE, 2002. (cited on page 153)

Burton Howard Bloom. Space/Time Tradeoffs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13:422–426, 1970. (cited on page 87)

Paolo Boldi, Massimo Santini, and Sebastiano Vigna. A Large Time-Aware Graph. *SIGIR Forum*, 42(2):33–38, 2008. (cited on pages 66 and 112)

Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: A Collaboratively Created Graph Database for Structuring Human Knowledge. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1247–1250. ACM, 2008. (cited on pages 91, 132 and 135)

Béla Bollobás and Oliver Riordan. The Diameter of a Scale-Free Random Graph. *Combinatorica*, 24:5–34, 2004. (cited on page 17)

Sergey Brin and Lawrence Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems*, 30:107–117, 1998. (cited on page 21)

Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph Structure in the Web. *Computer Networks*, 33(1-6):309–320, 2000. (cited on page 23)

Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: An Architecture for Storing and Querying RDF Data and Schema Information. In *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*, pages 197–222. MIT Press, 2003. (cited on page 32)

Diego Ceccarelli, Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Salvatore Trani. Learning Relatedness Measures for Entity Linking. In *CIKM'13: Proceedings of the 22nd ACM International Conference on Information and Knowledge Management*, pages 139–148. ACM, 2013. (cited on page 141)

Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. In *ICWSM'10: Proceedings of the 4th International AAAI Conference on Weblogs and Social Media*. AAAI, 2010. (cited on page 66)

Moses Charikar. Greedy Approximation Algorithms for Finding Dense Components in a Graph. In *Proceedings of the Third International Workshop on Approximation Algorithms for Combinatorial Optimization*, APPROX '00, pages 84–95. Springer, 2000. (cited on pages 148 and 154)

Duen Horng Chau, Christos Faloutsos, Hanghang Tong, Jason I. Hong, Brian Gallagher, and Tina Eliassi-Rad. Graphite: A Visual Query System for Large Graphs. In *ICDMW'08: Proceedings of the 8th IEEE International Conference on Data Mining (Workshops)*, pages 963–966. IEEE, 2008. (cited on page 153)

Yangjun Chen and Yibin Chen. An Efficient Algorithm for Answering Graph Reachability Queries. In *ICDE'08: Proceedings of the 24th IEEE International Conference on Data Engineering*, pages 893–902. IEEE, 2008. (cited on pages 45 and 63)

James Cheng, Yiping Ke, Wilfred Ng, and Jeffrey Xu Yu. Context-Aware Object Connection Discovery in Large Graphs. In *ICDE'09: Proceedings of the 25th IEEE International Conference on Data Engineering*, pages 856–867. IEEE, 2009. (cited on page 153)

James Cheng, Zechao Shang, Hong Cheng, Haixun Wang, and Jeffrey Xu Yu. K-Reach: Who is in Your Small World. *Proceedings of the VLDB Endowment (PVLDB)*, 5(11):1292–1303, 2012. (cited on page 109)

Jiefeng Cheng and Jeffrey Xu Yu. On-Line Exact Shortest Distance Query Processing. In *EDBT'09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 481–492. ACM, 2009. (cited on page 109)

Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-Memory Graph Algorithms. In *SODA'95: Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149. SIAM, 1995. (cited on page 35)

Joel Coffman and Alfred Weaver. An Empirical Performance Evaluation of Relational Keyword Search Techniques. *IEEE Transactions on Knowledge and Data Engineering*, 99, 2013. (cited on page 153)

Edith Cohen. Size-Estimation Framework with Applications to Transitive Closure and Reachability. In *Journal of Computer and System Sciences*, volume 55, pages 441–453, Orlando, FL, USA, December 1997. Academic Press, Inc. (cited on page 56)

Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and Distance Queries via 2-Hop Labels. In *SODA'02: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM/SIAM, 2002. (cited on pages 45, 63 and 108)

Richard Cole, Ramesh Hariharan, Moshe Lewenstein, and Ely Porat. A Faster Implementation of the Goemans-Williamson Clustering Algorithm. In *SODA'01: Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 17–25. SIAM, 2001. (cited on page 147)

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009. (cited on pages 23 and 56)

Anish Das Sarma, Alpa Jain, and Cong Yu. Dynamic Relationship and Event Discovery. In *WSDM'11: Proceedings of the 4th ACM International Conference on Web Search and Data Mining*, pages 207–216. ACM, 2011. (cited on pages 151 and 154)

Atish Das Sarma, Sreenivas Gollapudi, Marc Najork, and Rina Panigrahy. A Sketch-Based Distance Oracle for Web-Scale Graphs. In *WSDM'10: Proceedings of the 3rd ACM International Conference on Web Search and Data Mining*, pages 401–410. ACM, 2010. (cited on pages 75, 77, 78, 91, 93, 105 and 110)

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Proceedings of the 6th USENIX Symposium on Operating System Design and Implementation*, pages 137–150. USENIX Association, 2004. (cited on pages 4 and 36)

Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. Trading Off Space for Passes in Graph Streaming Problems. *ACM Transactions on Algorithms*, 6(1): 1–17, December 2009. (cited on page 36)

Reinhard Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, 4 edition, July 2010. (cited on page 23)

Edsger. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959. (cited on page 75)

David Easley and Jon Kleinberg. *Networks, Crowds, and Markets*. Cambridge University Press, July 2010. (cited on page 23)

Christos Faloutsos, Kevin S. McCurley, and Andrew Tomkins. Fast Discovery of Connection Subgraphs. In *KDD'04: Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 118–127. ACM, 2004. (cited on pages 133 and 152)

Lujun Fang, Anish Das Sarma, Cong Yu, and Philip Bohannon. REX: Explaining Relationships between Entity Pairs. *Proceedings of the VLDB Endowment (PVLDB)*, 5(3):241–252, 2011. (cited on pages 23, 134, 153, 161, 162 and 166)

Uriel Feige, Guy Kortsarz, and David Peleg. The Dense k-Subgraph Problem. *Algorithmica*, 29:410–421, 2001. (cited on page 138)

Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddarth Suri, and Jian Zhang. On Graph Problems in a Semi-Streaming Model. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *Automata, Languages* *and Programming*, volume 3142 of *Lecture Notes in Computer Science*, pages 531–543. Springer, 2004. (cited on pages 27, 36 and 95)

Matteo Fischetti, Horst W Hamacher, Kurt Jørnsten, and Francesco Maffioli. Weighted k-Cardinality Trees: Complexity and Polyhedral Structure. *Networks*, 24:11–21, 1994. (cited on page 139)

Philippe Flajolet and G. Nigel Martin. Probabilistic Counting Algorithms for Data Base Applications. *Journal of Computer and System Sciences*, 31:182–209, 1985. (cited on page 35)

Ada Wai-Chee Fu, Huanhuan Wu, James Cheng, and Raymond Chi-Wing Wong. IS-LABEL: An Independent-Set baed Labeling Scheme for Point-to-Point Distance Querying. In *Proceedings of the VLDB Endowment (PVLDB)*, 2013. (cited on pages 109, 110 and 113)

Evgeniy Gabrilovich, Michael Ringgaard, and Amarnag Subramanya. FACC1: Freebase Annotation of ClueWeb Corpora, Version 1 (release data 2013-06-26, format version 1, correction level 0). `http://lemurproject.org/clueweb12/FACC1/`, 2013. (cited on page 136)

Jun Gao, Ruoming Jin, Jiashuai Zhou, Jeffrey Xu Yu, Xiao Jiang, and Tengjiao Wang. Relational Approach for Shortest Path Discovery over Large Graphs. *Proceedings of the VLDB Endowment (PVLDB)*, 5(4):358–369, 2011. (cited on page 111)

Naveen Garg, Goran Konjevod, and R. Ravi. A Polylogarithmic Approximation Algorithm for the Group Steiner Tree Problem. *Journal of Algorithms*, 37(1):66–84, 2000. (cited on page 149)

Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz. Distance Labeling in Graphs. *Journal of Algorithms*, 53(1):85–112, October 2004. (cited on page 108)

Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *WEA'2008: Proceedings of the 7th International Workshop on Experimental Algorithms*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2008. (cited on pages 22 and 111)

Michel X. Goemans and David P. Williamson. A General Approximation Technique for Constrained Forest Problems. In *SODA'92: Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 307–316. ACM/SIAM, 1992. (cited on page 147)

Andrew V. Goldberg. Finding a Maximum Density Subgraph. Technical report, University of California at Berkeley, 1984. (cited on page 154)

Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A^* Search Meets Graph Theory. Technical Report MSR-TR-2004-24, Microsoft Research, 2004. (cited on pages 78 and 110)

Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI'12: Proceedings of the 10th USENIX Symposium on Operating System Design and Implementation*, pages 17–30. USENIX Association, 2012. (cited on page 38)

Gang Gou and Rada Chirkova. Efficient Algorithms for Exact Ranked Twig-Pattern Matching over Graphs. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 581–594, 2008. (cited on page 74)

Andrey Gubichev, Srikanta Bedathur, Stephan Seufert, and Gerhard Weikum. Fast and Accurate Estimation of Shortest Paths in Large Graphs. In *CIKM'10: Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, pages 499–508. ACM, 2010. (cited on pages 75, 76, 79, 93, 110, 111 and 117)

Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 289–300. ACM, 2014. (cited on page 32)

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968. (cited on page 110)

Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. BLINKS: Ranked Keyword Searches on Graphs. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 305–316. ACM, 2007. (cited on page 153)

Chinmay Hegde, Piotr Indyk, and Ludwig Schmidt. A Fast, Adaptive Variant of the Goemans-Williamson Scheme for the Prize-Collecting Steiner Tree Problem. In *11th DIMACS Implementation Challenge in Collaboration with ICERM: Steiner Tree Problems*, 2014. (cited on page 147)

C. S. Helvig, Gabriel Robins, and Alexander Zelikovsky. An Improved Approximation Scheme for the Group Steiner Problem. *Networks*, 37, 2001. (cited on page 149)

Johannes Hoffart, Fabian Suchanek, Klaus Berberich, and Gerhard Weikum. YAGO2: A Spatially and Temporally Enhanced Knowledge Base from Wikipedia. Research Report MPI-I-2010-5-007, Max Planck Institute for Informatics, November 2010. (cited on page 135)

Johannes Hoffart, Mohamed Amir Yosef, Ilaria Bordino, Hagen Fürstenau, Manfred Pinkal, Marc Spaniol, Bilyana Taneva, Stefan Thater, and Gerhard Weikum. Robust Disambiguation of Named Entities in Text. In *EMNLP'11: Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 782–792. Association for Computational Linguistics, 2011. (cited on page 23)

Johannes Hoffart, Stephan Seufert, Dat Ba Nguyen, Martin Theobald, and Gerhard Weikum. KORE: Keyphrase Overlap Relatedness for Entity Disambiguation. In *CIKM'12: Proceedings of the 21th ACM International Conference on Information and Knowledge Management*, pages 545–555. ACM, 2012. (cited on pages 66, 136 and 141)

Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. YAGO2: A Spatially and Temporally Enhanced Knowledge Base from Wikipedia. *Artificial Intelligence*, 194:28–61, 2013. (cited on pages 23, 32, 44, 91 and 132)

M. Shahriar Hossain, Patrick Butler, Arnold P. Boedihardjo, and Naren Ramakrishnan. Storytelling in Entity Networks to Support Intelligence Analysts. In *KDD'12: Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1375–1383. ACM, 2012. (cited on page 154)

Vagelis Hristidis and Yannis Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Data Bases*, pages 670–681. VLDB Endowment, 2002. (cited on page 153)

Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient IR-Style Keyword Search over Relational Databases. In *VLDB'03: Proceedings of the 29th International Conference on Very Large Data Bases*, pages 850–861. VLDB Endowment, 2003. (cited on page 153)

Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL Querying of Large RDF Graphs. *Proceedings of the VLDB Endowment (PVLDB)*, 4(11):1123–1134, 2011. (cited on page 32)

H. V. Jagadish. A Compression Technique to Materialize Transitive Closure. *ACM Transactions on Database Systems*, 15(4):558–598, December 1990. (cited on page 63)

Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. Efficiently Answering Reachability Queries on Very Large Directed Graphs. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 595–608. ACM, 2008. (cited on pages 45 and 66)

Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 3-HOP: A High-Compression Indexing Scheme for Reachability Query. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 813–826. ACM, 2009. (cited on pages 45, 64 and 66)

Ruoming Jin, Ning Ruan, Yang Xiang, and Haixun Wang. Path-Tree: An Efficient Reachability Indexing Scheme for Large Directed Graphs. *ACM Transactions on Database Systems (TODS)*, 36(1):7, 2011. (cited on pages 45, 63, 64 and 65)

Ruoming Jin, Ning Ruan, Saikat Dey, and Jeffrey Xu Yu. SCARAB: Scaling Reachability Computation on Large Graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 169–180, 2012a. (cited on page 63)

Ruoming Jin, Ning Ruan, Yang Xiang, and Victor Lee. A Highway-Centric Labeling Approach for Answering Distance Queries on Large Sparse Graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 445–456. ACM, 2012b. (cited on page 109)

Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional Expansion for Keyword Search on Graph Databases. In *VLDB'05: Proceedings of the 31st International Conference on Very Large Data Bases*, pages 505–516. VLDB Endowment, 2005. (cited on page 153)

Mehdi Kargar and Aijun An. Keyword Search in Graphs: Finding r-Cliques. *Proceedings of the VLDB Endowment (PVLDB)*, 4(10):681–692, 2011. (cited on page 153)

Gjergji Kasneci, Shady Elbassuoni, and Gerhard Weikum. MING: Mining Informative Entity-Relationship Subgraphs. In *CIKM'09: Proceedings of the 18th ACM International Conference on Information and Knowledge Management*, pages 1653–1656. ACM, 2009a. (cited on pages 134 and 153)

Gjergji Kasneci, Maya Ramanath, Mauro Sozio, Fabian M. Suchanek, and Gerhard Weikum. STAR: Steiner-Tree Approximation in Relationship Graphs. In *ICDE'09: Proceedings of the 25th IEEE International Conference on Data Engineering*, pages 868–879. IEEE, 2009b. (cited on page 153)

Irit Katriel and Ulrich Meyer. *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*, pages 62–84. Springer, 2003. (cited on page 35)

Jon Kleinberg, Aleksandrs Slivkins, and Tom Wexler. Triangulation and Embedding via a Small Set of Beacons. In *FOCS'04: Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*, pages 444–453. IEEE, 2004. (cited on page 77)

Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a Social Network or a News Media? In *WWW'10: Proceedings of the 19th International World Wide Web Conference*, pages 591–600. ACM, 2010. (cited on page 112)

Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI'12: Proceedings of the 10th USENIX Symposium on Operating System Design and Implementation*. USENIX Association, 2012. (cited on page 38)

Victor E. Lee, Ning Ruan, Ruoming Jin, and Charu Aggarwal. *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*, chapter 10: A Survey of Algorithms for Dense Subgraph Discovery, pages 303–336. Springer, 2010. (cited on page 154)

Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics*, 6:29–123, 2009. (cited on pages 111 and 112)

Jimmy Lin and Michael Schatz. Design Patterns for Efficient Graph Algorithms in MapReduce. In *MLG'10: Proceedings of the 8th Workshop on Mining and Learning with Graphs*, pages 78–85. ACM, 2010. (cited on page 37)

Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: A New Framework for Parallel Machine Learning. In *UAI'10: Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence*, pages 340–349. AUAI Press, 2010. (cited on page 38)

Yucheng Low, Danny Bickson, Joseph E. Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment (PVLDB)*, 5:716–727, April 2012. (cited on page 38)

Anirban Majumder, Samik Datta, and K. V. M. Naidu. Capacitated Team Formation Problem on Social Networks. In *KDD'12: Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1005–1013, 2012. (cited on page 74)

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 135–146. ACM, 2010. (cited on pages 4 and 37)

Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval.* Cambridge University Press, 2008. (cited on page 107)

William F. McColl. Scalability, Portability, and Predictability: The BSP Approach to Parallel Programming. *Future Generation Computer Systems*, pages 265–272, 1996. (cited on page 37)

Kurt Mehlhorn and Ulrich Meyer. External-Memory Breadth-First Search with Sublinear I/O. In *ESA'02: Proceedings of the 10th Annual European Conference on Algorithms*, pages 723–735. Springer, 2002. (cited on page 35)

Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox.* Springer, 2008. (cited on page 34)

Ulrich Meyer and Norbert Zeh. I/O-Efficient Shortest Path Algorithms for Undirected Graphs with Random of Bounded Edge Lenghts. *ACM Transactions on Algorithms*, 8(3 (Article 22)), 2012. (cited on page 35)

David Milne and Ian H. Witten. An Effective, Low-Cost Measure of Semantic Relatedness Obtained from Wikipedia Links. In *WIKIAI'08: Proceedings of the 2008 AAAI Workshop on Wikipedia and Artificial Intelligence*. AAAI, 2008. (cited on pages 136 and 140)

Alan Mislove, Massimiliano Marcon, Krishna Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and Analysis of Online Social Networks. In *IMC'07: Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, pages 29–42. ACM, 2007. (cited on page 22)

Kameshwar Munagala and Abhiram Ranade. I/O-Complexity of Graph Algorithms. In *SODA'99: Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 687–694. SIAM, 1999. (cited on page 35)

J. Ian Munro and Mike S. Paterson. Selection and Sorting with Limited Storage. *Theoretical Computer Science*, 12:315–323, 1980. (cited on page 35)

S. Muthukrishnan. Data Streams: Algorithms and Applications. *Foundations and Trends in Theoretical Computer Science*, 1:117–236, 2005. (cited on page 36)

Thomas Neumann and Gerhard Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *The VLDB Journal*, 19(1):91–113, 2010. (cited on pages 32 and 113)

Esko Nuutila. An Experimental Study on Transitive Closure Representations. Technical Report TKO-B134, Helsinki University of Technology, 1996. (cited on pages 63 and 64)

Christopher R. Palmer, Phillip B. Gibbons, and Christos Faloutsos. ANF: A Fast and Scalable Tool for Data Mining in Massive Graphs. In *KDD'02: Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 81–90. ACM, 2002. ISBN 1-58113-567-X. (cited on page 56)

David Peleg. Proximity-Preserving Labeling Schemes. *Journal Graph Theory*, pages 167–176, 2000. (cited on page 108)

Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. Fast Shortest Path Distance Estimation in Large Networks. In *CIKM'09: Proceedings of the 18th ACM International Conference on Information and Knowledge Management*, pages 867–876. ACM, 2009. (cited on pages 75, 77, 78 and 110)

Jeffrey Pound, Alexander K. Hudek, Ihab F. Ilyas, and Grant Weddell. Interpreting Keyword Queries over Web Knowledge Bases. In *CIKM'12: Proceedings of the 21th ACM International Conference on Information and Knowledge Management*, pages 305–314. ACM, 2012. (cited on page 23)

Zichao Qi, Yanghua Xiao, Bin Shao, and Haixun Wang. Toward a Distance Oracle for Billion-Node Graphs. *Proceedings of the VLDB Endowment (PVLDB)*, 7(2): 61–72, 2013. (cited on page 110)

Miao Qiao, Hong Cheng, Lijun Chang, and Jeffrey Xu Yu. Approximate Shortest Distance Computing: A Query Dependent Local Landmark Scheme. *IEEE Transactions on Knowledge and Data Engineering*, 2012. (cited on pages 77 and 110)

Cartic Ramakrishnan, William H. Milnor, Matthew Perry, and Amit P. Sheth. Discovering Informative Connection Subgraphs in Multi-Relational Graphs. *SIGKDD Explorations*, 7(2):56–63, 2005. (cited on page 153)

Gabriele Reich and Peter Widmayer. Beyond Steiner's Problem: A VLSI Oriented Generalization. In *Proceedings of the 15th International Workshop on Graph-theoretic Concepts in Computer Science*, pages 196–210. Springer, 1990. (cited on pages 149 and 151)

Michael Rice and Vassilis J. Tsotras. Graph Indexing of Road Networks for Shortest Path Queries with Label Restrictions. *Proceedings of the VLDB Endowment (PVLDB)*, 4(2):69–80, November 2010. ISSN 2150-8097. (cited on page 111)

Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media, 2013. (cited on pages 27, 29 and 30)

José F. Rodrigues, Jr., Hanghang Tong, Agma J. M. Traina, Christos Faloutsos, and Jure Leskovec. Gmine: A System for Scalable, Interactive Graph Visualization and Mining. In *VLDB'06: Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 1195–1198. VLDB Endowment, 2006. (cited on page 153)

Ning Ruan. personal communication, 2012. (cited on page 65)

Jan Matthias Ruhl. *Efficient Algorithms for New Computational Models*. PhD thesis, Massachusetts Institute of Technology, September 2003. (cited on page 36)

Semih Salihoglu and Jennifer Widom. GPS: A Graph Processing System. In *SSDBM'13: Proceedings of the 25th International Conference on Scientific and Statistical Database Management*. ACM, 2013. (cited on page 37)

Semih Salihoglu and Jennifer Widom. Optimizing Graph Algorithms on Pregel-like Systems. *Proceedings of the VLDB Endowment (PVLDB)*, 7:577–588, March 2014. (cited on page 37)

Peter Sanders and Dominik Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *ESA'05: Proceedings of the 13th Annual European Conference on Algorithms*, pages 568–579. Springer, 2005. (cited on pages 22 and 111)

Ralf Schenkel, Anja Theobald, and Gerhard Weikum. HOPI: An Efficient Connection Index for Complex XML Document Collections. In *EDBT'04: Proceedings of the 9th International Conference on Extending Database Technology*, volume 2992 of *Lecture Notes in Computer Science*, pages 237–255. Springer, 2004. (cited on pages 45, 64 and 109)

Ralf Schenkel, Anja Theobald, and Gerhard Weikum. Efficient Creation and Incremental Maintenance of the HOPI Index for Complex XML Document Collections. In *ICDE'06: Proceedings of the 22th IEEE International Conference on Data Engineering*, pages 360–371. IEEE, 2006. (cited on pages 108 and 109)

Arie Segev. The Node-Weighted Steiner Tree Problem. *Networks*, 17:1–17, 1987. (cited on page 145)

Stephan Seufert, Avishek Anand, Srikanta J. Bedathur, and Gerhard Weikum. FERRARI: Flexible and Efficient Reachability Range Assignment for Graph Indexing. In *ICDE'13: Proceedings of the 29th IEEE International Conference on Data Engineering*. IEEE, 2013. (cited on page 29)

Clifford A. Shaffer. *Data Structures and Algorithm Analysis*. Dover Publications, Mineola, NY, United States, 2011. (cited on page 106)

John C. Sheperdson and Howard E. Sturgis. Computability of Recursive Functions. *Journal of the ACM*, 10:217–255, 1963. (cited on page 34)

Julian Shun and Guy Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *PPoPP'13: Proceedings of the 2013 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 135–146. ACM, 2013. (cited on pages 95 and 103)

Christian Sommer. Shortest Path Queries in Static Networks. Preprint, 2012. (cited on page 111)

Mauro Sozio and Aristides Gionis. The Community-search Problem and How to Plan a Successful Cocktail Party. In *KDD'10: Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 939–948. ACM, 2010. (cited on pages 148 and 154)

Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. YAGO: A Core of Semantic Knowledge. In *WWW'07: Proceedings of the 16th International World Wide Web Conference*, pages 697–706. ACM, 2007. (cited on page 132)

Yufei Tao, Cheng Sheng, and Jian Pei. On k-skip Shortest Paths. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 421–432. ACM, 2011. (cited on page 111)

Mikkel Thorup and Uri Zwick. Approximate Distance Oracles. *Journal of the ACM*, 52(1):1–24, 2005. (cited on page 109)

Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From "Think Like a Vertex" to "Think Like a Graph". *Proceedings of the VLDB Endowment (PVLDB)*, 7(3):193–204, November 2013. (cited on page 37)

Hanghang Tong and Christos Faloutsos. Center-Piece Subgraphs: Problem Definition and Fast Solutions. In *KDD'06: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 404–413. ACM, 2006. (cited on pages 133, 140, 141, 142, 143, 153, 157, 163 and 166)

Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. Fast Random Walk with Restart and its Applications. In *ICDM'06: Proceedings of the 6th IEEE International Conference on Data Mining*, pages 613–622, 2006. (cited on pages 141, 142 and 143)

Jeffrey Travers and Stanley Milgram. An Experimental Study of the Small World Problem. *Sociometry*, 32(4):425–443, 1969. (cited on page 22)

Konstantin Tretyakov, Abel Armas-Cervantes, Luciano García-Ba nuelos, Jaak Vilo, and Marlon Dumas. Fast Fully Dynamic Landmark-Based Estimation of Shortest Path Distances in Very Large Graphs. In *CIKM'11: Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, pages 1785–1794. ACM, 2011. (cited on page 111)

Silke Trißl and Ulf Leser. Fast and Practical Indexing and Querying of Large Graphs. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 845–856. ACM, 2007. (cited on pages 45 and 64)

Charalampos E. Tsourakakis, Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Maria A. Tsiarli. Denser than the Densest Subgraph: Extracting Optimal Quasi-Cliques with Quality Guarantees. In *KDD'13: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 104–112. ACM, 2013. (cited on page 154)

Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990. (cited on page 37)

Sebastiaan J. van Schaik and Oege de Moor. A Memory Efficient Reachability Data Structure Through Bit Vector Compression. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 913–924. ACM, 2011. (cited on pages 29, 45, 52, 63 and 64)

Vijay V. Vazirani. *Approximation Algorithms*. Springer, 2001. (cited on page 23)

Monique V. Vieira, Bruno M. Fonseca, Rodrigo Damazio, Paulo B. Golgher, Davi de Castro Reis, and Berthier Ribeiro-Neto. Efficient Search Ranking in Social Networks. In *CIKM'07: Proceedings of the 16th ACM International Conference on Information and Knowledge Management*, pages 563–572. ACM, 2007. (cited on page 74)

Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. Dual Labeling: Answering Graph Reachability Queries in Constant Time. In *ICDE'06: Proceedings of the 22th IEEE International Conference on Data Engineering*, page 75. IEEE, 2006. (cited on pages 45 and 63)

Fang Wei. TEDI: Efficient Shortest Path Query Answering on Graphs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 99–110. ACM, 2010. (cited on page 111)

Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, and Dave Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *SWDB'03: Proceedings of the 1st International Workshop on Semantic Web and Databases*, pages 131–150. VLDB Endowment, 2003. (cited on page 32)

Yanghua Xiao, Wentao Wu, Jian Pei, Wei Wang, and Zhenying He. Efficiently Indexing Shortest Paths by Eploiting Symmetry in Graphs. In *EDBT'09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 493–504. ACM, 2009. (cited on page 111)

Jeffrey Xu Yu and Jiefeng Cheng. *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*, chapter 6: Graph Reachability Queries: A Survey, pages 181–215. Springer, 2010. (cited on page 63)

Hilmi Yıldırım, Vineet Chaoji, and Mohammed J. Zaki. GRAIL: Scalable Reachability Index for Large Graphs. In *Proceedings of the VLDB Endowment (PVLDB)*, volume 3, pages 276–284. VLDB Endowment, 2010. (cited on pages 29, 45, 50, 52 and 64)

Hilmi Yıldırım, Vineet Chaoji, and Mohammed J. Zaki. GRAIL: A Scalable Index for Reachability Queries in Very Large Graphs. *The VLDB Journal*, 21(4): 509–534, 2012. (cited on pages 29, 45, 50, 51, 52, 62, 64, 65, 66 and 67)

Xianghong Zhou, Ming-Chih J. Kao, and Wing Hung Wong. Transitive Functional Annotation by Shortest-Path Analysis of Gene Epression Data. *Proceedings of the National Academy of Sciences*, 99(20):12783–12788, 2002. (cited on page 74)