CONSTRAINTS AND CHANGES

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

von

Irit Katriel

Datum des Kolloquiums: 10.02.2005

Dekan: Prof. Dr. Jörg Eschmeier, Universität des Saarlandes, Saarbrücken

Vorsitzender der Prüfungskomission:

Prof. Dr. Raimund Seidel, Universität des Saarlandes, Saarbrücken

Gutachter:

Prof. Dr. Kurt Mehlhorn, MPI für Informatik, Saarbrücken

Prof. Dr. Martin Skutella, Universität Dortmund, Dortmund

Abstract

This thesis consists of four technical chapters. The first two chapters deal with filtering algorithms for global constraints. Namely, we show improved algorithms for the well known Global Cardinality Constraint. Then we define a new constraint, *UsedBy* and its special case *Same*, and show efficient filtering algorithms for both. All of the filtering algorithms follow the same approach: model the constraint as a flow problem.

The next two chapters deal with dynamic algorithms. That is, algorithms that maintain information about a directed acyclic graph (DAG) while the graph changes. The third chapter deals with the problem of maintaining the topological order of the nodes of a DAG upon a sequence of edge insertions. The fourth chapter deals with the problem of maintaining the longest paths in a directed acyclic graph upon edge insertions and deletions.

Kurze Zusammenfassung

Die vorliegende Arbeit umfasst vier Kapitel technischen Inhaltes. Die ersten beiden behandeln Filteralgorithmen für globale Randbedingungen (engl.: *constraints*): Wir behandeln zunächst verbesserte Algorithmen für den bekannten *Global Cardinality Constraint*. Dann definieren wir den *UsedBy*-Constraint und seinen Spezialfall *Same* und beschreiben effiziente Filteralgorithmen für beide. Für alle Filteralgorithmen verwenden wir den selben Ansatz: Wir modellieren die Randbedingung als Flussproblem.

Die nachfolgenden beiden Kapitel behandeln dynamische Algorithmen; nämlich Algorithmen, die Informationen über einen gerichteten kreisfreien Graphen aufrecht erhalten, während sich der Graph ändert: Kapitel 3 beschreibt die Aufrechterhaltung einer topologischen Sortierung der Knoten eines Graphen unter Kanteneinfügungen. Kapitel 4 behandelt das Problem, die Kenntnis der längsten Pfade eines gerichteten kreisfreien Graphen unter Kanteneinfügungen und -löschungen aufrecht zu erhalten.

Contents

Chapter 1

Introduction

Constraint programming is a programming paradigm in which the programmer models the problem she wishes to solve by a set of constraints over the program's variables. That is, the programmer specifies the semantics of the problem, without the algorithm by which it is to be solved. The actual solution is found by a *constraint solver*, which applies a general solving algorithm combined with constraint-specific heuristics.

Assume that a constraint solver receives a finite collection of constraints over a finite set of variables, each variable x having a finite domain $Dom(x)$ of values that can be assigned to it. Then the search space of all possible assignments of values to variables is finite and can be explored exhaustively: Generate all possible assignments and check, for each, whether it satisfies all of the constraints. In order to speed up the search, constraint solvers repeatedly shrink the search space by applying *filtering* steps in which algorithms that are aware of the semantics of the constraints reach conclusions such as "there is no solution in which the variable x is assigned the value $v \in Dom(x)$". Thus, v can be removed from the domain of x.

As a simple example of filtering, consider the constraint $X < Y$ with $Dom(X) = \{2, 3, 5, 6, 8\}$ and $Dom(Y) = \{1, 4, 5, 7\}$. It is easy to see that there is no solution in which $X = 8$ and there is no solution in which $Y = 1$. Therefore, we can reduce the domains to $Dom(X) = \{2, 3, 5, 6\}$ and $Dom(Y) = \{4, 5, 7\}$. Note that it is not possible to reduce the domains further: for every value in the domain of one of the variables, there is a solution in which the variable is assigned this value. In other words, the domains of X and Y are *consistent* with the constraint $X < Y$.

Two different types of consistency appear in this thesis. They will be formally defined in the relevant chapters, but in this introduction we briefly describe them. The first is *arc consistency*, which was illustrated above. Given a constraint C over a set of variables $X_1, \ldots, X_k$, we say that the domains of these variables are arc consistent if for every $v \in Dom(X_i)$, there is a solution to C in which $X_i = v$. The second is *bound consistency.* Here, we assume that the domain of each variable is an interval $Dom(X_i) = [\underline{X_i}, \overline{X_i}]$. We say that the variables are bound consistent with respect to the constraint if for every X_i, there is a solution to the constraint in which $X_i = \underline{X_i}$ and there is a solution to the constraint in which $X_i = \overline{X_i}$. We say that an algorithm achieves arc (bound) consistency for a certain constraint if it computes arc (bound) consistent domains for the variables without losing any solutions. That is, the algorithm removes a value v from the domain of a variable X only if there is no solution to the constraint in which $X = v$.

Chapters 2 and 3 deal with filtering algorithms for two global constraints. That is, constraints that are defined over a large number of variables. In chapter 2 we obtain a bound consistency algorithm for a well known constraint, *GCC*. Previously, it was only known how to efficiently compute arc consistency for this constraint. Simultanously with our research, another group designed an algorithm that is based on a different approach and can achieve bound consistency for some of the variables [22]. In chapter 3 we define a new constraint called *UsedBy* (and a special case which we call *Same*) and show that it too admits efficient arc and bound consistency computations. Our filtering algorithms follow the flow/matching-based approach that was first used in filtering algorithms by Régin [25]. The idea is to model the constraint by a variable-value graph such that each solution corresponds to a feasible integral flow in this graph. Then, one can show that the strongly connected components of the residual graph define which values can be removed from each variable's domain.

Chapter 4 deals with the problem of maintaining the topological order of the nodes of a DAG online. That is, while inserting new edges. We show that m edge insertions into an n-node DAG can be handled in $O(\min\{m^{3/2} \log n, m^{3/2} + n^2 \log n\})$ time. We then proceed to explore the complexity of the algorithm on structured graphs. We show that the algorithm will require only $O(mk \log^2 n)$ time if the treewidth of the input graph is k, and for the case of trees ($k = 1$), we further improve this to $O(n \log n)$, which is optimal.

Finally, Chapter 5 contains algorithms that maintain the heaviest paths in a DAG under both edge insertions and deletions. This problem was studied by Michel and Van Hentenryck [17] in the context of local search for constraint programming. We improve their algorithms by showing how to handle graphs that include edges with zero or negative weights, and how to obtain improved complexity for the case in which all edge weights are integers.

Chapter 2

Complete Bound Consistency for the Global Cardinality Constraint

The *Global Cardinality Constraint* $GCC(x_1, \cdots, x_n, c_{v_1}, \cdots, c_{v_{n'}})$ is specified on n assignment variables $x_1, \ldots, x_n$ and n' count variables $c_{v_1}, \ldots, c_{v_{n'}}$. The idea is that each assignment variable x_j takes a value in $D = \{v_1, \cdots, v_{n'}\}$ and each value v_i is used exactly c_{v_i} times. With each assignment variable x_j we associate a domain $D_j \subseteq D$, and the domain of a count variable c_{v_i} is an interval $E_i = [L_i, U_i]$. For a tuple $t \in D^n$ and $v \in D$ denote by $occ(v, t)$ the number of occurrences of the value v in t. Then the set S containing all solutions of the GCC is defined as follows:

$$S = \{(w_1, \ldots, w_n; o_1, \ldots, o_{n'}) | \forall_j w_j \in D_j \ \wedge \ \forall_i occ(v_i, (w_1, \ldots, w_n)) = o_i \in E_i\}$$

An example of a problem that can be modeled with a GCC is the *shift assignment problem* [6, 26] in which we are given a set of workers $W = \{W_1, \ldots, W_s\}$ and a set of shifts $S = \{S_1, \ldots, S_t\}$ and the problem is to assign each worker to one of the shifts while fulfilling the constraints posed by the workers and the boss: Each worker W_i specifies in which of the shifts she is willing to work and for each shift S_i the boss specifies a lower and upper bound on the number of workers that should be assigned to this shift. In the GCC, the workers would be represented by the assignment variables and the shifts by the count variables. The domain of an assignment variable would contain the set of shifts that the respective worker is willing to work in and the interval corresponding to each count variable would match the lower and upper bounds specified by the boss for this shift.

Given a constraint with domains for the variables, the first question is whether $S \neq \emptyset$, which means that there is an assignment of values to the variables which satisfies the constraint. The *arc consistency* problem for the assignment variables is to reduce the domains of these variables such that D_j is the projection of S onto its jth component. In the *bound consistency* problem we assume that $v_1 < \ldots < v_{n'}$ and each domain $\mathcal{D}$ is a contiguous interval of values, i.e. $\mathcal{D} = [\underline{\mathcal{D}}, \overline{\mathcal{D}}]$. The problem is to shrink the intervals of both the assignment and the count variables to the minimum sizes such that $S \subseteq D_1 \times \cdots \times D_n \times E_1 \times \cdots \times E_{n'}$. This means that for each $1 \leq j \leq n$, there is at least one tuple in S whose jth component is the smallest (largest) value in D_j. And for $i = 1, \ldots, n'$, there is a tuple in S whose $(n+i)$th component is the smallest (largest) value in E_i.

The *AllDifferent* constraint [31] is the special case of GCC in which $[L_i, U_i] = [0, 1]$ for all i. Naturally, filtering algorithms for AllDifferent appeared first and the generalizations to GCC followed. Two parallel approaches were explored (see Table 2.1). The first is to identify Hall intervals among the values and the second is based on finding a matching or flow in the *variable-value graph*, a bipartite graph that represents the problem at hand. The first approach was used only for bound consistency algorithms while the second yields both arc consistency and bound consistency algorithms.

	Hall Intervals	Matchings/Flows	
	Bound cons.	Arc cons.	Bound cons.
AllDifferent	Puget[20], López-Ortiz [13]	Régin [25]	Mehlhorn, Thiel [16]
GCC	Quimper et al. [22]	Régin [26]	Here

Table 2.1: The two approaches for filtering of AllDifferent and GCC constraints

The righthand side of Table 2.1 shows the history of the matching/flow based approach. It begins with Régin's arc-consistency algorithm for AllDifferent, which finds a matching in the variable-value graph and computes the strongly connected components (SCCs) of an oriented graph defined by this matching.

When computing bound consistency, the domain of each variable is an interval so the variable-value graph is *convex*, which means that the neighborhood[1] of each variable node is a contiguous sequence of value nodes. Mehlhorn and Thiel exploit this structural property of

[1]The neighborhood of a node in a graph is the set of all nodes that are adjacent to it.

the graph to obtain a bound consistency algorithm that does the same as Régin's algorithm, but more efficiently.

For GCC, Régin generalized his AllDifferent algorithm. Instead of a matching he finds a flow in a slightly augmented variable-value graph and then proceeds similarly with an SCC computation. This algorithm runs in time $O(n^{3/2}n')$ and is dominated by the complexity of finding a maximum flow using Ford and Fulkerson's algorithm [10]. We complete this line of research by showing an efficient version of Régin's algorithm for bound consistency, again by exploiting the convexity of the graph. Our algorithm runs in time $O(n + n')$ plus the time required for sorting the variables according to the endpoints of their domains.

In addition, we show two linear-time algorithms that narrow the domains of the count variables. The first is very simple and fast in practice but does not always achieve bound consistency. The second does achieve bound consistency but requires more elaborate data structures. This is the first efficient algorithm that achieves bound consistency for the count variables.

The rest of this chapter is organized as follows. In Section 2.1 we show how the correctness of our construction can be derived from Régin's results on arc consistency. In Sections 2.2–2.4 we show the algorithm for bound consistency of the assignment variables and in Sections 2.5 and 2.6 we show the two algorithms that filter the domains of the count variables. We conclude with some open problems.

2.1 Preliminaries

2.1.1 Normalization of the x-ranges

The x-ranges are called *normalized* if $D = \{1, \cdots, n'\}$. This has the advantage that we can use the values as array indices. Normalization can always be achieved by identifying each value v_i with its index i in the sorted order $v_1 \leq \cdots \leq v_n$. To do this we need to compute for every original domain $D_j = [v_{l_j}, v_{h_j}]$ the corresponding normalized domain $D'_j = [l_j, h_j]$. After sorting the endpoints of $D_1, \cdots, D_n$ in ascending order, this can be done in time $O(n + n')$. We want to point out that sorting can be done in time $O(n)$, if the v_i's are integers drawn from the range $[1, n^k]$ for some fixed k. After achieving bound consistency for the normalized domains $D'_1, \ldots, D'_n$, we can easily narrow the original domains to bound consistency.

2.1.2 A generalization of matching

A *matching* in a graph is a subset M of its edges such that each node is adjacent to at most one edge in M. We generalize this notion. We consider capacitated graphs $G = (V, E, C)$ where C is a function that maps every node $\nu \in V$ to an interval $C(\nu) = [L_\nu, U_\nu]$. We call $C(\nu)$ the *capacity requirement* of ν. For a set M of edges and a node ν we denote by $M(\nu)$ the set of all nodes that are adjacent to ν by an edge in M. A *generalized matching* in G is a subset M of its edges such that for each node $\nu \in V$ we have $|M(\nu)| \in [L_\nu, U_\nu]$. We call $M(\nu)$ the set of *matching mates of* ν.

As in [16], we define the *variable-value graph* of a GCC, which is in this case a capacitated bipartite graph with n nodes $\{x_1, \cdots, x_n\}$ representing the variables on one side and n' nodes $\{y_1, \cdots, y_{n'}\}$ representing the values in the ranges of the variables on the other side. The capacity ranges of the variable nodes are all $[1, 1]$, indicating that each variable must be assigned exactly one value. For a value node, the capacities are according to the count requirements of the value represented by the node:

$$[L_\nu, U_\nu] = \begin{cases} [L_i, U_i] & \nu = y_i \\ [1, 1] & \nu = x_j \end{cases}$$

The edge (x_j, y_i) exists in the graph if and only if $i \in [\underline{D_j}, \overline{D_j}]$.

Running example: Throughout this chapter, we will illustrate the algorithms with the following example: $GCC(x_1, \cdots, x_6, c_{v_1}, \cdots, c_{v_4})$ where the assignment variable ranges D_j and the count variable ranges E_i are as in Figure 2.1.

E_1	E_2	E_3	E_4
[1,3]	[1,2]	[1,1]	[1,1]

D_1	D_2	D_3	D_4	D_5	D_6
[1,1]	[1,2]	[1,2]	[2,2]	[3,4]	[3,4]

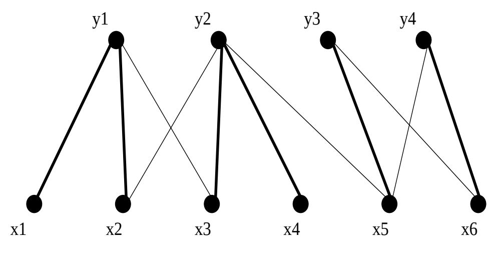


Figure 2.1: The variable-value graph of the example GCC and a generalized matching in it, marked with bold edges.

Lemma 1 *Let T denote the projection of S onto its first n components. Every generalized matching $M = \{\{x_i, y_{g(i)}\} | 1 \leq i \leq n\}$ corresponds to the tuple $(g(1), \cdots, g(n))$ in T and vice versa.*

Proof Immediate from the definition of a generalized matching. ▮

2.1.3 The connection to Régin's algorithm

Régin's algorithm computes a maximum flow in a directed graph $\vec{H}$ (see Figure 2.2) which is similar to our variable-value graph with a few differences: it is directed, contains two additional nodes s and t, and the capacity requirements are on the edges and not on the nodes. The edge set of $\vec{H}$ is defined as follows.[2] There is an edge (x_j, y_i) with capacity requirement $[0, 1]$ iff $i \in D_j$. For every y_i there is an edge (y_i, s) with capacity bounds $[L_i, U_i]$, and for every x_j we have the edge (t, x_j) with capacity requirement $[0, 1]$. Régin shows that the constraint has a solution iff the maximum flow from t to s has value n.

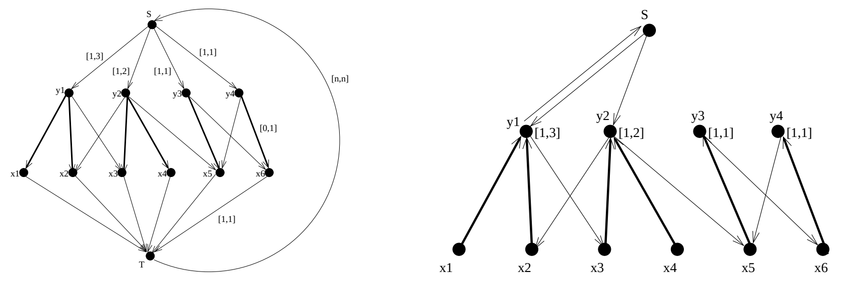


Figure 2.2: The graphs $\vec{H}$ (left) and $\vec{R}_F$ (right) for our running example.

His algorithm searches for a flow F of value n. If none exists, it reports failure. Otherwise, it constructs the residual graph $\vec{R}_F$ (see Figure 2.2), which allows to determine for each variable the arc consistent values in its domain. $\vec{R}_F$ has the same nodes as $\vec{H}$ (except for t) and the following edges: an edge (x_j, y_i) iff $i \in D_j \wedge F(x_j, y_i) = 0$, an edge (y_i, x_j) iff $i \in D_j \wedge F(x_j, y_i) = 1$, an edge (y_i, s) iff $F(y_i, s) < U_i$, and finally an edge (s, y_i) iff $F(y_i, s) > L_i$. In Corollary 2 in [26], Régin proves that a value i in the domain D_j is consistent for x_j iff $F(x_j, y_i) = 1$ or x_j and y_i belong to the same SCC of $\vec{R}_F$.

[2] In fact, the edges in $\vec{H}$ have the opposite direction compared to Régin's graph. We prefer the reversed orientation because it is equivalent and simplifies the following presentation.

We will now translate his construction into the language of generalized matchings in the variable-value graph. Given an variable-value graph G, a generalized matching M corresponds to the following flow F_M in $\vec{H}$. For an edge (x_j, y_i), $F_M(x_j, y_i) = 1$ if $\{x_j, y_i\} \in M$ and $F_M(x_j, y_i) = 0$ otherwise. $F_M(y_i, s) = |M(y_i)|$ for all i, and $F_M(t, x_j) = 1$ for all j. Observe that the total flow from t to s has value n.

We augment the residual graph $\vec{R}$ that corresponds to F_M such that we obtain a graph $\vec{G}$ with the following property: A value i in the domain of a variable x_j is arc consistent iff x_j and y_i are in the same SCC of $\vec{G}$. If x_j is matched with y_i in M, $\vec{R}$ contains the edge (y_i, x_j), and in $\vec{G}$ we add the edge (x_j, y_i) in the opposite direction. So if x_j and y_i are matched (equivalently, $F_M(x_j, y_i) = 1$) then they belong to the same SCC of $\vec{G}$. We call $\vec{G}$ the *oriented variable-value graph.* Note that a similar augmentation appears in [16].

Lemma 2 follows easily from the discussion above and Régin's results cited in it:

Lemma 2 *An edge $\{x_j, y_i\}$ in G belongs to a generalized matching iff x_j and y_i are in the same SCC of $\vec{G}$.*

This lemma implies the correctness of the following bound consistency algorithm for the assignment variables:

1. Find a generalized matching in G.
2. Construct $\vec{G}$ and compute its SCCs.
3. Narrow the ranges as much as possible such that they still represent all edges within the SCCs.

We want to point out that our algorithm uses $O(n + n')$ space and does not construct any of these graphs explicitely.

2.2 Finding a generalized matching in the variable-value graph

The algorithm for finding a generalized matching in a convex graph is given in Figure 2.3. It receives as input: (1) The ranges of assignment variables, $D_j = [\underline{D}_j, \overline{D}_j]$ for each x_j, $1 \leq j \leq n$ (2) A capacity requirement $[L_i, U_i]$ for each value $1 \leq i \leq n'$. It constructs a generalized matching if it exists and returns *failure* otherwise.

The algorithm makes three passes over the y nodes. In the first two passes it goes from left to right and uses a priority queue P to which x nodes are inserted when they become candidates for matching and in which they are sorted according to the upper endpoints of their domains. That is, the node x that is extracted from P by an *ExtractMin* operation is the one whose domain ends earliest. So any node that remains in P can match the same future y-nodes as x (by convexity), but maybe even more. And hence, it is reasonable to extract x and keep the others.

In the first pass the algorithm ignores the lower bound capacities and finds a generalized matching in the graph $G_{L=0} = (V, E, C_{L=0})$, where $C_{L=0}(y_i) = [0, U_i]$ for all i and $C_{L=0}(x_j) = [1, 1]$ for all j. $G_{L=0}$ is the same as G except that the lower bound capacities of all y nodes are zero. This generalized matching is constructed as follows: The y nodes are traversed from $y_1, \cdots, y_{n'}$. When y_i is reached, all of the x nodes that are connected to y_i, but not to any node with a smaller index, are inserted into the queue; they are now candidates for matching. Then y_i is matched with up to U_i nodes from P (less if P does not contain that many nodes). If, while processing y_i, the algorithm extracts a node from the queue which is not connected to y_i, it reports failure (cf. Lemma 3).

In the second pass, the algorithm makes another traversal of the y nodes in the same order, but this time it ignores the upper bounds and constructs a generalized matching in the graph $G^{U=\infty} = (V, E, C^{U=\infty})$, where $C^{U=\infty}(y_i) = [L_i, \infty]$ for all i and $C^{U=\infty}(x_j) = [1, 1]$ for all j. In this matching, each y node is matched with the minimal number of x nodes such that its lower capacity bound is respected and all x nodes which are not connected to y nodes with higher indices are matched. If the queue becomes empty while the algorithm tries to fulfil the lower capacity bound L_i for node y_i, the algorithm reports failure (see Lemma 6).

The matching found in the first pass is used during the second pass to determine the order in which nodes are inserted into the queue. We will show that if a generalized matching exists, then there also exists a generalized matching such that each y_i is matched only with nodes that were matched in the first pass with nodes from $y_1, \cdots, y_i$ (cf. Lemma 4). Hence, when y_i is reached in the second pass, the nodes that were matched with it in the first pass are inserted into the queue as candidates for matching with it.

The matching found in the second pass may violate the upper capacity bounds in G. The third pass corrects this by traversing the y nodes from $y_{n'}$ to y_1 and shifting these excesses to y nodes with lower indices.

For our running example, we show in the table below the mate of x_j after the respective pass of the algorithm.

j	1	2	3	4	5	6
pass 1	1	1	1	2	2	3
pass 2	1	2	2	2	3	4
pass 3	1	1	2	2	3	4

Lemma 3 *If the algorithm reports failure in the first pass, then there is no generalized matching in $G_{L=0}$. And hence, there is also none in G.*

Proof Suppose the algorithm reports failure in iteration i after extracting a node x_j (observe that any extraction in iteration $n'+1$ causes failure.) Then in iteration $i-1$ it only extracts x nodes that are not connected to y_i, and P contains x_j afterwards. Let i' be the maximum iteration before $i-1$ such that an x node connected to y_i is extracted or P becomes empty during iteration i' (see Figure 2.4). If no such iteration exists, choose $i'=0$. Let X denote the x nodes extracted in iterations $i'+1,\cdots,i-1$. Since P never becomes empty during these iterations, we have $|X|=\sum_{k=i'+1}^{i-1} U_k$, i.e. X exhausts the capacities of the nodes in $Y=\{y_{i'+1},\cdots,y_{i-1}\}$.

The nodes in X are not connected to y_i or a y node to its right. We show that there is also no connection to $y_{i'}$ or a y node to its left. Suppose otherwise, i.e. $x_\ell \in X$ is connected to $y_{i''}$ with $i'' \leq i'$. As x_ℓ was inserted to P in iteration i' or earlier and is extracted in a later iteration, P was never empty in iteration i'. By the choice of i', this implies that a node $x_{j'}$ connected to y_i was extracted in iteration i'. As x_l was removed later, it must also be connected to y_i, a contradiction to the choice of i'. Thus all neighbors of the nodes in X are in Y. A similar argument proves that the node x_j, which caused the failure in iteration i, is only connected to nodes in Y.

Since a generalized matching in $G^{U=\infty}$ would have to match the nodes in Y with at least $|X|+1$ nodes, but can only match them with $|X|$ different nodes, such a matching cannot exist. ∎

```
(* 1st pass: find a matching in $G_{L=0}$ (encoded in $m_1$) *)
$P \leftarrow []$ (* priority queue containing $x$ nodes sorted according to $\overline{D}$ *)
$j \leftarrow 0$; $U_{n'+1} \leftarrow \infty$
for $i = 1$ to $n' + 1$ do
    $u \leftarrow 0$; forall $x_h$ with $\underline{D}_h = i$ do $P.Insert\ x_h$
    while $P$ is not empty and $u < U_i$ do
        $j \leftarrow j + 1$; $x_{f(j)} \leftarrow P.ExtractMin$
        $m_1[f(j)] \leftarrow i$; $u \leftarrow u + 1$
        if $\overline{D}_{f(j)} < i$ then report failure (* No failure implies $i \in D_{f(j)}$ *)
    end
    $\beta_i \leftarrow j$
endfor
(* 2nd pass: find a matching in $G^{U=\infty}$ and ensure feasibility for $G$ *)
$j \leftarrow 0$; $\alpha_0 \leftarrow 0$
for $i = 1$ to $n'$ do
    forall $x_h$ with $m_1[h] = i$ do $P.Insert\ x_h$
    for $\ell = 1$ to $L_i$ do
        if $P$ is empty then report failure
        $j \leftarrow j + 1$; $x_{g(j)} \leftarrow P.ExtractMin$
        match $y_i \leftrightarrow x_{g(j)}$    (* $m_1[g(j)] \leq i$ *)
    endfor
    while $P$ is not empty and $P.MinPriority < i + 1$ do
        $j \leftarrow j + 1$; $x_{g(j)} \leftarrow P.ExtractMin$
        match $y_i \leftrightarrow x_{g(j)}$    (* $m_1[g(j)] \leq i$ *)
    end
    $\alpha_i \leftarrow j$
endfor
(* 3rd pass: transform the matching from 2nd pass into a matching in $G$ *)
for $i = n'$ to 1 do
    if $y_i$ has currently $h$ mates s.th. $e := h - U_i > 0$ then
        for $k = 1$ to $e$ do
            choose a current mate $x_{j_k}$ of $y_i$ with $i' := m_1[j_k] < i$
            match $y_{i'} \leftrightarrow x_{j_k}$    (* $y_i$ looses $x_{j_k}$ *)
        endfor
    endif
endfor
```

Figure 2.3: Algorithm for generalized matching in a convex capacitated bipartite graph

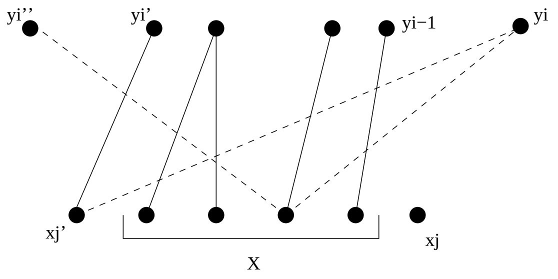


Figure 2.4: Illustration of the proof of Lemma 3

For any matching M and subset $Y \subseteq \{y_1, \cdots, y_{n'}\}$, let $M(Y) = \bigcup_{y \in Y} M(y)$, i.e. $M(Y)$ is the set of all x nodes that are matched with nodes of Y.

Lemma 4 *If there is a generalized matching M' in G, then there is also a generalized matching M with the following property: For all edges $\{x_j, y_i\} \in M$, we have $m_1[j] \leq i$.*
M can be chosen such that $|M(y_i)| = |M'(y_i)|$ for $i = 1, \cdots, n'$. And hence, it is impossible to match $\{y_1, \cdots, y_i\}$ with more than β_i nodes[3].

Proof Let M' be a generalized matching in G, and suppose it does not have the desired property. Let i be minimal such that there is an edge $\{x_j, y_i\}$ with $k = m_1[j] > i$. Not all nodes x_ℓ with $m_1[\ell] = i$ can be matched with y_i in M', otherwise neither the queue nor the capacity of y_i would have been exhausted after iteration i in the first pass. So there is an edge $\{x_{j'}, y_{i'}\} \in M'$ with $i' \neq i$ and $m_1[j'] = i$ (see Figure 4). By the choice of i we have $i' > i$. When $x_{j'}$ was extracted in the first pass, x_j was also in the queue. So we conclude that x_j is also connected to $y_{i'}$. Thus we can swap the mates of x_j and $x_{j'}$ and obtain a new generalized matching M''. M'' does not violate the property for the nodes $y_1, \cdots, y_{i-1}$ and the number of violations at y_i is less than in M'. This shows that we can transform the matching until we eventually obtain a matching M with the desired property.
The last statement follows from the fact that $\beta_i = |\{x_j \mid m_1[j] \leq i\}|$. ∎

Lemma 5 *If the algorithm does not report failure in iteration i of the second pass, then for any generalized matching M in G, $|M(\{y_1, \cdots, y_i\})| \geq \alpha_i$.*

Proof By induction on i. For $i = 0$ and $i = 1$ the claim is easy to verify. So let us assume that it holds for $i' = 1, \cdots, i-1$ and prove it for i. If the body of the while-loop is not exe-

[3]We will refer to the β_i values in Section 2.5.

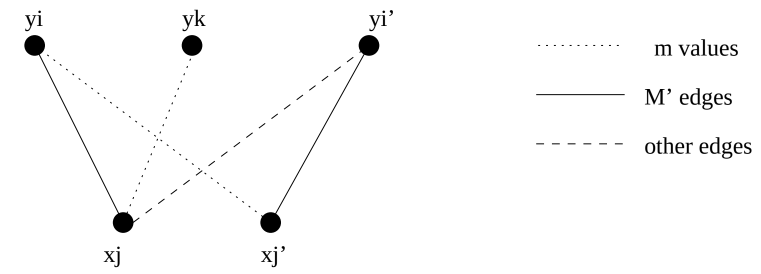


Figure 2.5: Illustration of the proof of Lemma 4

cuted, we have $\alpha_i = \alpha_{i-1} + L_i$, and applying the induction for $i' = i - 1$ immediately proves the claim. So suppose that nodes are extracted in the while-loop, which implies that no x that is extracted in iteration i is connected to y_{i+1}. Let i' be the maximum iteration before i such that a node connected to y_{i+1} is removed. If no such iteration exists, choose $i' = 0$. Let X denote the x nodes extracted in the iterations $i' + 1, \ldots, i$, and let $Y = \{y_{i'+1}, \cdots, y_i\}$. A similar argument as in the proof of Lemma 3 shows that the neighbors of any node $x \in X$ are contained in Y. Thus $|M(Y)| \geq |X| = \alpha_i - \alpha_{i'}$. By the induction hypothesis for i', $|M(\{y_1, \cdots, y_i\})| \geq \alpha_{i'} + |M(Y)| = \alpha_i$. ▮

Lemma 6 *If the algorithm reports failure in the second pass, then there is no generalized matching in G.*

Proof Suppose that the algorithm reports failure in iteration i, although a generalized matching M exists. By Lemma 4, we can assume that for any edge $\{x_j, y_k\}$ in M we have $m_1[j] \leq k$. So $y_1, \cdots, y_i$ are matched only with nodes that have been inserted into P so far. From the previous lemma we can conclude that there must be at least $\alpha_{i-1} + L_i$ such nodes. But since the algorithm reports failure, this is not the case, a contradiction to the existence of M. ▮

Lemma 7 *If the algorithm does not report failure, it constructs a generalized matching in G.*

Proof If the second pass succeeds, we have a generalized matching in $G^{U=\infty}$, because we fulfil the lower capacity bounds of every y node (cf. the for-loop), and we match every x

node with a neighbor on the y side (see the while-loop). So the only problem is the upper capacity bounds of the y nodes. The third pass takes care of these. Observe that this pass sweeps over the y nodes from right to left and distributes the excess mates of a node y_i only to y nodes with lower indices, so that it cannot increase the number of mates of a y node after it was processed. Furthermore, if x_j is matched with y_i at some point in time then $m_1[j] \leq i$. Since we have equality for at most U_i nodes, we can always select e excessive mates to distribute, if necessary. Suppose a node x_j is removed from y_i and matched with $y_{i'}$ in the third pass, where $i' = m_1[j]$. From the first pass, it is easy to see that $i' \in D_j$, i.e., x_j and $y_{i'}$ are connected. ∎

Implementing the algorithm in linear time (for normalized domains)

Now we discuss some implementation details of the algorithm. First we show how to implement it in time $O(n' + n \log n)$, and then we refine this implementation to obtain $O(n' + n)$ time. In the first variant we use a binary heap of size n to implement the priority queue P. Then the operations *Insert* and *ExtractMin* take time $O(\log n)$ and *MinPriority* runs in constant time. Before we run the algorithm we sort the x nodes according to their lower range endpoints, which takes time $O(n + n')$ because all domains are in $[1, n']$. This sorting allows us to determine efficiently the nodes that have to be inserted into P in each iteration of the first pass. Recall that the order by which the x nodes are extracted in the first pass determines the order by which they are inserted in the second pass. So the two passes can be implemented in time $O(n' + n \log n)$. The third pass takes linear time. To see this we notice that every x node changes its mate at most once. So we maintain for every y node two separate lists for the mates that it received in the second and the third pass respectively. Thus we can make sure that we process every x node only once.

In order to shave off the logarithmic factor in the running time, we have to find a faster implementation of the priority queue. As in [16] we simulate the priority queue by creating an instance of the offline-min problem [1, Chapter 4.8], which can be solved in linear time using a special union-find data structure [9]. (The offline-min algorithm needs a sorting of the x's according to their upper interval endpoints, which can be computed in time $O(n + n')$.)

Except for failure detection, the algorithm does not use the mapping f in the first pass at all. So the whole sequence σ of *Insert* and *ExtractMin* operations can be computed without knowing the results of the extractions. Checking if P is empty can be done by counting the number of insertions and comparing it with the number of extractions (which is equal to j). Failure detection can be done after the mapping f has been computed.

We give a brief description of the offline-min algorithm. When it is applied to a sequence σ it determines for every insertion the corresponding extraction. Let $E_1, \cdots, E_n$ be the extract operations in the order in which they occur in σ. The node x_j with minimum priority is the output of the first extraction E_k that follows its insertion. After deleting E_k from σ we process the x with next higher priority, and so on. We implement this with a union-find structure that maintains a partition $\mathcal{P}$ of $E_1, \cdots, E_n$. Every set in $\mathcal{P}$ has the form $\{E_h, E_{h+1}, \cdots, E_k\}$, where all extractions except for E_k have been deleted. Suppose we process x_j and let E_s be the first extraction in the original sequence after the insertion of x_j. Observe that E_s may have been deleted already. To determine the extraction E_k for x_j, we simply have to find the set S in $\mathcal{P}$ containing E_s. Deleting E_k amounts to uniting S with the set of E_{k+1}.

We cannot use the offline-min algorithm directly for the second pass, because we need to know the result of the *MinPriority* operation in the while-loop online. But a slight enhancement will do the job (see Figure 2.6). We initialize our data structures as above but we only create a set for the mandatory extractions that are made in the for-loop. When we find a tentative extraction E_k for x_j, we verify that E_k occurs in iteration $i \leq \overline{D}_j$. If not, then we know that x_j is extracted by the while-loop in iteration $\overline{D}_j$, and we do not delete E_k. We want to point out that we detect failure if there is a mandatory extraction for which no corresponding insertion is found. Assuming w.l.o.g. that $\sum_{i=1}^{n'} L_i \leq n$, the algorithm runs in time $O(n' + n)$.

2.3 Finding the SCCs of the oriented variable-value graph

As described above, we construct an oriented graph $\vec{G}$ from the variable-value graph and the generalized matching M (see Figure 2.7). The edges which are not within a strongly connected component of $\vec{G}$ describe inconsistent assignments of values to variables. Mehlhorn and Thiel [16] gave an algorithm that finds the SCCs of a simpler oriented variable-value

```
$k \leftarrow 1$(* $E_k$ will always be the next mandatory extraction *)
for $i = 1$ to $n'$ do
    forall $x_j$ with $m_1[j] = i$ do
        (* $x$'s inserted in iteration $i$ (before $E_k$) *)
        $s[j] \leftarrow k$
    endfor for $\ell = 1$ to $L_i$ do
        create set $\{E_k\}$ labelled $\mathcal{L}_k^{(i,\ell)}$ (* for mandatory extraction $\ell$ in iter. $i$ *)
        $k \leftarrow k + 1$
    endfor
endfor
create set $\{E_k\}$ labelled with $\mathcal{L}_k^{(n'+1,0)}$ (* dummy extraction *)
forall $x_j$ sorted in ascending order according to $\overline{D}$ do
    $S \leftarrow$ find($E_{s[j]}$); let $\mathcal{L}_k^{(i,\ell)}$ be its label
    if $i \leq \overline{D}_j$ then
        $x_j$ is removed by extraction $\ell$ in the for-loop of iteration $i$
        unite $S$ with the set $S'$ containing $E_{k+1}$ and
            label the union with the former label of $S'$
    else
        $x_j$ is extracted by the while-loop in iteration $\overline{D}_j$
    endif
endfor
```

Figure 2.6: Enhanced offline-min algorithm for the second pass

graph in $O(n + n')$ time. Their graph is simpler than $\vec{G}$ in that it does not contain the additional node s, which violates the convexity property of the graph because there is an edge from s to each node in $\{y_i : |M(y_i)| > L_i\}$, and this set is not necessarily consecutive. We first use their algorithm to compute the SCCs of the graph $\vec{G}\backslash s$. We can do this despite the fact that now a y node may be matched with more than one x node by merging its neighbors into one x node (note that the neighbors of this merged x node still form an interval of the y-nodes). Let C_s denote the SCC of the node s in $\vec{G}$. An SCC of G which is different from C_s is also an SCC of $\vec{G}\backslash s$. C_s, however, may be composed of zero or more SCCs of $\vec{G}\backslash s$. We wish to find these SCCs and merge them to obtain C_s. Each of these SCCs has the property that it can reach s and can be reached from s. For each SCC C of $\vec{G}\backslash s$ we compute two flags, *reached_from_s*$[C]$ and *reaches_s*$[C]$, and merge all components for which both flags are set to true into C_s.

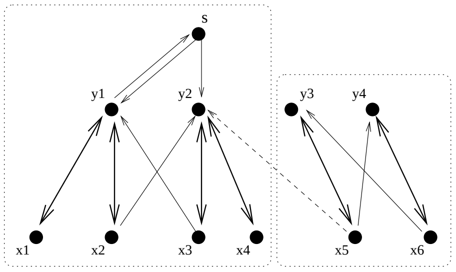


Figure 2.7: The oriented variable-value graph for our running example. It has two SCCs: $\{s, x_1, x_2, x_3, x_4, y_1, y_2\}$ and $\{x_5, x_6, y_3, y_4\}$. The dashed edge is inconsistent.

We will use the following notation. For each SCC C of $\vec{G}\backslash s$, let $\mathit{min\_y}[C] = \min\{i|y_i \in C\}$ and $\mathit{max\_y}[C] = \max\{i|y_i \in C\}$ be the lowest and highest indices of y nodes in C. Moreover, let $\mathit{reaches\_left}[C] = \min\{\underline{D}_j|x_j \in C\}$ and $\mathit{reaches\_right}[C] = \max\{\overline{D}_j|x_j \in C\}$ be the minimum and maximum indices of y nodes that can be reached from C.

The following lemma states that the y nodes reachable by an edge from C form a contiguous interval.

Lemma 8 *C can reach every y_i with $\mathit{reaches\_left}[C] \leq i \leq \mathit{reaches\_right}[C]$ by one edge from an x node in C.*

Proof

Case 1: $\mathit{reaches\_left}[C] \leq i \leq \mathit{min\_y}[C]$. Let $x_\ell \in C$ be such that $\underline{D}_\ell = \mathit{reaches\_left}[C]$ and let y_k be the node matched with x_ℓ. Then $\underline{D}_\ell \leq i \leq k$ and by convexity $i \in D_\ell$.

Case 2: $\mathit{max\_y}[C] \leq i \leq \mathit{reaches\_right}[C]$ is symmetric to Case 1.

Case 3: $\mathit{min\_y}[C] < i < \mathit{max\_y}[C]$. Since C is strongly connected, there is a path $(y_{i_1}, x_{j_1}) \circ \cdots \circ (x_{j_{k-1}}, y_{i_k})$ in C where $i_1 = \mathit{min\_y}[C]$ and $i_k = \mathit{max\_y}[C]$. Then there exists $1 \leq \kappa < k$ such that $i_\kappa < i \leq i_{\kappa+1}$. This implies that $\underline{D}_{j_\kappa} \leq i_\kappa < i \leq i_{\kappa+1} \leq \overline{D}_{j_\kappa}$, and by convexity $i \in D_{j_\kappa}$, i.e., y_i is a neighbor of x_{j_κ}. ∎

Now we will explain how to determine whether s can reach a component C. If this is the case then there is a path Q from s to some node y_i in C. And hence, if we delete the first edge of Q, we obtain a path P from a node $y_{i'}$ to y_i in $\vec{G}\backslash s$. In the lemma below, we state that this implies the existence of path P' from $y_{i'}$ to y_i that traverses the SCCs of $\vec{G}\backslash s$ in monotonous manner, either from left to right (cf. Condition (1)) or from right to left (cf. Condition (2)):

Lemma 9 *If there is a path from y_{i_1} to y_{i_k} in $\vec{G}\backslash s$ then there is a path $P = (y_{i_1}, x_{j_1}) \circ (x_{j_1}, y_{i_2}) \circ \cdots \circ (x_{j_{k-1}}, y_{i_k})$ in $\vec{G}\backslash s$ such that if we let C_κ denote the component of y_{i_κ} for each $1 \leq \kappa \leq k$, then one of the following holds:*

$$\forall_{1\leq\kappa<k}\ \mathit{min\_y}[C_\kappa] \leq \mathit{min\_y}[C_{\kappa+1}] \quad \text{right-monotony} \tag{2.1}$$

$$\forall_{1\leq\kappa<k}\ \mathit{max\_y}[C_\kappa] \geq \mathit{max\_y}[C_{\kappa+1}] \quad \text{left-monotony} \tag{2.2}$$

Proof If P visits more than one component, let κ be the smallest value such that $C_\kappa \neq C_{\kappa+1}$. Assume that Condition (1) holds for κ. If P does not fulfil Condition (1), then there must be three components C_1, C_2, C_3 which are visited consecutively by P in that order such that $\mathit{min\_y}[C_1] < \mathit{min\_y}[C_2]$ and $\mathit{min\_y}[C_3] < \mathit{min\_y}[C_2]$. We will show that there is a path which skips C_2 and goes directly from C_1 to C_3.
If $\mathit{min\_y}[C_3] < \mathit{min\_y}[C_1] < \mathit{min\_y}[C_2]$ then by Lemma 8, C_2 can reach C_1 by a single edge, which closes a cycle that visits both of them. This means that $C_1 = C_2$, in contradiction to the assumption that they are distinct.
We therefore know that $\mathit{min\_y}[C_1] < \mathit{min\_y}[C_3] < \mathit{min\_y}[C_2]$. Then by Lemma 8, C_1 can reach C_3 by one edge. So we can construct a path from y_{i_1} to y_{i_k} which visits the same components as P except that it goes from C_1 directly to C_3 without going through C_2. Since this decreases the number of transitions between components along the path, we can obtain a monotonous path by a finite number of such modifications.

If Condition (1) does not hold for the first transition in P between two distinct components $C_\kappa, C_{\kappa+1}$ then we show that Condition (2) must hold and then it is clear that we can similarly find a path that fulfils Condition (2) for all $1 \leq \kappa < k$.

Assume that neither condition holds. Then $\mathit{min\_y}[C_{\kappa+1}] < \mathit{min\_y}[C_\kappa]$ and $\mathit{max\_y}[C_\kappa] < \mathit{max\_y}[C_{\kappa+1}]$. By Lemma 8, $C_{\kappa+1}$ reaches back to C_κ, a contradiction. ∎

The algorithm in Figure 2.8 performs a left to right scan to mark all the components of $\vec{G}\backslash s$ that can be reached from s via a right-monotonous path. In the initialization phase it marks all components that can be reached from s by a single edge. During the scan it maintains the index *max_y_reached_from_s* which describes the rightmost node that can be

```
forall SCCs $C$ of $\vec{G}\backslash s$ do
    $reached\_from\_s[C] \leftarrow false$
end
for $i = 1$ to $n'$ do
    if $(s, y_i)$ exists then $reached\_from\_s[C[y_i]] \leftarrow$ true
endfor
(* Scan from left to right and find the maximum $y$'s reached from $s$ *)
$max\_y\_reached\_from\_s \leftarrow 0$
for $i = 1$ to $n'$ do
    $C \leftarrow C[y_i]$
    (* Check whether $C$ can be reached from $s$ via a right-monotonous path *)
    if $max\_y\_reached\_from\_s \geq i$ then
        $reached\_from\_s[C] \leftarrow$ true
    endif
    (* Advance $max\_y\_reached\_from\_s$ if possible *)
    if $reached\_from\_s[C]$ then
        $max\_y\_reached\_from\_s \leftarrow \max\{max\_y\_reached\_from\_s, reaches\_right[C]\}$
    endif
endfor
```

Figure 2.8: Algorithm for finding SCCs that can be reached from s via a right-monotonous path

reached from a scanned marked component by one edge. The running time is $O(n')$. A symmetric algorithm can be used to find left-monotonous paths.

Lemma 10 *The algorithm in Figure 2.8 sets reached_from_s$[C]$ to true iff s can reach C by a right-monotonous path.*

Proof It is easy to see that if a component becomes marked, then it can be reached from s. We will now prove the converse. Consider a path $P = (s, y_{i_1}) \circ \cdots \circ (y_{i_{k-1}}, x_{j_{k-1}}) \circ (x_{j_{k-1}}, y_{i_k})$ such that P without its first edge is right-monotonous. We will show the following claim by induction on the length of P: after the i_k-th iteration, the component C_k of y_{i_k} is marked "reached from s".

For $|P| = 1$: $P = (s, y_{i_1})$ and the component of y_{i_1} is marked before the scan.

For $|P| > 1$: By the induction hypothesis, *reached_from_s*$[C_{k-1}]$ is set to true after iteration $min\_y[C_{k-1}]$. [4] Since $x_{j_{k-1}}$ is in C_{k-1}, $reaches\_right[C_{k-1}] \geq i_k \geq min\_y[C_k]$. In iteration $min\_y[C_{k-1}]$, $max\_y\_reached\_from\_s$ is set to at least $min\_y[C_k]$. By Lemma 9, we know that iteration $min\_y[C_k]$ does not preceed iteration $min\_y[C_{k-1}]$. Since the index $max\_y\_reached\_from\_s$ can only increase, we get that in iteration $min\_y[C_k]$, the component of y_{i_k} will be marked as reached from s. ▮

In order to determine whether a component C can reach s, we can use the same approach: if we delete the last edge of a path in $\vec{G}$ from a node in C to s, we obtain a path in $\vec{G} \backslash s$. So we can mark all components that can reach s by two scans in time $O(n')$. The complete algorithm is given in Figure 2.9. Finally, we merge all components that can reach s and can be reached from s into the single component C_s.

2.4 Narrowing the Bounds of the Assignment Variables

Let S denote the set of all solutions of the constraint, which is defined above, and for $j = 1, \ldots, n$ let S_j be the projection of S onto the jth component. We will discuss how to compute the values $\overline{S}_1, \ldots, \overline{S}_n$ in time $O(n + n')$, a symmetric procedure can be used for the lower endpoints of the narrowed ranges.

Consider a node x_j and let C be its SCC in $\vec{G}$. By Lemma 2, a value $i \in D_j$ is contained in S_j iff $y_i \in C$. Thus $\overline{S}_j$ is the index of the rightmost y node in C that is connected to x_j. Suppose the y nodes $y_{i_1}, \ldots, y_{i_k}$ in C are sorted such that $i_1 < \cdots < i_k$. Then $\overline{S}_j = i_\kappa$ where $i_\kappa \leq \overline{D}_j < i_{\kappa+1}$ (and $i_{k+1} = n' + 1$). So let us further assume that the x nodes $x_{j_1}, \ldots, x_{j_l}$ in C are sorted such that $\overline{D}_{j_1} \leq \cdots \leq \overline{D}_{j_l}$. Then we can determine $\overline{S}_j$ for every x_j in C by merging the sorted sequences $(i_1, \ldots, i_k)$ and $(\overline{D}_{j_1}, \ldots, \overline{D}_{j_l})$ in time $O(k + l)$.

We need to say how these sorted sequences are constructed. For the y's this can be done with bucketsort. We have a bucket for each component, and for $i = 1, \ldots, n'$, we append y_i to the bucket corresponding to its component. This takes time $O(n')$ and constructs the y-sequences for all SCCs of $\vec{G}$. Since we have a global sorting of the x's according to their upper range endpoints, we can use the same approach to sort them in time $O(n)$.

For the first SCC of our example we would merge the y sequence $(1, 2)$ with the x upper

[4] We may assume w.l.o.g. that if P visits a component C then it also visits the node $y_{min\_y[C]}$.

```
$SCCs \leftarrow$ Find_SCCs($\vec{G} \setminus s$)
**forall** $C \in SCCs$ **do**
    $reached\_from\_s[C] \leftarrow false$
    $reaches\_s[C] \leftarrow false$
**end**
**for** $i = 1$ to $n'$ **do**
    **if** $(s, y_i)$ exists **then** $reached\_from\_s[C[y_i]] \leftarrow$ true
    **if** $(y_i, s)$ exists **then** $reaches\_s[C[y_i]] \leftarrow$ true
**end**
(* Scan from start to end and find the maximum $y$'s connected to $s$ *)
$max\_y\_reached\_from\_s \leftarrow 0$; $max\_y\_reaches\_s \leftarrow 0$
**for** $i = 1$ to $n'$ **do**
    $C \leftarrow C[y_i]$
    (* Check whether $C$ can be reached from $s$ *)
    **if** $max\_y\_reached\_from\_s \geq i$ **then** $reached\_from\_s[C] \leftarrow$ true
    **if** $reached\_from\_s[C]$ **then**
            $max\_y\_reached\_from\_s \leftarrow \max\{max\_y\_reached\_from\_s, reaches\_right[C]\}$
    (* Check whether $C$ can reach $s$ *)
    **if** $reaches\_left[C] \leq max\_y\_reaches\_s$ **then** $reaches\_s[C] \leftarrow$ true
    **if** $reaches\_s[C]$ **then** $max\_y\_reaches\_s \leftarrow \max\{max\_y\_reaches\_s, i\}$
**end for**
(* Scan from end to start and find the minimum $y$'s connected to $s$ *)
$min\_y\_reached\_from\_s \leftarrow n' + 1$; $min\_y\_reaches\_s \leftarrow n' + 1$
**for** $i = n'$ to 1 **do**
    $C \leftarrow C[y_i]$
    (* Check whether $C$ can be reached from $s$ *)
    **if** $min\_y\_reached\_from\_s \leq i$ **then** $reached\_from\_s[C] \leftarrow$ true
    **if** $reached\_from\_s[C]$ **then**
            $min\_y\_reached\_from\_s \leftarrow \min\{min\_y\_reached\_from\_s, reaches\_left[C]\}$
    (* Check whether $C$ can reach $s$ *)
    **if** $reaches\_right[C] \geq min\_y\_reaches\_s$ **then** $reaches\_s[C] \leftarrow$ true
    **if** $reaches\_s[C]$ **then** $min\_y\_reaches\_s \leftarrow \min\{min\_y\_reaches\_s, i\}$
**end for**
(* Merge all components that are strongly connected through $s$ *)
$C \leftarrow \{s\}$
**foreach** $C' \in SCCs$ **do** **if** $reaches\_s[C'] \wedge reached\_from\_s[C']\}$ **then** $C \leftarrow C \circ C'$
```

Figure 2.9: Algorithm for finding SCCs of the oriented variable-value graph

bounds $(2_{x_1}, 3_{x_2}, 3_{x_3})$ and narrow the domains of x_2 and x_3 from $[1,3]$ to $[1,2]$.

2.5 Narrowing the Bounds of the Count Variables

This section and the next deal with the projections of S onto its components $S_{n+1}, \cdots, S_{n+n'}$. We show how to compute lower and upper bounds for the values in S_{n+i} for each $i = 1, \cdots, n'$. In this section we show a very fast algorithm that narrows the domains of the count variables but does not achieve bound consistency. In the next section we show a different algorithm that achieves bound consistency for the count variables. It has the same linear-time asymptotic complexity but uses more elaborate data structures that can be expected to make it slower in practice.

For the rest of this section we consider a fixed graph G. We assume that the generalized matching algorithm has computed the α_i and β_i values for G and terminated without reporting failure. Now we show that in any generalized matching, the number of x nodes matched to each y_i are in a certain range that we can compute in linear time.

Lemma 11 *Let M be a generalized matching and for $i = 1, \cdots, n'$ let $\mu_i = |M(\{y_i\})|$. Then for all i*

$$\max\left(L_i, n - \beta_{i-1} - \sum_{j=i+1}^{n'} \mu_j\right) \le \mu_i \le \min\left(U_i, n - \alpha_{i-1} - \sum_{j=i+1}^{n'} \mu_j\right) \quad (*)$$

Proof By the choice of the μ's we have $|M(\{y_1, \cdots, y_{i-1}\})| = n - \sum_{j=i}^{n'} \mu_j$. By Lemmas 4, 5 we know that $\alpha_{i-1} \le |M(\{y_1, \cdots, y_{i-1}\})| \le \beta_{i-1}$.
Therefore $\alpha_{i-1} \le n - \sum_{j=i}^{n'} \mu_j \le \beta_{i-1}$ or $n - \beta_{i-1} \le \sum_{j=i}^{n'} \mu_j \le n - \alpha_{i-1}$.
And hence $n - \beta_{i-1} - \sum_{j=i+1}^{n'} \mu_j \le \mu_i \le n - \alpha_{i-1} - \sum_{j=i+1}^{n'} \mu_j$. ∎

This motivates the following definition. We call a sequence $(\mu_s, \cdots, \mu_{n'})$ a *legal count choice* iff inequality (*) is satisfied for all i in $[s, n']$. The Lemma above implies that any generalized matching M in G induces the legal choice $(|M(y_1)|, \ldots, |M(y_{n'})|)$. This allows us to compute for any y_i a lower bound l_i and an upper bound u_i on $|M(y_i)|$. Unfortunately there exist examples where the bounds are not tight. Choosing $D_1 = [1,3], D_2 = [2,2]$ and $E_1 = E_2 = E_3 = [0,1]$ is such an example. The lower endpoint of E_2 will not be narrowed to 1 by our algorithm.

Apart from these bounds the algorithm below determines two count choices. The algorithm maintains the invariant that in iteration i both $(\kappa_i, \ldots, \kappa_{n'})$ and $(\lambda_i, \ldots, \lambda_{n'})$ are legal count choices such that for any legal count choice $(\mu_i, \ldots, \mu_{n'})$ we have $\sum_{j=i}^{n'} \kappa_j \leq \sum_{j=i}^{n'} \mu_j \leq \sum_{j=i}^{n'} \lambda_j$. As all sums that appear in the algorithm can be computed incrementally, the running time is $O(n')$.

```
for $i = n'$ to 1 do
    $l_i \leftarrow \max\left(L_i, n - \beta_{i-1} - \sum_{j=i+1}^{n'} \lambda_j\right)$; $u_i \leftarrow \min\left(U_i, n - \alpha_{i-1} - \sum_{j=i+1}^{n'} \kappa_j\right)$
    $\kappa_i \leftarrow \max\left(L_i, n - \beta_{i-1} - \sum_{j=i+1}^{n'} \kappa_j\right)$; $\lambda_i \leftarrow \min\left(U_i, n - \alpha_{i-1} - \sum_{j=i+1}^{n'} \lambda_j\right)$
endfor
```

We will now prove by induction on i that the algorithm computes the correct bounds for the count variables and that the invariant holds. For $i = n'$ Lemma 11 implies that $l_{n'}$ and $u_{n'}$ are lower and upper bounds. Since all sums are empty, we have $\kappa_{n'} = l_{n'}$ and $\lambda_{n'} = u_{n'}$, and hence, the claimed invariant holds.

We come to the induction step, we assume that our claim holds for $i + 1$ and verify it for i. Looking at the left-hand side of inequality (*), we see that the minimum legal value for μ_i is obtained if $\sum_{j=i+1}^{n'} \mu_j$ is set to its largest possible value. Applying the invariant for $i + 1$ this sum is maximized by the legal choice $(\lambda_{i+1}, \ldots, \lambda_{n'})$. So by Lemma 11, the algorithm computes a lower bound l_i. A similar argument holds for the upper bound u_i. By the definition of a legal choice, extending our two count choices by κ_i and λ_i respectively yields two legal choices again.

Fix a legal choice $(\mu_i, \ldots, \mu_{n'})$. What remains to prove are the two inequalities on the sum of the μ's.

$$\begin{aligned}
\sum_{j=i}^{n'} \mu_j &= \mu_i + \sum_{j=i+1}^{n'} \mu_j \overset{\text{L 11}}{\geq} \max(L_i, n - \beta_{i-1} - \sum_{j=i+1}^{n'} \mu_j) + \sum_{j=i+1}^{n'} \mu_j \\
&= \max(L_i + \sum_{j=i+1}^{n'} \mu_j, n - \beta_{i-1}) \\
&\overset{\text{IH}}{\geq} \max(L_i + \sum_{j=i+1}^{n'} \kappa_j, n - \beta_{i-1}) \\
&= \max(L_i, n - \beta_{i-1} - \sum_{j=i+1}^{n'} \kappa_j) + \sum_{j=i+1}^{n'} \kappa_j \\
&= \kappa_i + \sum_{j=i+1}^{n'} \kappa_j = \sum_{j=i}^{n'} \kappa_j
\end{aligned}$$

An analogous computation shows $\sum_{j=i}^{n'} \mu_j \leq \sum_{j=i}^{n'} \lambda_j$. So we have just shown the following lemma:

Lemma 12 *Suppose that $S \neq \emptyset$ and let $\ell_1, \cdots, \ell_{n'}$ and $u_1, \cdots, u_{n'}$ be the values computed by our algorithm. Then $\underline{S}_{n+i} \geq \ell_i$ and $\overline{S}_{n+i} \leq u_i$ holds for $i = 1, \cdots, n$.*

If we apply the algorithm to our running example, it computes the values listed below:

i	l_i	u_i	κ_i	λ_i	$\sum_{j=i}^{n'} \kappa_i$	$\sum_{j=i}^{n'} \lambda_i$
4	1	1	1	1	1	1
3	1	1	1	1	2	2
2	1	2	1	2	3	4
1	2	3	3	2	6	6

E_1, then, is narrowed from $[1, 3]$ to $[2, 3]$ and $E_2, \ldots, E_4$ remain as they were.

2.6 Bound Consistency for the Count Variables

In this section we show a linear-time algorithm that achieves bounds consistency for the count variables. The following lemma extends the result of Lemma 4.

Lemma 13 *Let $m_2[j]$ be the index of the y node that x_j was matched with in the second pass of the algorithm. If there is a generalized matching M' in G, then there is also a generalized matching M with the following property: For all edges $\{x_j, y_i\} \in M$, we have $m_1[j] \leq i \leq m_2[j]$ and for all $1 \leq i \leq n'$, $|M(y_i)| = |M'(y_i)|$.*

Proof By Lemma 4, we can assume that for all edges $\{x_j, y_i\} \in M'$, we have $m_1[j] \leq i$. Assume that M' does not fulfil the condition on the $m_2[j]$'s and let i be the minimum index such that there is an edge $\{x_j, y_i\} \in M'$ with $i > m_2[j] = k$. Since x_j is matched in M' with y_i, $\overline{D}(x_j) \geq i > k$ (see Figure 2.10). This implies that y_k was matched in the second pass with L_k nodes. In M', y_k is not matched with x_j so it must be matched with a node $x_{j'}$ with $m_2[j'] \neq k$. By the choice of i, we know that $k' = m_2[j'] > k$, and as M' fulfils Lemma 4 we know that $m_1[j'] \leq k$. Since $k = m_2[j]$ and $m_1[j'] \leq k \leq m_2[j']$, we get that both x_j and $x_{j'}$ were in P during the extractions of the kth iteration of the second pass. Since x_j was extracted first, we get that $\overline{D}(x_{j'}) \geq \overline{D}(x_j) \geq i$, hence there is an edge $(x_{j'}, y_i)$. We can therefore replace in M' the edges $(x_{j'}, y_k)$ and (x_j, y_i) by the edges $(x_{j'}, y_i)$ and (x_j, y_k). We get a new matching such that $|M(y_i)| = |M'(y_i)|$ for all $1 \leq i \leq n'$ and the matching mates of

y_k still do not violate the claim. As for y_i, let d_i be the sum of the differences $i - m_2[\ell]$ over all violating matching mates x_ℓ of y_i. Since $k' > k$, d_i decreases and we can continue performing such substitutions until $d_i = 0$, which means that there are no violations at y_i. We can then advance to y_{i+1} and continue making such adjustments until we obtain the matching M. ▮

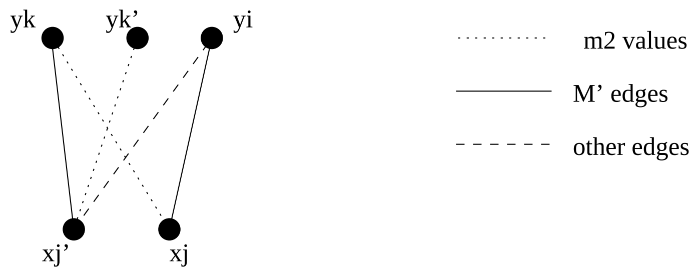


Figure 2.10: Illustration of the proof of Lemma 13

The algorithm in Figure 2.11 traverses the y nodes from right to left and deterines for each y_i the minimum number of nodes that it must be matched with in a generalized matching M that fulfils the conditions of Lemma 13. It uses a priority queue P in which the x nodes are sorted according their m_1 values. For each y_i, it first inserts into P all nodes x_κ with $m_2[\kappa] = i$; these nodes are candidates for matching with y_i. By the time the algorithm finishes handling each $y_{i'}$, it extracts from P as many nodes as $y_{i'}$ can be matched with: $U_{i'}$ nodes if there are that many in P or all nodes in P if there are less. When the algorithm comes to handle y_i, then, $y_{i+1} \dots y_{n'}$ are matched with as many nodes as possible. The algorithm extracts L_i nodes from P, because this is the minimum number of nodes that must be matched with y_i. Then it extracts from P all nodes x_κ with $m_1[\kappa] = i$, because these nodes cannot be matched in a Lemma 13-type matching M with y nodes to the left of y_i. The total number of nodes extracted so far is the minimum number of nodes that must be matched with y_i so $MinL_i$ is set to it. The algorithm then proceeds to extract additional nodes from P, up to a total of U_i as described above, so that y_i is also matched with as many nodes as possible before the algorithm proceeds to handle y_{i-1}. In the following, *iteration i* will not refer to the ith iteration but rather to the iteration that handles y_i.

Lemma 14 *The algorithm in Figure 2.11 achieves bound consistency for the lower capacity bounds of the count variables.*

```
$P \leftarrow []$ (* Priority queue sorted by $m_1[j]$ *)
for $i = n'$ to 1 do
    (* Insert into $P$ all mates of $y_i$ in $m_2$. *)
    forall $x_h$ with $m_2[h] = i$ do $P.Insert\ x_h$
    $\ell \leftarrow 0$
    for $\ell = 1$ to $L_i$ do
        $P.ExtractMax$
        $\ell \leftarrow \ell + 1$
    end for
    while $P$ is not empty and $P.MaxPriority = i$ do
        $P.ExtractMax$
        $\ell \leftarrow \ell + 1$
    end while
    $MinL_i \leftarrow \ell$
    while $P$ is not empty and $\ell < U_i$ do
        $P.ExtractMax$
        $\ell \leftarrow \ell + 1$
    end while
endfor
```

Figure 2.11: Narrowing algorithm for the lower bounds of the count variables.

Proof Let M' be the matching generated by the algorithm in Figure 2.11. That is, $M' = \{(x_j, y_i) \mid x_j \text{ was extracted in iteration } i\}$. Assume that the claim does not hold and let i be maximal such that $MinL_i > L_i$ and there is a generalized matching M with $L_i \leq |M(y_i)| < MinL_i$. By Lemma 13 we can assume that for every edge $(x_j, y_i) \in M$, $m_1[j] \leq i \leq m_2[j]$.

Let N_i be the set of nodes that the algorithm extracted from P in the first two loops of iteration i. Since $MinL_i > L_i$, after the first L_i nodes were extracted, there was at least one node $x_\kappa \in P$ with $m_1[x_\kappa] = i$ and this implies that all nodes that were extracted in iteration i have m_1-value i so they are not matched in M with any y node to the left of y_i. Let i' be the minimum index such that $i' > i$ and either P was empty at the end of iteration i' of the algorithm in Figure 2.11 or during iteration i' a node $x_{j'}$ with $m_1[j'] < i$ was extracted. If no such $i' \leq n'$ exists, let $i' = n' + 1$. Let x_κ be a node that was extracted after iteration i' and not later than iteration i. Then $m_1[\kappa] \geq i$ (if x_κ was not extracted in iteration i, this follows from the choice of i'). In addition, $m_2[\kappa] < i'$ because if we assume otherwise then x_κ was in P in iteration i', but was not extracted then. So P was not empty at the end of iteration i', which by the choice of i'' implies that some $x_{\kappa'}$ with $m_1[\kappa'] < i$ was extracted. So all nodes that remained in P after iteration i' (in particular x_κ) have m_1-value smaller than i. We get that x_κ can be matched in M only with one of $y_i \ldots y_{i'-1}$. In addition, since P was not empty after any of iterations $i+1 \ldots i'-1$, we get that for each $i < \ell < i'$, $|M'(y_\ell)| = U_\ell$.

In total, we get that $y_i \ldots y_{i'-1}$ must be matched in M with all nodes that they are matched with in M', whose number is at least $MinL_i + \sum_{\ell=i+1}^{i'-1} U_\ell$. This implies that $|M(y_i)| \geq MinL_i$.

Finally, we need to show that for all i, there is a generalized matching M_i with $|M_i(y_i)| = MinL_i$. We construct M_i as follows. First put in M_i all the edges representing the matchings of iterations n' to $i+1$. For each such edge $(x_{j'}, y_{i'})$, we have by construction that $m_1[j'] \leq i' \leq m_2[j']$. In addition, the number of nodes matched with $y_{i'}$ is between $L_{i'}$ and $U_{i'}$: At least $L_{i'}$ because there are at least $L_{i'}$ nodes x_κ with $m_2[\kappa] = i'$ which were inserted into P in iteration i'. At most $U_{i'}$, because if any node was extracted in the second loop because its m_1 value is i', then the same is true for all of the nodes that were extracted in iteration i', and this, by construction, holds for at most $U_{i'}$ nodes.

y_i is matched in M_i with the $MinL_i$ nodes that were extracted in the first two loops of the algorithm. The rest of the nodes, $y_1 \ldots y_{i-1}$, are matched with the remaining x nodes; those that remain in the queue after the $MinL_i$ extractions of iteration i and those that were not inserted to the queue up to iteration i. To match these nodes with $y_1 \ldots y_{i-1}$, we first match the nodes in P with y_{i-1} and each node $x_{j'}$ that was not yet inserted into P is matched with $y_{m_2[j']}$. The nodes that were not inserted yet to P are exactly all the nodes with $m_2[j'] < i$, so they fulfil the $L_{i'}$ values of $y_1, \ldots, y_{i-1}$. For a node $x_{j''}$ in P, $m_1[j''] \leq i-1 < m_2[j'']$. We can then apply the third pass of the generalized matching algorithm to shift nodes to the left such that the $U_{i'}$ bounds are fulfilled for $y_1, \ldots, y_{i-1}$. ∎

Figure 2.12 shows an algorithm that achieves bound consistency for the upper bounds of the count variables. Again, it uses a priority queue P for the x nodes, sorted by m_1 values. In each iteration it matches the respective y-node with the minimum required number of x nodes. At the beginning of iteration i it inserts into P all nodes which were matched with y_i in the second pass of the generalized matching algorithm. P contains at this point all candidates for matching with y_i. Let $|P|$ be the number of nodes in P at this point in time. The algorithm compares $|P|$ and U_i and sets $MaxU_i$ to the smaller of them. We will show that since the nodes that were extracted before iteration i are the minimum number of nodes that must be matched with y nodes to the right of y_i and the nodes that were not inserted into P before iteration $i-1$ cannot be matched with it, the number of nodes in P is an upper bound on the number of nodes that y_i can be matched with. The algorithm then proceeds to extract from P the minimum number of nodes that must be matched with y_i in a Lemma 13 type matching; first L_i nodes and then all nodes x_j with $m_1[j] = i$. It can then advance to iteration $i-1$.

```
$P \leftarrow []$ (* Priority queue sorted by $m_1[j]$ *)
for $i = n'$ to 1 do
    (* Insert into $P$ all mates of $y_i$ in $m_2$. *)
    forall $x_h$ with $m_2[h] = i$ do $P.Insert\ x_h$
    $MaxU_i \leftarrow \min\{U_i, |P|\}$
    for $\ell = 1$ to $L_i$ do
        $P.ExtractMax$
    end for
    while $P$ is not empty and $P.MaxPriority = i$ do
        $P.ExtractMax$
    end while
endfor
```

Figure 2.12: Narrowing algorithm for the upper bounds of the count variables.

Lemma 15 *The algorithm in Figure 2.12 achieves bound consistency for the upper capacity bounds of the count variables.*

Proof Let M be a generalized matching, by Lemma 13 we can assume that for every edge $(x_\kappa, y_\ell) \in M$, $m_1[\kappa] \leq \ell \leq m_2[\kappa]$. We fix some i with $1 \leq i \leq n'$ and we will prove $|M(y_i)| \leq MaxU_i$. We may assume that $MaxU_i < U_i$, otherwise there is nothing to show. Let M' be the partial matching generated by the algorithm in Figure 2.12 up to iteration i. That is, for $\ell = i+1, \ldots, n'$, $M'(y_\ell)$ is the set of all nodes extracted in iteration ℓ, and $M'(y_i)$ consists of all nodes that are in P when $MaxU_i$ is determined. Observe that any

x node that has not been inserted into P up to iteration i cannot be matched in M with $y_i, \ldots, y_{n'}$ because of its m_2-value. Thus $M(y_i, \ldots, y_{n'}) \subseteq M'(y_i, \ldots, y_{n'})$.

Below we identify subsets of $y_{i+1} \ldots y_{n'}$ such that for any subset S the x nodes in $M'(S)$ must be matched with the y nodes in S in any Lemma 13-type matching. We call such a subset *a block* and define it to be a consecutive set of y nodes $y_{i_\ell} \ldots y_{i_r}$ such that (1) for every matching mate x_ℓ of y_{i_ℓ}, $m_1[\ell] = i_\ell$ and (2) i_r is minimal such that $i_r \geq i_\ell$ and y_{i_r+1} is matched with some node x_r with $m_1[r] < i_\ell$ (see Figure 2.13).

Let $B = \{y_{i_\ell} \ldots y_{i_r}\}$ be a block. By construction, for any x_j which is matched with $y_\kappa \in B$, $m_1[j] \geq i_\ell$. In addition, for every such x_j, $m_2[j] \leq i_r$; otherwise x_j was in P in iteration i_{r+1} and since x_r was extracted first, $m_1[j] \leq m_1[r] < i_\ell$, contradicting condition (1) in the definition of a block above. We get that $M'(B) \subseteq M(B)$.

We partition $y_{i+1} \ldots y_{n'}$ into two sets B_{in} and B_{out} where B_{in} contains the nodes that belong to some block and B_{out} contains the nodes that do not belong to any block. If a node y_κ for some $\kappa > i$ is matched in M' with more than L_κ nodes then it is in B_{in} because for any of its matching mates x_j, $m_1[j] = \kappa$. This implies that any $y_\kappa \in B_{\text{out}}$ is matched with exactly L_κ x nodes in M'.

From the arguments above we conclude that $M(y_i \cup B_{\text{out}} \cup B_{\text{in}}) \subseteq M'(y_i \cup B_{\text{out}} \cup B_{\text{in}})$ and $M'(B_{\text{in}}) \subseteq M(B_{\text{in}})$, which implies $M(y_i \cup B_{\text{out}}) \subseteq M'(y_i \cup B_{\text{out}})$. So we get that $|M(y_i \cup B_{\text{out}})| \leq MaxU_i + \sum_{y_\kappa \in B_{\text{out}}} L_\kappa$. Thus $|M(y_i)| \leq MaxU_i$.

It remains to show that for every i there is a generalized matching M_i with $|M_i(y_i)| = MaxU_i$. For a node $y_{i'}$ with $i' > i$, let $M_i(y_{i'})$ be the nodes that were extracted from P in iteration i'. If $|M_i(y_{i'})| > L_{i'}$ then for every node $x_{j'}$ that was extracted in iteration i', $m_1[j'] = i'$ so $|M_i(y_{i'})| \leq U_{i'}$. Otherwise, $|M_i(y_{i'})| = L_{i'}$. In either case the capacities are respected. $M_i(y_i)$ is set to the topmost $MaxU_i$ nodes in the queue at the time when $MaxU_i$ is computed. By construction, for each such node x_j, $m_1[j] \leq i \leq m_2[j]$. Finally, $y_1, \ldots y_{i-1}$ are matched with the remaining nodes in P and the x nodes that were not yet inserted into P as described in the proof of Lemma 14. ∎

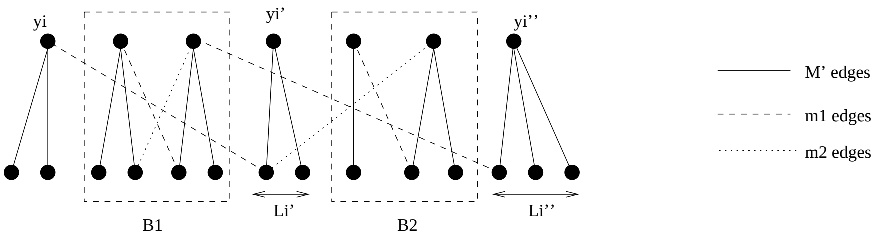


Figure 2.13: Example of the blocks in the proof of Lemma 15

Implementing the algorithm in linear time

As with the generalized matching algorithm, the algorithms in Figures 2.11 and 2.12 can be implemented in time $O(n' + n \log n)$ using a binary heap for the priority queue P. To shave off the logarithmic factor, we once again simulate the queue with offline-min type algorithms. In the lower bounds algorithm (Figure 2.11), the insertions and extractions of each iteration do not depend on the results of previous extractions from P; in iteration i the algorithm extracts $\max\{U_i, |P|\}$ nodes. This means that we can use the usual offline-min algorithm to determine the sequence of insertions and extractions and then use this sequence to find the $MinL_i$ values. For the upper bounds algorithm (Figure 2.12) we need to know the results of the extractions in the while loop in order to know how many nodes were extracted in each iteration. Note that this is similar to the situation of the second pass of the generalized matching algorithm (cf. Figure 2.3), and can be handled by the algorithm in Figure 2.14 which is analogous to the one in Figure 2.6 on Page 17.

2.7 Conclusion

We have designed a propagation algorithm for the Global Cardinality Constraint that achieves bound consistency for the assignment variables. We wish to point out that there are two possible implementations for the algorithm. One runs in time $O((n' + n) \log n)$, uses very simple data structures and performs well in practice. The other one requires more elaborate data structures (for the offline-min computation) and achieves a running time of $O(n + n')$ plus the time required to sort the assignment variables by the endpoints of their ranges. In some cases the latter is asymptotically better.

In addition, we present two linear-time algorithms that narrow the bounds of the count

```
$k \leftarrow 1$(* $E_k$ will always be the next mandatory extraction *)
**for** $i = n'$ to 1 **do**
    **forall** $x_j$ with $m_2[j] = i$ **do**
        (* $x$'s inserted in iteration $i$ (before $E_k$) *)
        $s[j] \leftarrow k$
    **endfor**
    **for** $\ell = 1$ to $L_i$ **do**
        create set $\{E_k\}$ labelled $\mathcal{L}_k^{(i,\ell)}$ (* for mandatory extraction $\ell$ in iter. $i$ *)
        $k \leftarrow k + 1$
    **endfor**
**endfor**
create set $\{E_k\}$ labelled with $\mathcal{L}_k^{(n'+1,0)}$ (* dummy extraction *)
**forall** $x_j$ sorted in descending order according to $m_1[j]$ **do**
    $S \leftarrow$ find($E_{s[j]}$); let $\mathcal{L}_k^{(i,\ell)}$ be its label
    **if** $i \geq m_1[j]$ **then**
        $x_j$ is removed by extraction $\ell$ in the for-loop of iteration $i$
        unite $S$ with the set $S'$ containing $E_{k+1}$ and
            label the union with the former label of $S'$
    **else**
        $x_j$ is extracted by the while-loop in iteration $m_1[j]$
    **endif**
**endfor**
```

Figure 2.14: Enhanced offline-min procedure for the algorithm in Figure 2.12

variables. The first is very simple but does not always achieve bound consistency. The second does achieve bound consistency but uses the more elaborate data structures that are required for the offline-min computations.

Several questions remain open. The first is whether there is an efficient algorithm that uses the Hall interval approach and can narrow the domains of the count variables. The second is whether there are efficient algorithms that achieve more complex tasks, such as narrowing the domains of the count variables when the assignment variables are not intervals. It has been shown that computing arc consistency for all variables is NP-hard [21].

Chapter 3

Filtering Algorithms for the *Same* and *UsedBy* Constraints

As a motivating example, we consider simple scheduling problems of the following type. The organization Doctors Without Borders [28] has a list of doctors and a list of nurses, each of whom volunteered to go on one rescue mission in the next year. Each volunteer specifies a list of possible dates and each mission should include one doctor and one nurse. The task is to produce a list of pairs such that each pair includes a doctor and a nurse who are available on the same date and each volunteer appears in exactly one pair. Since the list of potential rescue missions at any given date is infinite, it does not matter how the doctor-nurse pairs are distributed among the different dates.

We model this problem by the $\mathit{Same}(X = \{x_1, \ldots, x_n\}, Z = \{z_1, \ldots, z_n\})$ constraint which is defined on two sets X and Z of distinct variables such that $|X| = |Z|$ and each $v \in X \cup Z$ has a domain $D(v)$. A solution is an assignment of values to the variables such that the value assigned to each variable belongs to its domain and the multiset of values assigned to the variables of X is identical to the multiset of values assigned to the variables of Z.

This problem can be generalized to the case in which there are more nurses than doctors and the task is to create a list of pairs as above, with the requirement that every doctor appears in exactly one pair and every nurse in at most one pair (naturally, not all of the nurses will be paired). For this version we use the general case of the $\mathit{UsedBy}(X = \{x_1, \ldots, x_n\}, Z =$

$\{z_1, \ldots, z_m\}$) constraint where $|X| = n \geq m = |Z|$ and a solution is an assignment of values to the variables such that the multiset of values assigned to the variables of Z is contained in the multiset of values assigned to the variables of X.

Given a constraint with domains for the variables, the first question is whether there exists an assignment of values to the variables which satisfies the constraint. The second question is whether we can efficiently identify elements in the domains of the variables that cannot participate in any solution to the constraint. An algorithm for this task is called a *filtering* algorithm. Filtering algorithms are classified according to the level of consistency they achieve. The *arc consistency* problem is to reduce the domains of the variables such that for all $v \in X \cup Z$ and $i \in D(v)$, there is a solution to the constraint in which v is assigned the value i. In the *bound consistency* problem we assume that there is a total order on the values in the domains of the variables and that for each $v \in X \cup Z$, $D(v)$ is a contiguous interval of values, i.e., $D(v) = [\underline{D}(v), \overline{D}(v)]$. The problem is to shrink these intervals to the minimum possible sizes without losing any solutions. This implies that if the domains are bound consistent, then for each $v \in X \cup Z$, there is at least one solution to the constraint in which v is assigned the value $\underline{D}(v)$ and there is at least one solution in which it is assigned the value $\overline{D}(v)$.

3.0.1 *Same* $= 2\times$ *GCC*?

The *Same* constraint can be modeled by two *Global Cardinality* constraints, one on the set X and the other on the set Z, where count variables which are associated with the same value are not duplicated. We show here that consistency for all of the variables of the *GCC* constraints (including assignment and count variables) does not imply consistency for the *Same* constraint.

In our example, $|X| = |Z| = 2$ and $|Y| = 4$. The domains of the assignment variables are: $D(x_1) = \{1, 2\}$, $D(x_2) = \{3, 4\}$, $D(z_1) = \{1, 2, 3, 4\}$ and $D(z_2) = \{3, 4\}$. By examining the variable-value graphs[1] shown in Figure 3.1, one can easily see that all values are consistent with respect to the two *GCC* constraints $GCC(\{x_1, x_2\}, \{v_1, v_2, v_3, v_4\})$ and $GCC(\{z_1, z_2\}, \{v_1, v_2, v_3, v_4\})$, but that an arc consistency or bound consistency computation for $Same(\{x_1, x_2\}, \{z_1, z_2\})$ would remove 3 and 4 from the domain of z_1 – if z_1 is assigned

[1]This construction will be formally defined in Section 3.1.

3 or 4, then 1 and 2 cannot both be assigned to the same number of variables from X and Z because one of them must be assigned to x_1 and neither can be assigned to z_2.

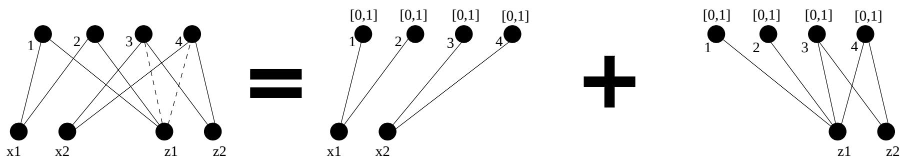


Figure 3.1: Example showing that consistency of the two *GCC*'s does not imply consistency of the *Same* constraint.

3.0.2 Our contribution

In this chapter we show filtering algorithms for the *Same* and *UsedBy* constraints. Let Y be the union of the domains of the variables in $X \cup Z$ and let $n' = |Y|$. The arc consistency algorithms run in time $O(n^2 n')$ and the bound consistency algorithms in time $O(n'\alpha(n', n') + n \log n)$, where α is the inverse of Ackermann's function[2].

We begin with *Same* and then generalize to *UsedBy*. The general approach resembles the flow-based filtering algorithms for the *AllDifferent* [16, 25] and *Global Cardinality* (*GCC*) [11, 26] constraints: We construct a bipartite variable-value graph, find a single solution in it and compute the strongly connected components (SCCs) of the residual graph. We show that an edge is consistent iff both of its endpoints are in the same SCC.

The main difference compared to the previous constraints that were solved by the flow-based approach is that we now have three sets of nodes. One set for each set of variables and a third set for the values. This difference significantly complicates the bound consistency algorithms, in particular the SCC computation compared to the corresponding stage in the *AllDifferent* and *GCC* cases. Our contribution is therefore not only in providing a solution to these constraints but also in showing that the ideas that appear in the previous algorithms can be extended to much more complex variable-value graphs.

In Section 3.1 we define the variable-value graph for the *Same* constraint and characterize the solutions to the constraint in terms of subsets of the edges in this graph. In Section 3.2 we show the arc consistency algorithm for the *Same* constraint and in Section 3.3 we show the bound consistency algorithm. Finally, in Section 3.4 we deal with the generalization to

[2]For all practical purposes, $\alpha(n', n')$ can be regarded as a small constant.

UsedBy. Source code for the bound consistency algorithms is available by request from the authors. A preliminary version of Sections 3.1-3.3 appeared as [3].

3.1 The *Same* constraint

We represent the *Same* constraint as a bipartite graph $B = (X \cup Z, Y, E)$, which we call the *variable-value graph*, where $E = \{\{v, y\} | v \in X \cup Z \wedge y \in Y \wedge y \in Dom(v)\}$. That is, the nodes on one side represent the variables and the nodes on the other represent the values and every variable is connected by an edge to all values in its domain.

The following definition and lemma characterize the set of all solutions to the constraint in terms of subsets of edges of B.

Definition 3.1.1 *Let $M \subseteq E$ be a set of edges of B. For any node $v \in X \cup Y \cup Z$, let $N_M(v)$ be the set of nodes which are neighbors of v in $B' = (X \cup Z, Y, M)$. We say that M is a* parity matching *in B iff $\forall_{v \in X \cup Z} |N_M(v)| = 1$ and $\forall_{y \in Y} |N_M(y) \cap X| = |N_M(y) \cap Z|$.*

Lemma 16 *There is a one to one correspondence between the solutions to the Same constraint and the parity matchings in B.*

Proof Given a parity matching M in B, we can construct the solution

$$Same(\{N_M(x_1), \ldots, N_M(x_n)\}, \{N_M(z_1), \ldots, N_M(z_n)\}).$$

Since $|N_M(v)| = 1$ for all $v \in X \cup Z$, all of the assignments are well defined. In addition, for each edge (v, y) in B, and in particular in M, $y \in Dom(v)$. Finally, since $|N_M(y) \cap X| = |N_M(y) \cap Z|$ for all $y \in Y$, each value is assigned the same number of times to variables of X and Z. Hence, the constraint is satisfied.
On the other hand, given a solution $Same(\{y(x_1), \ldots, y(x_n)\}, \{y(z_1), \ldots, y(z_n)\})$ where $y(v)$ is the value assigned to the variable v, we can obtain the set of edges $M = \{\{v, y\} | v \in X \cup Z \wedge y = y(v)\}$. Since $y(v) \in Dom(v)$ for all v, we have $M \subseteq E$. In addition, since $y(v)$ is determined for all variables, we have that $|N_M(v)| = 1$ for all $v \in X \cup Z$ and since each $y \in Y$ appears the same number of times in $\{y(x_1), \ldots, y(x_n)\}$ and in $\{y(z_1), \ldots, y(z_n)\}$), we have that $|N_M(y) \cap X| = |N_M(y) \cap Z|$, so M is a parity matching. ∎

j	$D(x_j)$	$D(z_j)$
1	{1,2}	{2,3}
2	{3,4}	{4,5}
3	{4,5,6}	{4,5}

Table 3.1: Domains of the variables for our example.

In the next sections we show the filtering algorithms for the *Same* constraint, first the arc consistency algorithm and then bound consistency. We will illustrate the algorithm with the aid of the following example. $|X| = |Z| = 3$, $|Y| = 6$ and the domains of the variables of $X \cup Z$ are as in Table 3.1. The domains are intervals, which is suitable for bound consistency. For the arc consistency setting, each domain $[a, b]$ is to be interpreted as the set $\{a, \dots, b\}$.

3.2 Arc consistency

In this section we show an algorithm that achieves arc consistency for the *Same* constraint. Inspired by Régin [26], we convert the graph B into a capacitated and directed graph $\vec{B} = (\vec{V}, \vec{E})$, as follows. We direct the edges from X to Y and from Y to Z and assign capacity requirements of $[0, 1]$ to each of these edges. We add two nodes s and t, an edge with capacity $[1, 1]$ from s to each $v \in X$ and from each $v \in Z$ to t and an edge with capacity $[n, n]$ from t to s. Figure 3.2 shows this graph for the constraint on the variables in Table 3.1. A flow in $\vec{B}$ is feasible iff there is a flow of value n on the arc from t to s. This implies that one unit of flow goes through every node in $X \cup Z$. By flow conservation, every node in Y is connected by edges that carry flow to the same number of nodes from X and from Z.

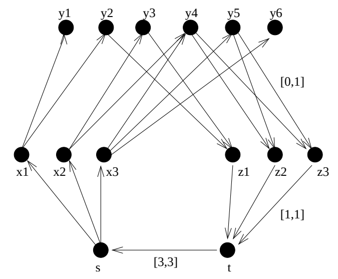


Figure 3.2: The capacitated graph for the example in Table 3.1.

Observation 1 *There is a one to one correspondence between parity matchings in B and* *feasible flows in $\vec{B}$.*

The algorithm uses Ford and Fulkerson's augmenting paths method to find a feasible flow f in $\vec{B}$ (e.g., the flow shown by the bold edges in Figure 3.3). If there is no such flow it reports that the constraint is not satisfiable. Otherwise, it removes the nodes s and t and builds the *residual graph* $\vec{B}_f = (\vec{V}_f, \vec{E}_f)$ where $\vec{V}_f = X \cup Y \cup Z$ and $\vec{E}_f = \vec{E} \cup \{(v,u) | u, v \in \vec{V} \wedge (u,v) \in \vec{E} \wedge f(e) = 1\}$. That is, all edges appear in their original orientation and the edges that carry flow appear also in reverse direction (see Figure 3.4). The following lemma shows that $\vec{B}_f$ can be used to determine which of the edges of the graph are consistent.

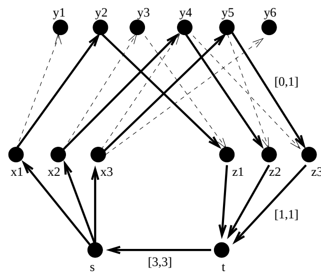


Figure 3.3: A feasible flow in the graph of Figure 3.2.

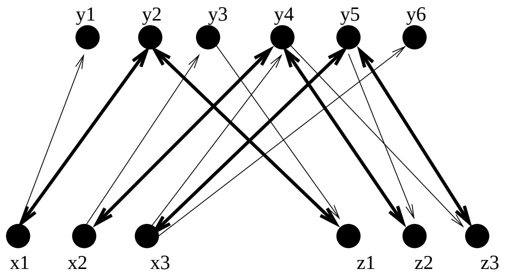


Figure 3.4: The residual graph with respect to the flow of Figure 3.3.

Lemma 17 *An edge $e = (u,v) \in \vec{B}_f$ is consistent iff u and v belong to the same SCC.*

Proof If $f(e) = 1$ then e participates in the solution that corresponds to the flow f and is therefore consistent. u and v are connected by two antiparallel edges so they are in the same SCC. Let $M = \{e' | f(e') = 1\}$ be the parity matching that corresponds to the flow f. Assume that $e \notin M$ and that u and v are in the same SCC. Then there is a cycle P which uses e in $\vec{B}_f$. Starting at any y node on the cycle, number its nodes: $y_0, v_0, y_1, v_1, \ldots, y_{p-1}, v_{p-1}$ where $y_i \in Y$ and $v_i \in X \cup Z$ for all $0 \leq i < p$. Let $M' = M \oplus P$ be the symmetric difference between

the parity matching M and the cycle P. That is, $M' = \{e' | e' \in M \wedge e' \in P \wedge e' \notin M \cap P\}$. If we look at $y_i \in P$ and its two neighbors, we have four possibilities: If $v_{(i-1) \bmod p}, v_i \in X$ or $v_{(i-1) \bmod p}, v_i \in Z$ then in M' each of $v_{(i-1) \bmod p}, v_i$ remains matched and y_i is matched with the same number of nodes from each of X and Z as it was in M'.
The two other options are that $v_{(i-1) \bmod p} \in X$ and $v_i \in Z$ or $v_{(i-1) \bmod p} \in Z$ and $v_i \in X$. Then the edges $(v_{(i-1) \bmod p}, y_i)$ and (y_i, v_i) are either both in M or both not in M. To see that this is true, note that the cycle must leave each $v \in X \cup Z$ that it enters. It can only enter a node $v \in X$ by an edge which is in M so it must leave it by an edge which is not in M. On the other hand, it can only enter a node $v \in Z$ by an edge which is not in M and leave it by an edge in M. This implies that either $v_{(i-1) \bmod p} \in Z$ and $v_i \in X$ and both edges are in M or $v_{(i-1) \bmod p} \in X$ and $v_i \in Z$ and both edges are in M'. In either case, when comparing M with M', y_i either lost or gained a neighbor from each of X and Z, so the number of neighbors of y_i in each of those sets remains equal.

Each of $v_{(i-1) \bmod p}, v_i$ is adjacent on P to one edge from M and one edge which is not in M. Hence, in M' it is still matched with exactly one y-node. We get that M' is a parity matching that contains e, so e is consistent.

It remains to show that an edge e which is not in an SCC of $\vec{B}_f$ (which implies $f(e) = 0$) is not consistent. Let $C_1, \ldots, C_k$ be the SCCs of $\vec{B}_f$. Since all edges in M appear in both directions, any edge between two SCCs is not in M. For any node v, let $C(v)$ be the SCC that v belongs to and let $E_{in} = \{(u, v) | C(u) = C(v)\}$ be the edges for which both endpoints are in the same SCC. Assume that an edge $e \in E \setminus E_{in}$ is consistent and let M' be a parity matching such that $e \in M'$. Consider the graph $\vec{B}'_f = (\vec{V}_f, \vec{E}'_f)$ where $\vec{E}'_f = E_{in} \cup M'$. That is, $\vec{B}'_f$ contains all the edges within SCCs plus the edges between SCCs which are in M'. If we shrink each SCC of $\vec{B}'_f$ into a single node, we get a DAG (directed acyclic graph). Let D_e be the connected component of this DAG which contains e and let C be a root of D_e, i.e., there are only outgoing edges from C. Let E_{xy} be the $x \to y$ edges and E_{yz} the $y \to z$ edges out of C. Since C is not an isolated node, $|E_{xy}| + |E_{yz}| \geq 1$. Let Y_C be the set of y nodes in the SCC represented by C. Then $|M'(Y_C) \cap X| = |M(Y_C) \cap X| - |E_{xy}|$ and $|M'(Y_C) \cap Z| = |M(Y_C) \cap Z| + |E_{yz}|$. Hence, M and M' cannot both be parity matchings, contradicting our assumption. ▮

Figure 3.5 shows the SCCs of $\vec{B}_f$ for the example in Table 3.1. The nodes of each SCC have a distinct shape and the inconsistent edges are dashed.

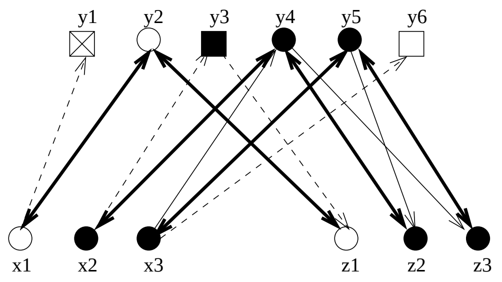


Figure 3.5: The SCCs of the residual graph of Figure 3.4.

Let $|E|$ denote the number of edges in $\vec{B}$ and recall that $n = |X| = |Z|$ and $n' = |Y|$. Clearly, $|E| = O(nn')$. The running time of the algorithm is dominated by the time required to find a flow, which is $O(n|E|) = O(n^2n')$ [10, 26].

3.3 Bound consistency

In this section we show the bound consistency algorithm. It does the same as the arc consistency algorithm of the previous section, but achieves faster running time by exploiting the simpler structure of the variable-value graph: Since the domain of every variable node is an interval, the variable-value graph is *convex*, which means that the neighborhood of every variable node is a consecutive sequence of value nodes. In the next subsections we show how each step of the arc consistency algorithm can be performed efficiently on a convex graph. In particular, we show how to find a parity matching in time $O(n' + n \log n)$ and how to compute the SCCs of the residual graph in time $O(n'\alpha(n', n'))$.

3.3.1 Finding a parity matching

Figure 3.6 shows the algorithm for finding a parity matching in the graph B. It uses two priority queues, P_x for the nodes in X and P_z for the nodes in Z. In both queues the nodes are sorted by the upper endpoints of their domains.

For any $v \in X \cup Z$, let $\underline{D}(v)$ and $\overline{D}(v)$ denote the lower and upper endpoints of $D(v)$, respectively. The algorithm traverses the value nodes from y_1 to $y_{n'}$ and for each y_i inserts to the respective queue all variable nodes $v \in X \cup Z$ with $\underline{D}(v) = i$. It then checks whether

```
(* Assumption: $X$ and $Z$ are sorted according to $\underline{D}$. *)
$P_x \leftarrow []$ (* priority queue containing $x$ nodes sorted by $\overline{D}$ *)
$P_z \leftarrow []$ (* priority queue containing $z$ nodes sorted by $\overline{D}$ *)
$j \leftarrow 0$
for $i = 1$ to $n'$ do
    forall $x_h$ with $\underline{D}(x_h) = i$ do $P_x.Insert\ x_h$
    forall $z_h$ with $\underline{D}(z_h) = i$ do $P_z.Insert\ z_h$
    (* Assume that MinPriority of an empty queue is $\infty$ *)
    while $P_x.MinPriority = i$ or $P_z.MinPriority = i$ do
        if $P_x$.IsEmpty or $P_z$.IsEmpty then report failure
        $j \leftarrow j + 1$
        $x \leftarrow P_x.ExtractMin$; match $x$ with $y_i$
        $z \leftarrow P_z.ExtractMin$; match $z$ with $y_i$
    end while
endfor
if $P_x$ and $P_z$ are both empty then report success
else report failure
```

Figure 3.6: Algorithm to find a parity matching in a convex graph.

there is a node in one of the queues (the node with minimum priority) whose domain ends at i. If so, it tries to match this node and a node from the other queue with y_i. If the other queue is empty, it declares that there does not exist a parity matching in the graph.

Figure 3.7 shows the parity matching obtained by the algorithm for the example in Table 3.1. It corresponds to the flow shown in Figure 3.3. Note that while the flow algorithm could have produced any one of several different flows, this matching is one of the only two possible outputs of the algorithm of Figure 3.6. The other option would be to match z_2 with y_5 and z_3 with y_4 instead of the other way around.

Lemma 18 *If there is a PM in B then the algorithm in Figure 3.6 finds one.*

Proof We show by induction on i that if there is a PM M then for all $0 \leq i \leq n'$, there is a PM M_i in B which matches $\{y_1, \ldots, y_i\}$ with the same matching mates as the algorithm.

For $i = 0$, the claim holds with $M_0 = M$. For larger i, given a PM M, we can assume by the induction hypothesis that there is a PM M_{i-1} that matches the nodes in $\{y_1, \ldots, y_{i-1}\}$

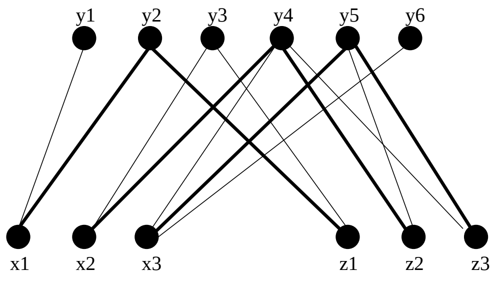


Figure 3.7: The parity matching obtained by the algorithm in Figure 3.6 for our running example.

with the same matching mates as the algorithm. We show how to construct M_i from M_{i-1}. As long as the matching mates of y_i are not the same as the ones determined by the algorithm, perform one of the following transformations:

If y_i is matched with a pair x_j, z_k such that neither one of x_j and z_k was matched with y_i by the algorithm, then since x_j and z_k were not matched by the algorithm with any of $\{y_1, \ldots, y_{i-1}\}$, they both remained in the queues after iteration i, which implies $\overline{D}(x_j) > i$ and $\overline{D}(z_k) > i$. In M_i we match both of them with y_{i+1}.

The other option is that the algorithm matched y_i with a pair x_j, z_k which are not both matched with y_i in M_i. Since the algorithm extracted them from the queues, we know that at least one of them has upper domain endpoint equal to i. Assume w.l.o.g. that $\overline{D}(x_j) = i$. Since M_{i-1} agrees with the algorithm on the matching mates of $\{y_1, \ldots, y_{i-1}\}$, we get that in M_{i-1}, x_j is matched with y_i and z_k is matched with $y_{i'}$ for some $i' > i$. Hence, there is some other node $z_{k'} \in Z$ which is matched with y_i in M_{i-1} and which was matched by the algorithm with $y_{i''}$ for some $i'' > i$. When the algorithm extracted z_k from the queue, $z_{k'}$ was in the queue because it is a neighbor of y_i. Hence, $\overline{D}(z_{k'}) \geq \overline{D}(z_k)$ so we can replace z_k and $z_{k'}$. That is, we can match z_k with y_i and $z_{k'}$ with $y_{i'}$.

Each time we apply one of the transformations above to M_{i-1}, we decrease the number of differences between the set of matching mates of y_i in M_{i-1} and in the matching generated by the first i iterations of the algorithm. So we can continue until we obtain a matching M_i that agrees with the algorithm on the matching mates of $\{y_1, \ldots, y_i\}$. ▮

Lemma 19 *If the algorithm in Figure 3.6 reports success then it constructs a parity matching in B.*

Proof If the algorithm reports success then P_x and P_z are empty at the end, which means that all nodes in $X \cup Z$ were extracted and matched with y nodes. In addition, since the algorithm did not report failure during the extractions, whenever $v \in X \cup Z$ was matched with y_i, we have that $\underline{D}(v) \leq i \leq \overline{D}(v)$. For all $1 \leq i \leq n'$, whenever y_i is matched with some node from X it is also matched with a node from Z, and vice versa. Hence, we get that the matching that was constructed is a parity matching. ▮

3.3.2 Finding strongly connected components

Having found a parity matching in B, which we can interpret as a flow in $\vec{B}$, we next wish to find the SCCs of $\vec{B}_f$ (cf. Figure 3.5). Mehlhorn and Thiel [16] gave an algorithm that does this in the residual graph of the *AllDifferent* constraint in time $O(n)$ plus the time required for sorting the variables according to the lower endpoints of their domains. Katriel and Thiel [11] enhanced this algorithm for the *GCC* constraint, in which a value node can be matched with more than one variable node. For our graph, we need to construct a new algorithm that can handle the distinction between the nodes in X and in Z and the more involved structure of the graph.

As in [11, 16], the algorithm in Figure 3.10 begins with n' initial components, each containing a node $y_i \in Y$ and its matching mates (if any). It then merges these components into larger ones. While the algorithm used for the *AllDifferent* graph can do this in one pass over the y nodes from $y_1, \ldots, y_{n'}$, our algorithm makes two such passes for reasons that will be explained in the following. The first pass resembles the SCC algorithm for the *AllDifferent* graph. It traverses the y nodes from y_1 to $y_{n'}$ and uses a stack to merge components which are strongly connected and are adjacent to each other. It maintains a list *Comp* of completed components and a stack *CS* of temporary components. The components in both *Comp* and *CS* are not guaranteed to be SCCs of $\vec{B}_f$. They are strongly connected but may not be maximal. However, a component in *Comp* is completed with respect to the first pass, while the components in *CS* may still be merged with unexplored components and with other components in *CS*.

Let $\vec{B}_f^i$ be the graph induced by $\{y_1, \ldots, y_i\}$ and their matching mates. The first pass begins with the empty graph $\vec{B}_f^0$ and in iteration i moves from the graph $\vec{B}_f^{i-1}$ to $\vec{B}_f^i$, as follows. As long as the topmost component in CS does not reach any $y_{i'}$ with $i' > i$ by a single edge, this component is popped from CS and appended to $Comp$. Then the algorithm creates a new component C with y_i and its matching mates. It repeatedly checks if C and the topmost component in CS reach each other by a single edge in each direction. If so, it pops the component from CS and merges it into C. Finally, it pushes C onto CS and proceeds to the next iteration (see Figure 3.8).

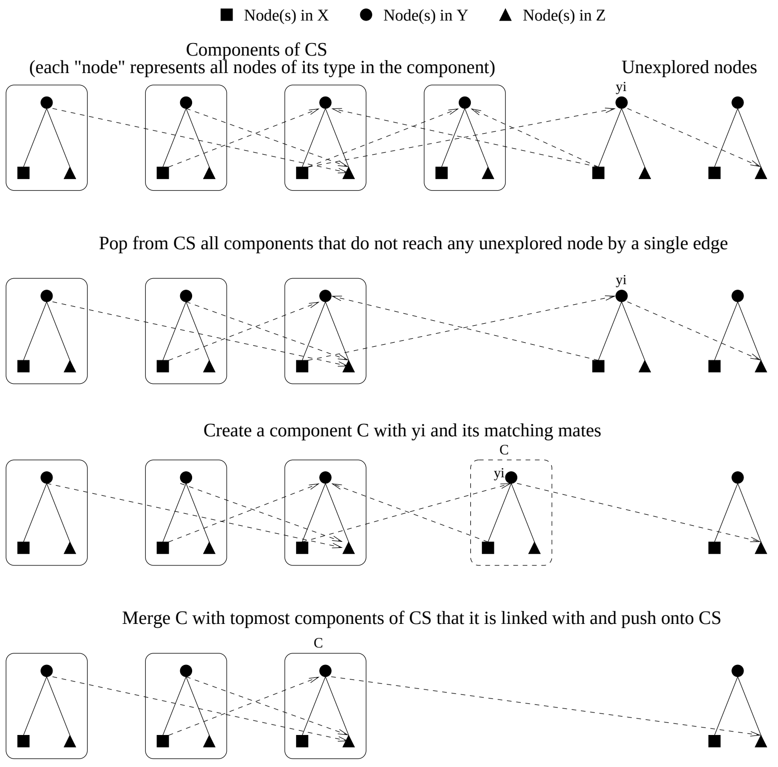
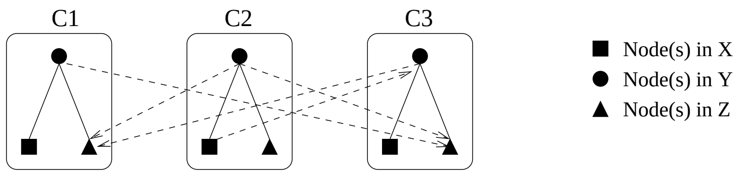

■ Node(s) in X
● Node(s) in Y
▲ Node(s) in Z
Components of CS
(each "node" represents all nodes of its type in the component)
Unexplored nodes
yi
Pop from CS all components that do not reach any unexplored node by a single edge
yi
Create a component C with yi and its matching mates
C
yi
Merge C with topmost components of CS that it is linked with and push onto CS
C

Figure 3.8: Iteration i of the first pass of the algorithm in Figure 3.10.

The reason that this pass is enough for the *AllDifferent* graph but not for ours is that in our case the outgoing edges of a y-node do not fulfil the convexity criteria: It could be that there is an edge from y_i to a z node which is matched with $y_{i'}$ for some $i' > i + 1$ while there is no edge from y_i to any of y_{i+1}'s matching mates. In the *AllDifferent* case, this could not happen: If a matching mate of y_i can reach any $y_{i'}$ by a single node then convexity implies that it can reach all y nodes between y_i and $y_{i'}$. This means that in our graph, there could be two components C, C' in CS such that C reaches C' by a single edge and C' reaches C by a single edge, but this is not detected when the second of them was inserted into CS because the first was not the topmost in the stack (see, e.g., Figure 3.9). The second pass, which merges such components, will be described later.



C1 and C2 are strongly connected but will not be merged in the first pass because C2 (and not C1) is the topmost component in CS when C3 is created and pushed onto CS.

Figure 3.9: Example of components that will not be merged by the first pass of the SCC algorithm.

In the following, whenever we speak of a component C of $\vec{B}_f$, we refer to a set of nodes such that for every node in C, all of its matching mates are also in C. This means that a component C is strongly connected, but may not be maximal. We say that a component C *reaches* a component C' if there is a path in $\vec{B}_f$ from a node in C to a node in C'. In addition, two components C and C' are *linked* if there is an edge from C to C' and there is an edge from C' to C. They are *linked by* $x \rightarrow y$ *(linked by* $y \rightarrow z$*) edges* if the edges in both directions are $x \rightarrow y$ ($y \rightarrow z$) edges.

For clarity, the pseudo-code in Figure 3.10 uses the following shortcuts. We will show how to implement them in Subsection 3.3.4. Let C, C' be two components.

- $MinY[C]$ ($MaxY[C]$) is the minimum (maximum) index of a y node in C.
- *ReachesRight*$[C]$ is the largest index i such that y_i or one of its matching mates can be reached by a single edge from a node in C.
- *xyLeftLinks*$[C]$ (*yzLeftLinks*$[C]$) is true iff C is linked with some component to its left by $x \rightarrow y$ ($y \rightarrow z$) edges.
- *Linked*$[C, C']$ is true iff C and C' are linked.
- *xyLinks*$[C, C']$ (*yzLinks*$[C, C']$) is true iff C and C' are linked by $x \rightarrow y$ ($y \rightarrow z$) edges.

The following lemmas examine the components that are generated by the first pass of the algorithm, first with respect to their order and connectivity in CS and then with respect to the SCCs of $\vec{B}_f$ that they compose. They will help us to show that the SCCs of $\vec{B}_f$, when viewed as combinations of components that are generated by the first pass of the algorithm, have a relatively simple structure which enables to identify them in the second pass.

```
(* Pass 1: Start with singleton components and merge adjacent ones *)
$Comp \leftarrow$ empty list; $CS \leftarrow$ empty stack
for $i = 1$ to $n'$ do
    while $CS$ not empty $\wedge ReachesRight(\mathrm{Top}(CS)) < i$ do (* Top($CS$) cannot reach $y_{i'}$ with $i' \geq i$. *)
        $C' \leftarrow \mathrm{Pop}(CS)$; append $C'$ to $Comp$
    end while
    $C \leftarrow \{y_i\} \cup N_M(y_i)$
    while $CS$ not empty $\wedge Linked(TOP(CS), C)$ do
        $C' \leftarrow \mathrm{Pop}(CS)$; $C \leftarrow C' \circ C$ (* Merge components *)
    end while
    push $C$ onto $CS$
end for
while $CS$ not empty do $C' \leftarrow \mathrm{Pop}(CS)$; append $C'$ to $Comp$
(* Pass 2: merge non-adjacent components *)
$SCCs \leftarrow$ empty list; $CS \leftarrow$ empty stack
for $i = 1$ to $|Comp|$ do (* Traverse the components of $Comp$ by $MinY[C]$ order *)
    while $CS$ not empty $\wedge ReachesRight(\mathrm{Top}(CS)) < MinY[C_i]$ do (* Top($CS$) cannot reach $C_{i'}$ with $i' \geq i$. *)
        $C \leftarrow \mathrm{Pop}(CS)$
        if $xyLeftLinks[C]$ then push $C$ onto $CSxy$
        else if $yzLeftLinks[C]$ then push $C$ onto $CSyz$
        else append $C$ to $SCCs$
        if $xyLinks[\mathrm{Top}(CS), \mathrm{Top}(CSxy)]$ then
            $C \leftarrow \mathrm{Pop}(CS)$; $C' \leftarrow \mathrm{Pop}(CSxy)$; $C \leftarrow C' \circ C$ (* Merge components *)
            while $Linked[C, \mathrm{Top}(CS)]$ do $C' \leftarrow \mathrm{Pop}(CS)$; $C \leftarrow C' \circ C$ (* Merge components *)
            push $C$ onto $CS$
        endif
        if $yzLinks[\mathrm{Top}(CS), \mathrm{Top}(CSyz)]$ then
            $C \leftarrow \mathrm{Pop}(CS)$; $C' \leftarrow \mathrm{Pop}(CSyz)$; $C \leftarrow C' \circ C$ (* Merge components *)
            while $Linked[C, \mathrm{Top}(CS)]$ do $C' \leftarrow \mathrm{Pop}(CS)$; $C \leftarrow C' \circ C$ (* Merge components *)
            push $C$ onto $CS$
        endif
    end while
    push $C_i$ onto $CS$
endfor
return $SCCs$
```

Figure 3.10: Algorithm to find the SCCs of the residual graph.

Lemma 20 *Let $CS = < C_1, C_2, \ldots >$ be the components in CS at the end of iteration i of the first pass (ordered from bottom to top). Then for all κ, $MaxY[C_\kappa] < MinY[C_{\kappa+1}]$. In other words, no component is nested in another and the components appear in CS in increasing order of the indices of their y nodes.*

Proof At the beginning CS is empty and the claim clearly holds. Assume that the claim holds after iteration $i-1$ and consider the changes made to the stack during iteration i. First some of the topmost components are popped from the stack; this does not affect the correctness of the claim. Then some of the topmost components are merged with each other and with the new component C and the result is pushed to the top of the stack. By the induction hypothesis, all components that were popped and merged contain y nodes with larger indices than the components that remained in the stack. In addition, all y nodes in CS after iteration $i-1$ have indices smaller than i. We get that the claim holds after iteration i. ∎

Lemma 21 *Let $CS = < C_1, C_2, \ldots >$ be the components in CS at the end of iteration i of the first pass (ordered from bottom to top). Then for all κ, C_κ and $C_{\kappa+1}$ are not linked.*

Proof Again, the claim clearly holds for the empty stack. Assume that it is true after iteration $i-1$. By the induction hypothesis, the claim holds for every adjacent pair of components that remained in CS after popping the completed components. If the new component C that is pushed onto CS is linked with the component C' which is immediately below it, then the algorithm would have popped C' and merged it with C. Hence, the claim holds at the end of iteration i. ∎

Lemma 22 *If $CS = < C_1, C_2, \ldots >$ is as in Lemma 20 and $\{C_{i_1}, \ldots, C_{i_\kappa}\}$ is a maximal set of components in CS which belong to the same SCC of $\vec{B}_f$ such that $i_1 \leq \ldots \leq i_\kappa$ then $C_{i_{\kappa-1}}$ and C_{i_κ} are linked.*

Proof Assume that there is such a set of components $\{C_{i_1}, \ldots, C_{i_\kappa}\}$ in CS where $C_{i_\kappa - 1}$, C_{i_κ} are not linked.

Case 1: C_{i_κ} does not reach $C_{i_{\kappa-1}}$ by a single edge. Then it must reach a component C_{i_j} with

$i_j < i_{\kappa-1}$ by a single edge. This edge must be a $y \to z$ edge because otherwise convexity and Lemma 20 would imply that it also reaches $C_{i_{\kappa-1}}$ by an $x \to y$ edge, in contradiction to our assumption. Let z_γ be the target of this edge and y_β be its matching mate. Assume that β is maximal among β' such that $y_{\beta'}$ is in one of the components $C_{i_1}, \ldots, C_{i_{\kappa-2}}$ and it has a matching mate $z_{\gamma'}$ which is reachable from C_{i_κ} by a $y \to z$ edge. There is a path from z_γ to $C_{i_{\kappa-1}}$. If the path includes an $x \to y$ edge from a matching mate x_α of y_β to $y_{\beta'}$ with $\beta' > \beta$, then $\overline{D}(x_\alpha) > \beta$ and $\overline{D}(z_\gamma) > \beta$ so the algorithm in Figure 3.6 could not have matched x_α and z_γ with y_β, a contradiction. If the path includes a $y \to z$ edge from $y_{\beta'}$ with $\beta' \leq \beta$ to a node $z_{\gamma''}$ which is matched with $y_{\beta''}$ where $\beta'' > \beta$, then $z_{\gamma''}$ was in P_z when z_γ was extracted, so by convexity and Lemma 20 it is reachable from C_{i_κ} by a $y \to z$ edge, in contradiction to the maximality of β. We get that the path must go from z_γ to the left and then bypass y_β by an $x \to y$ edge $(x_{\alpha'}, y_{\beta''})$ such that $x_{\alpha'}$ is matched with $y_{\beta'}$ and $\beta' < \beta < \beta''$. We can assume w.l.o.g. that the path from z_γ to $x_{\alpha'}$ does not go to the left of $x_{\alpha'}$; if it does then one can show that either Lemma 21 is violated or there also exists a path that shortcuts the part that goes to the left of $x_{\alpha'}$. If the path ends with a $y \to z$ edge into a matching mate $z_{\gamma'}$ of $y_{\beta'}$ followed by the matching edges $(z_{\gamma'}, y_{\beta'}) \circ (y_{\beta'}, x_{\alpha'})$ then $\overline{D}(x_{\alpha'}) > \beta'$ and $\overline{D}(z_{\gamma'}) > \beta'$ so the algorithm in Figure 3.6 could not have matched $x_{\alpha'}$ and $z_{\gamma'}$ with $y_{\beta'}$, a contradiction. So the path ends with an $x \to y$ edge $(x_{\alpha'''}, y_{\beta'})$ followed by the edge $(y_{\beta'}, x_{\alpha'})$, where $x_{\alpha'''}$ is matched with $y_{\beta'''}$ for some $\beta' < \beta''' \leq \beta$. $x_{\alpha'''}$ was in P_x when $x_{\alpha'}$ was extracted, so it also reaches $y_{\beta''}$ by an $x \to y$ edge. By continuing backwards along the path and applying the same considerations, we get that there is an $x \to y$ edge from x_α to $y_{\beta''}$, hence again $\overline{D}(x_\alpha) > \beta$ and $\overline{D}(z_\gamma) > \beta$, so the algorithm in Figure 3.6 could not have matched x_α and z_γ with y_β, a contradiction.

Case 2: $C_{i_{\kappa-1}}$ does not reach C_{i_κ} by a single edge. Then C_{i_κ} is reached from another component C_{i_j} by an $x \to y$ edge $(x_\alpha, y_{\beta'})$. Assume that x_α is matched with y_β and that β is maximal among β'' such that $y_{\beta''}$ is in one of the components $\{C_{i_1}, \ldots, C_{i_{\kappa-2}}\}$ and has a matching mate $x_{\alpha''}$ which reaches C_{i_κ} by an $x \to y$ edge. There is a path from $C_{i_{\kappa-1}}$ to y_β. If there is an edge $(x_{\alpha''}, y_\beta)$ from $x_{\alpha''}$ which is matched with $y_{\beta''}$ for some $\beta < \beta'' < \beta'$, then by convexity and Lemma 20, $x_{\alpha''}$ reaches C_{i_κ} by an $x \to y$ edge, in contradiction to the maximality of β. If there is an edge $(y_{\beta''}, z_\gamma)$ such that z_γ is matched with y_β and $\beta'' > \beta$ then $\overline{D}(x_\alpha) > \beta$ and $\overline{D}(z_\gamma) > \beta$ so the algorithm in Figure 3.6 could not have matched x_α

and z_γ with y_β, a contradiction. We get that the path must bypass y_β by a $y \to z$ edge $(y_{\beta''}, z_{\gamma'''})$ such that $z_{\gamma'''}$ is matched with $y_{\beta'''}$ for some $\beta''' < \beta < \beta''$, and then return from $z_{\gamma'''}$ to y_β. With arguments similar to the ones used in case 1, we get that the path from $z_{\gamma'''}$ to y_β must consist of $y \to z$ edges, which implies that there is a $y \to z$ edge from $y_{\beta''}$ to z_γ, hence again $\overline{D}(x_\alpha) > \beta$ and $\overline{D}(z_\gamma) > \beta$ so the algorithm in Figure 3.6 could not have matched x_α and z_γ with y_β, a contradiction. ∎

Corollary 1 *Let $CS =< C_1, C_2, \ldots >$ and $\{C_{i_1}, \ldots, C_{i_\kappa}\}$ be as in Lemma 22. Then $C_{i_{\kappa-1}}$ and C_{i_κ} are either linked by $x \to y$ edges or linked by $y \to z$ edges.*

Proof Lemma 22 guarantees that $C_{i_{\kappa-1}}$ and C_{i_κ} are linked. Assume that these edges are not of the same type. That is, one is an $x \to y$ edge and the other is a $y \to z$ edge. Then by convexity and Lemma 20 we get that either $C_{i_{\kappa-1}}$ and $C_{i_{\kappa-1}+1}$ or $C_{i_\kappa-1}$ and C_{i_κ} are linked, and this contradicts Lemma 21. ∎

Lemma 23 *Let $CS =< C_1, C_2, \ldots >$ and $\{C_{i_1}, \ldots, C_{i_\kappa}\}$ be as in Lemma 22 and assume that there is at least one unexplored node which is in the same SCC of $\vec{B}_f$ as $\{C_{i_1}, \ldots, C_{i_\kappa}\}$. Then there is an edge from C_{i_κ} to an unexplored node.*

Proof Assume the converse. Then there is a component in $\{C_{i_1}, \ldots, C_{i_{\kappa-1}}\}$ which is connected by a single edge to an unexplored node. Let j be the maximal index such that $C_j \in \{C_{i_1}, \ldots, C_{i_{\kappa-1}}\}$ has an edge to an unexplored node. If this is a $y \to z$ edge then by convexity and Lemma 20 C_{i_κ} is also connected by a $y \to z$ edge to an unexplored node. So it must be an $x \to y$ edge. Let $x_j \in C_j$ be its source. With arguments similar to the ones used in the proof of Lemma 22 we can show that there cannot be a path from C_{i_κ} back to x_j, in contradiction to the assumption that C_{i_κ} and C_j are in the same SCC of $\vec{B}_f$. ∎

Corollary 2 *If $\{C_1, C_2, \ldots, C_{p_1}\}$ are the components found by the first pass of the algorithm and $\{C_{i_1}, \ldots, C_{i_\kappa}\}$ is a maximal subset of these components which are strongly connected between them such that $MinY[C_{i_1}] \leq \ldots \leq MinY[C_{i_\kappa}]$. Then*
(1) No component in $\{C_{i_1}, \ldots, C_{i_\kappa}\}$ is nested in another. That is, for all $j, j' \in \{i_1, \ldots, i_\kappa\}$

such that $j < j'$, $MaxY[C_j] < MinY[C_{j'}]$.
(2) $C_{i_\kappa - 1}$ and C_{i_κ} are linked. Furthermore, they are either linked by $x \rightarrow y$ edges or linked by $y \rightarrow z$ edges.

Proof Assume that there is a component C_j in the set which is nested in another component $C_{j'}$. Then $C_{j'}$ consists of nodes which are both to the left and to the right of C_j. This means that at some point, the algorithm merged these nodes into one component. At this point in time, the topmost component in CS consisted only of nodes which are to the left of C_j and it was merged with a component that contains only nodes to the right of C_j. By Lemma 20, this means that C_j was popped before this iteration, but this contradicts Lemma 23, so we have shown that (1) holds.
By Lemma 23, we know that none of $\{C_{i_1}, \ldots, C_{i_{\kappa-1}}\}$ were popped from CS before C_{i_κ} was pushed onto it. At a certain iteration, C_{i_κ} was pushed onto CS and stayed there at least until the next iteration. This, together with Corollary 1, implies (2). ∎

To sum up, the first pass partitions the nodes into components such that the components that compose an SCC of $\vec{B}_f$ are not nested within one another, and the two rightmost components of each SCC are linked by edges of the same type. The second pass of the algorithm merges components that belong to the same SCC. It starts with an empty stack CS and an empty list $SCCs$ and traverses the components found in the first pass by increasing order of $MinY[C]$. When considering a new component C, it first pops from CS all topmost components that cannot reach C or beyond it. For each such component C', it first checks if C' is linked with a component to its left by $x \rightarrow y$ edges. If so, there are components in CS that it needs to be merged with. So the algorithm pushes C' to a second stack $CSxy$. Otherwise, it checks if C' is linked with a component to its left by $y \rightarrow z$ edges and if so, pushes it to a third stack $CSyz$. Otherwise, it appends C' to $SCCs$ because C' is not linked with any component in CS and it does not reach unexplored components.

Before popping the next component, it checks whether the topmost component in CS and the topmost component in $CSxy$ are linked by $x \rightarrow y$ edges. If so, it pops each from its stack and merges them. It then repeatedly checks if the merged component is linked with the topmost component on CS. If so, the two are merged. Finally, the component which is the result of the merges is pushed back onto CS. The algorithm then checks whether the topmost component in CS and the topmost component in $CSyz$ are linked by $y \to z$ edges and if so, handles this in a similar way.

In the remaining part of this section we show that this algorithm finds the SCCs of $\vec{B}_f$. Denote the set of components found in the first pass of the algorithm by $Comp = \{C_1, C_2, \ldots, C_{p_1}\}$, such that for all $1 \leq i < p_1$, $MaxY[C_i] < MinY[C_{i+1}]$. Since each of these components is strongly connected but is not necessarily an SCC of $\vec{B}_f$, the components in $Comp$ are partitioned into sets of components such that the components of each set compose an SCC of $\vec{B}_f$.

Definition 3.3.1 *A subset $S = \{C_{i_1}, \ldots, C_{i_\kappa}\}$ of Comp is an* SCC set *if the union of the components in S is an SCC of $\vec{B}_f$. In the following we will use this notation while assuming that $MinY[C_{i_1}] < \ldots < MinY[C_{i_\kappa}]$.*

Definition 3.3.2 *Let $S_1 = \{C_{i_1}, \ldots, C_{i_\kappa}\}$ and $S_2 = \{C_{j_1}, \ldots, C_{j_{\kappa'}}\}$ be distinct SCC sets. Then S_1 is* nested *in S_2 if there exists $j \in \{j_1, \ldots, j_{\kappa'-1}\}$ such that for all $C_i \in S_1$, $MinY[C_j] < MinY[C_i] < MinY[C_{j+1}]$. S_1 and S_2 are* interleaved *if there exist $i, i' \in \{i_1, \ldots, i_{\kappa-1}\}$ and $j, j' \in \{j_1, \ldots, j_{\kappa'-1}\}$ such that $MinY[C_i] < MinY[C_j] < MinY[C_{i'}] < MinY[C_{j'}]$.*

The SCC sets of $Comp$ can be interleaved in one another, but in the following lemma we show that this can only occur in a restricted form. This will help us to show that the second pass of Algorithm 3.10 identifies all SCC sets in $Comp$.

Lemma 24 *Let $S_1 = \{C_{i_1}, \ldots, C_{i_\kappa}\}$ and $S_2 = \{C_{j_1}, \ldots, C_{j_{\kappa'}}\}$ be interleaved SCC sets. If $MinY[C_i] < MinY[C_j] < MinY[C_{i+1}] < MinY[C_{j+1}]$ then there are links between $S_i^l = \{C_1, \ldots, C_i\}$ and $S_i^h = \{C_{i+1}, \ldots, C_{i_\kappa}\}$ and there are links between $S_j^l = \{C_1, \ldots, C_j\}$ and $S_j^h = \{C_{j+1}, \ldots, C_{j_{\kappa'}}\}$. These links can be of one of two forms: (1) S_i^l and S_i^h are linked by $x \to y$ edges; S_j^l and S_j^h are linked by $y \to z$ edges. (2) S_i^l and S_i^h are linked by $y \to z$ edges; S_j^l and S_j^h are linked by $x \to y$ edges.*

Proof We show that any other option is not possible. If S_i^l and S_i^h are linked by an $x \to y$ edge from S_i^l to S_i^h and a $y \to z$ edge from S_i^h to S_i^l then by convexity and Lemma 20, S_i^l and S_j^l are also linked, which means that the components of S_1 and S_2 belong to the same

SCC of $\vec{B}_f$, contradicting the assumption that S_1 and S_2 are distinct SCC sets. The same follows if S_i^l and S_i^h are linked by a $y \to z$ edge from S_i^l to S_i^h and an $x \to y$ edge from S_i^h to S_i^l and if S_j^l and S_j^h are linked by an $x \to y$ edge in one direction and a $y \to z$ edge in the other.

Assume that each of the pairs S_i^l, S_i^h and S_j^l, S_j^h is linked by $x \to y$ edges in both directions. Then by convexity and Lemma 20, S_j^l and S_i^h are also linked by $x \to y$ edges and again the components of S_1 and S_2 are in the same SCC of $\vec{B}_f$. The same holds if the pairs S_i^l, S_i^h and S_j^l, S_j^h are both linked by $y \to z$ edges. ∎

Figure 3.11 shows an example of two interleaved SCC sets $S_1 = \{C_1, C_3, C_5\}$ and $S_2 = \{C_2, C_4\}$. For clarity, some of the edges were not drawn but the reader should assume that all edges that are implied by convexity exist in the graph. The SCCs of S_1 are linked by $x \to y$ edges and the SCCs of S_2 are linked by $y \to z$ edges.

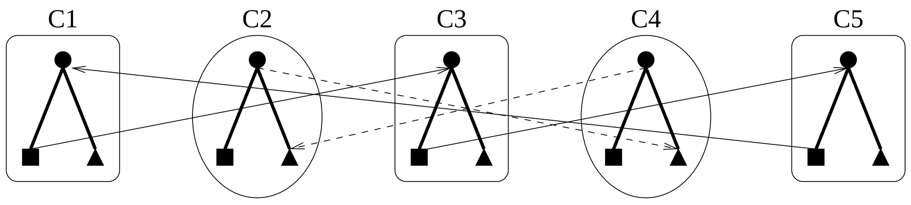


Figure 3.11: Two interleaved SCC sets.

The following resemble Lemmas 20 and 23 but refer to the second pass.

Lemma 25 *Let $CS =< C_1, C_2, \ldots >$ be the components in CS at the end of iteration i of the second pass (ordered from bottom to top). Then for all κ, $MaxY[C_\kappa] < MinY[C_{\kappa+1}]$.*

Proof At the beginning CS is empty and the claim clearly holds. Assume that the claim holds after iteration $i-1$ and consider the changes made to the stack during iteration i. When a topmost component of the stack is popped it does not affect the correctness of the claim. When a new component is pushed onto the stack it is the result of merging some of the topmost components in the stack with components from the temporary stacks $CSxy$ and $CSyz$. Since the temporary stacks contain only components with higher y nodes than CS, the merged component contains indices which are all higher than what is in components which are below it in CS. ∎

Lemma 26 *Let $S = \{C_{i_1}, \ldots, C_{i_\kappa}\}$ be an SCC set. Then for all $\ell \in \{1, \ldots, \kappa - 1\}$, C_{i_ℓ} reaches at least one of $\{C_{i_{\ell+1}}, \ldots, C_{i_\kappa}\}$ by a single edge.*

Proof By Lemmas 20 and 25, the components of S appear in CS in the second pass in the same order in which they appear in the first pass. This implies that after a certain iteration of the second pass, $\{C_{i_1}, \ldots, C_{i_\ell}\}$ are in CS and $\{C_{i_{\ell+1}}, \ldots, C_{i_\kappa}\}$ are unexplored. Since there is an edge from one of $\{C_{i_1}, \ldots, C_{i_\ell}\}$ to one of $\{C_{i_{\ell+1}}, \ldots, C_{i_\kappa}\}$, we can show the claim by arguments which are similar to the ones used in the proof of Lemma 23. ∎

We can now show that the second pass of Algorithm 3.10 identifies the SCCs of $\vec{B}_f$.

Lemma 27 *Let $S = \{C_{i_1}, \ldots, C_{i_\kappa}\}$ be an SCC set. Then in the second pass of the algorithm, the components of S will be merged.*

Proof Let S be an SCC set such that all SCC sets which are nested in S were merged. We show by induction that all components of S are merged in the second pass. That is, we show that for all i from $i_{\kappa-1}$ to i_1, $C_i, \ldots, C_{i_\kappa}$ will be merged. For $i = i_{\kappa-1}$, we know by Corollary 2 that $C_{i_{\kappa-1}}$ and C_{i_κ} are linked by either $x \to y$ edges in both directions (case 1) or $y \to z$ edges in both directions (case 2). By Lemma 26, we know that $C_{i_{\kappa-1}}$ is not popped from CS before C_{i_κ} is pushed onto it. When C_{i_κ} is popped from CS for the first time, it is pushed onto the stack CS' where CS' is $CSxy$ (case 1) or $CSyz$ (case 2). Assume that when $C_{i_{\kappa-1}}$ became the topmost component in CS, C_{i_κ} was not the topmost component in CS'. If it was popped from CS' before that time, this is because a component above $C_{i_{\kappa-1}}$ in CS is linked with it. Since this component is not in S, this contradicts the assumption that S is an SCC set. On the other hand, if there was another component C' above C_{i_κ} in CS', then by Lemma 24 this is because C' is linked by $x \to y$ edges (case 1) or $y \to z$ edges (case 2) with another component C'' which was above $C_{i_{\kappa-1}}$ in CS. If C' and C'' were not merged, we get that there must have been a component above C' in CS' when C'' was the topmost component on CS. Applying the same argument recursively, we get that the number of components is infinite. Hence, C' and C'' were merged before $C_{i_{\kappa-1}}$ was the topmost component in CS, so C' could not have been above C_{i_κ} in CS'.

Assume that $C_{i_j+1}, \ldots, C_{i_\kappa}$ were merged by the algorithm into a larger component C. By Lemma 26 we know that C_{i_j} reaches C by a single edge and this implies that it was not popped from CS before any of $C_{i_j+1}, \ldots, C_{i_\kappa}$. In addition, since C and C_{i_j} are in the same SCC, there is a path from C to C_{i_j}. This path does not go through components that are between C and C_{i_j} in the stack because that would place these components in the same SCC as C and C_{i_j}, contradicting the assumption that S is an SCC set. Assume that the path begins with an $x \to y$ edge from C to C_{i_j} or a component below it in CS. Then by convexity there is also an $x \to y$ edge from C to C_{i_j}. If the edge from C_{i_j} to C is a $y \to z$ edge then by convexity and Lemma 25, C is linked with C_{i_j} and all components which are above it in the stack and will be merged with them by the algorithm. If this edge is an $x \to y$ edge then C will be pushed onto $CSxy$ and as in the base case, it will be merged with C_{i_j} when the later will become the topmost component in CS.

If the path from C to C_{i_j} begins with a $y \to z$ edge from C to a component below C_{i_j} then arguments similar to the ones used in the proof of Lemma 22 imply that C also reaches C_{i_j} by a $y \to z$ edge. If the edge from C_{i_j} to C is an $x \to y$ edge then again by convexity and Lemma 25 C_{i_j} and all components above it in CS are linked with C and will be merged with it. If it is a $y \to z$ edge then C will be pushed onto $CSyz$ and will be merged with C_{i_j} when the later will become the topmost component in CS. ∎

3.3.3 Narrowing the bounds

After the SCC computation we have for each node in $X \cup Z$ the index of its y mate and for each $y_i \in Y$ the component it belongs to. As in [11], we can in time $O(n + n')$ obtain for every component a list of its y nodes sorted by index and two lists of its x and z nodes, one sorted by the lower endpoints and one by the upper endpoints of their domains. Let C be a component, let $I_C = (i_1, \ldots, i_k)$ be the sorted list of the indices of its y nodes and let $\overline{D}_C = (\overline{D}_{j_1}, \ldots, \overline{D}_{j_\ell})$ be the sorted list of the upper domain endpoints of its x nodes. Then the upper endpoint of the narrowed domain of $x_j \in C$ is i_k where $i_k \leq \overline{D}_j < i_{k+1}$ (with $i_{k+1} = n' + 1$) and this value can be computed for all x nodes in C with one simultaneous scan of I_C and $\overline{D}_C$. Narrowing the lower endpoints of the domains of the x nodes and both endpoints of the domains of the z nodes can be done in a similar manner with the relevant lists.

3.3.4 Implementation details

Figure 3.12 shows how the shortcuts that are used in the pseudo-code in Figure 3.10 can be implemented with linear-time preprocessing. After finding the parity matching and before beginning the SCC algorithm, we compute for each $y_i \in Y$ several values:

$D^x[i] = [\underline{D^x}[i], \overline{D^x}[i]]$ is the union of the domains of the x nodes which are matched with y_i. Similarly, $D^z[i] = [\underline{D^z}[i], \overline{D^z}[i]]$ is the union of the domains of the z nodes which are matched with y_i. Both can be computed in linear time. For the $D^x[i]$'s we traverse the x nodes and for each x_j, set $\underline{D^x}[i] = \min\{\underline{D}(x_j), \underline{D^x}[i]\}$ and $\overline{D^x}[i] = \max\{\overline{D}(x_j), \overline{D^x}[i]\}$ where y_i is the value node that x_j is matched with. The computation of the $D^z[i]$'s is similar.

$MinYZLink[i]$ ($MaxYZLink[i]$) is the minimum (maximum) index i' of a y node which is matched with a node $z \in Z$ such that $i \in Dom(z)$, i.e., there is an edge from y_i to a matching mate of $y_{i'}$. The $MinYZLink[i]$'s can be computed in linear time as follows: Traverse the y nodes from left to right. For each y_i and for each $i' \in D^z(y_i)$, $MinYZLink[i']$ should be set to i unless it was already set to a smaller value. Since the $D^z(y_i)$'s are intervals and each $i \in D^z(y_i)$, it suffices to maintain the index of the largest i' for which $MinYZLink[i']$ was set and to traverse each $D^z(y_i)$ beginning at this index. The computation of $MaxYZLink[i]$ is symmetric.

Finally, $MaxLeftXYLink[i]$ is the maximum index $i' < i$ such that $y_{i'}$ is matched with some $x_j \in X$ with $i \in Dom(x_j)$ (if no such $y_{i'}$ exists then $MaxLeftXYLink[i] = n' + 1$, i.e., infinity). Symmetrically, $MinRightXYLink[i]$ is the minimum index $i' > i$ with the same property, and if none exists then $MaxLeftXYLink[i] = -1$, i.e., minus infinity. We begin by initializing $MaxLeftXYLink[i] = n' + 1$ for all i. For $i = 1$, this value remains and for higher i's $MaxLeftXYLink[i]$ can be computed as follows. Push y_1 onto a stack and then traverse the y nodes from y_2 to $y_{n'}$. When processing y_i, first pop from the stack all topmost nodes $y_{i'}$ such that $\overline{D^x}[i'] < i$. If the stack is not empty afterwards, then $MaxLeftXYLink[i]$ is the topmost node on the stack; this is the latest node that was pushed and has a matching mate $x \in X$ whose domain contains i. Last, y_i is pushed onto the stack. A symmetric algorithm can compute $MinRightXYLink[i]$ in linear time.

Once these values are known for each individual y node, the SCC algorithm can maintain them for every component: When a new component is created it contains a single y node and its matching mates, and when two components are merged, each value can be updated in constant time for the merged component. Similarly, it can maintain for each component C the value $MinY[C]$ ($MaxY[C]$), which is the smallest (largest) index of a y node in C. The shortcuts in the pseudocode can then be implemented as follows:

ReachesRight$[C]$ is $\max\{MaxYZLink[C], \overline{D^x}[C]\}$.

xyLeftLinks$[C]$ is true iff $\underline{D^x}[C] \leq MaxLeftXYLink[C] < MinY[C]$, which means that C is reached by an $x \rightarrow y$ edge from a component C' to its left and it also reaches C' by an $x \rightarrow y$ edge.

yzLeftLinks$[C]$ is true iff (1) $\underline{D^z}[C] < MinY[C]$ and (2) $\underline{D^z}[C] \leq MinYZLink[C]$, which means that (1) C is reached by a $y \rightarrow z$ edge from at least one component to its left and (2) there is a component C' for which (1) holds and which is also reached from C by a $y \rightarrow z$ edge.

xyLinks$[C, C']$, where C is to the left of C', is true iff (1) $\overline{D^x}[C] \geq MinY[C']$ and (2) $\underline{D^x}[C'] \leq MaxY[C]$, which means that (1) C reaches C' by an $x \rightarrow y$ edge and (2) C' reaches C by an $x \rightarrow y$ edge. Similarly, *yzLinks*$[C, C']$ is true iff $\overline{D^z}[C] \geq MinY[C']$ and $\underline{D^z}[C'] \leq MaxY[C]$.

Last, *Linked*$[C, C']$ is true iff $Reaches[C, C'] \wedge Reaches[C', C]$, where $Reaches[C_1, C_2]$ is true iff (1) $\overline{D^x}[C_1] \geq MinY[C_2]$ or (2) $\underline{D^z}[C_2] \leq MaxY[C_1]$, which means that (1) there is an $x \rightarrow y$ edge from C_1 to C_2 or (2) there is a $y \rightarrow z$ edge from C_1 to C_2.

3.3.5 Complexity analysis

The parity matching algorithm performs $2n$ *Insert* and *ExtractMin* operations on the priority queues while it traverses the y nodes. This takes time $O(n' + n \log n)$. In the SCC computation everything takes linear time except one thing: Maintaining the component list. We wish to be able to merge components and to find which component a node belongs to. For this we use a Union-Find[9] data structure over the y nodes. We perform $O(n')$ operations on this structure, which in total take time $O(n'\alpha(n', n'))$. We get that the total running time of the algorithm is $O(n'\alpha(n', n') + n \log n)$.

```
for $i = 1$ to $n'$ do
    $\underline{D^x}[i] \leftarrow i$; $\overline{D^x}[i] \leftarrow i$; $\underline{D^z}[i] \leftarrow i$; $\overline{D^z}[i] \leftarrow i$
    $MaxYZLink[i] \leftarrow i$; $MinYZLink[i] \leftarrow i$
    $MinRightXYLink[i] \leftarrow -1$; $MaxLeftXYLink[i] \leftarrow n' + 1$
endfor
for $j = 1$ to $n$ do
    $i \leftarrow \mathrm{YMate}(x_j)$; $\underline{D^x}[i] \leftarrow \min\{\underline{D^x}[i], \underline{D}(x_j)\}$; $\overline{D^x}[i] \leftarrow \max\{\overline{D^x}[i], \overline{D}(x_j)\}$
    $i \leftarrow \mathrm{YMate}(z_j)$; $\underline{D^z}[i] \leftarrow \min\{\underline{D^z}[i], \underline{D}(z_j)\}$; $\overline{D^z}[i] \leftarrow \max\{\overline{D^z}[i], \overline{D}(z_j)\}$
endfor
$MinYZLink[1] \leftarrow 1$
$i \leftarrow 2$
for Yindex $= 1$ to $n'$ do
    while $i \geq \overline{D^z}[\mathrm{Yindex}]$ do $MinYZLink[i] \leftarrow$ Yindex; $i \leftarrow i + 1$
endfor
$MaxYZLink[n'] \leftarrow n'$
$i \leftarrow n' - 1$
for Yindex $= n'$ to $1$ do
    while $i \geq \underline{D^z}[\mathrm{Yindex}]$ do $MaxYZLink[i] \leftarrow$ Yindex; $i \leftarrow i - 1$
endfor
$S \leftarrow []$ (* Empty stack *)
$S$.Push 1
for $i = 2$ to $n'$ do
    while $S$.NotEmpty $\wedge$ $\overline{D^x}[S.\mathrm{Top}] < i$ do $S$.Pop
    if $S$.NotEmpty then $MaxLeftXYLink[i] \leftarrow S$.Top
    $S$.Push $i$
endfor
$S \leftarrow []$ (* Empty stack *)
$S$.Push $n'$
for $i = n' - 1$ to $1$ do
    while $S$.NotEmpty $\wedge$ $\underline{D^x}[S.\mathrm{Top}] > i$ do $S$.Pop
    if $S$.NotEmpty then $MinRightXYLink[i] \leftarrow S$.Top
    $S$.Push $i$
endfor
```

Figure 3.12: Preprocessing stage for the algorithm in Figure 3.10

3.4 The *UsedBy* constraint

In this section we show algorithms that achieve arc consistency and bound consistency for the general case of the *UsedBy* constraint. The simplest solution is to add $n - m$ dummy variables to Z with domains that include all values: $\{y_1, \ldots, y_{n'}\}$ in the arc consistency case and $[y_1, y_{n'}]$ in the bound consistency case. Let $\overline{Z}$ be the set obtained by padding Z in this way. Then we have that $|X| = |\overline{Z}| = n$ and we can apply the algorithms for the *Same* constraint. The correspondence between solutions to $Same(X, \overline{Z})$ and solutions to $UsedBy(X, Z)$ should be obvious.

In the rest of this section we show how the *UsedBy* constraint can be solved directly. While these algorithms are not asymptotically faster than what we get with the padding method described above, they should be faster in practice.

As in the case of the *Same* constraint, we construct the bipartite graph B and look for a solution in it. This time, a solution is a *z-parity matching*, defined as follows.

Definition 3.4.1 *Let $M \subseteq E$ be a set of edges of B. For any node $v \in X \cup Y \cup Z$, let $N_M(v)$ be the set of nodes which are neighbors of v in $B' = (X \cup Z, Y, M)$. We say that M is a z-*parity matching* in B iff $\forall_{v \in X \cup Z} |N_M(v)| = 1$ and $\forall_{y \in Y} |N_M(y) \cap X| \geq |N_M(y) \cap Z|$.*

The following is a simple extention of Lemma 16.

Lemma 28 *There is a one to one correspondence between the solutions to the UsedBy constraint and the z-parity matchings in B.*

3.4.1 Arc consistency

As with the *Same* constraint, the arc consistency algorithm builds the directed graph $\vec{B}$ and finds an s-t flow of value m in it. The edges participating in this flow match all nodes of Z and an equal number of nodes from X, such that every z node corresponds to an x node which is assigned the same value. To get a z-parity matching M, we take the set of edges that participate in the flow and an edge connecting each of the remaining x nodes with an arbitrary value in its domain. We then construct the residual graph $\vec{B}_f$. As in the case of the *Same* constraint, $\vec{B}_f$ contains all edges in their original orientation and the matching edges also in reverse direction. Finally, we add a node s with an edge (y_i, s) for each value node $y_i \in Y$ and an edge (s, y_i) for each $y_i \in Y$ such that $|N_M(y_i) \cap X| > |N_M(y_i) \cap Z|$ (see Figure 3.13).

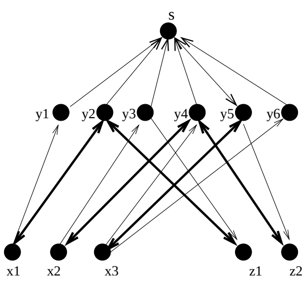


Figure 3.13: Example of a residual graph with the added node s.

The purpose of adding the node s is as follows. Let $y_i \in Y$ be such that $|N_M(y_i) \cap X| > |N_M(y_i) \cap Z|$ and let $x_j \in N_M(y_i) \cap X$. By matching x_j with any of its neighbors $y_{i'}$ with $i' \neq i$, we get an alternative z-parity-matching. More generally, let M be a z-parity matching obtained as described above and let P be an alternating path that begins at a node y_i with $|N_M(y_i) \cap X| > |N_M(y_i) \cap Z|$ and ends at another node $y_{i'} \in Y$. Then $M \oplus P$ is also a z-parity matching. By construction, there is an edge from s to y_i and an edge from $y_{i'}$ to s, so all edges on this path belong to the same SCC of $\vec{B}_f$. We prove this in the following lemma.

Lemma 29 *An edge $e = (u, v) \in \vec{B}_f$ is consistent iff u and v belong to the same SCC.*

Proof Let M be the z-parity matching that was obtained from the flow f as described above. If $e \in M$ then e participates in the solution that corresponds to the matching M and is therefore consistent. u and v are connected by two antiparallel edges so they are in the same SCC. Assume that $e \notin M$ and that u and v are in the same SCC. Then there is a cycle P which uses e in $\vec{B}_f$. If the cycle does not contain the special node s then as in Lemma 17, $M' = M \oplus P$ is a z-parity matching. If the cycle does contain the node s, consider the path $P' = P \setminus s$ which is the cycle P without s and the edges adjacent to it. P starts with an edge (y_i, v) from a node $y_i \in Y$ such that $|N_M(y_i) \cap X| > |N_M(y_i) \cap Z|$ and ends with an edge (u, y_j) into another node $y_j \in Y$. We show that $M' = M \oplus P'$ is a z-parity-matching. For an internal node (a node which is different from y_i and y_j), the situation is as in Lemma 17 so all nodes preserve the properties required of a z-parity-matching.

If $v \in X$ then the edge (y_i, v) is in M because otherwise it cannot be traversed in this direction. y_i is matched in M' with one X node less than in M, but this is fine because in M it is matched with more X than Z nodes. If $v \in Z$ then the edge may or may not be in M. In both cases we get that in M' y_i is matched with at least as many X nodes as Z nodes.

If $u \in Z$ then the edge (u, y_j) is in M and then y_j is matched in M' with less Z nodes and the same number of X nodes compared to M. If $u \in X$ then it is not in M, because there must be another edge incoming into u on the path and this must be the single edge of M which is adjacent to u. We get once again that y_j is matched in M' with one more X node than in M. So M' is a z-parity-matching.
The proof that an edge e which is not in an SCC of $\vec{B}_f$ is not consistent is identical to the corresponding part of Lemma 17. ▮

3.4.2 Bound consistency

The algorithm in Figure 3.14 finds a z-parity matching in a convex graph. As the parity matching algorithm of Figure 3.6, it traverses the nodes of Y from left to right. In iteration i it first inserts into the queue P_x (P_z) all nodes of X (Z) whose domains start at i, since they become matching candidates. It then pops from P_z all nodes whose domains end at i and matches each of them with y_i along with a node from P_x. If P_x becomes empty during this process, then not all of these z nodes can be matched so the algorithm reports failure. Next, it pops from P_x all nodes whose domains end at i and matches them with y_i. For each such x node, if P_z is not empty then a node is popped from it and is matched with y_i as well. If all n' iterations complete without reporting failure, the algorithm reports success.

Lemma 30 *If there is a z-parity matching in B then the algorithm in Figure 3.14 finds one.*

Proof The proof of this lemma is similar to the proof of Lemma 18. There are two cases that need to be added. The first is that y_i was matched by the algorithm with a node x_j and without a corresponding z node. This implies that $\overline{D}(x_j) = i$ and P_z was empty when x_j was extracted. Hence, in M_{i-1} x_j must also be matched with y_i and it does not have a corresponding z node. The second case is that y_i was matched in M_{i-1} with x_j and without a corresponding z node, and y_i was not matched with x_j by the algorithm. Then $\overline{D}(x_j) > i$

and we change M_{i-1} by matching x_j with y_{i+1} instead of y_i. ∎

Lemma 31 *If the algorithm in Figure 3.14 reports success then it constructs a z-parity matching in B.*

Proof If the algorithm reports success then P_x and P_z are empty at the end, which means that all nodes in $X \cup Z$ were extracted and matched with y nodes. In addition, since the algorithm did not report failure during the extractions, whenever $v \in X \cup Z$ was matched with y_i, we have that $\underline{D}(v) \leq i \leq \overline{D}(v)$. For all $1 \leq i \leq n'$, whenever y_i is matched with some node from Z it is also matched with a node from X. Hence, we get that the matching that was constructed is a z-parity matching. ∎

After finding a z-parity matching in B, we wish to find the SCCs of $\vec{B}_f$. We first find the SCCs of $\vec{B}_f \setminus \{s\}$, which is the graph $\vec{B}_f$ without the node s and the edges adjacent to it. For this we can use the algorithm in Figure 3.10; the fact that a y node may be matched with more x nodes than z nodes does not affect the algorithm or the proofs.

We then merge the SCCs of $\vec{B}_f \setminus \{s\}$ into the SCCs of $\vec{B}_f$. The general idea follows the one used in [11]: For each SCC of $\vec{B}_f \setminus \{s\}$ we determine whether it reaches s and whether it is reached from s. We then merge all components which reach s and are reached from s into one component. Unlike in the *GCC* case, all y nodes in the *UsedBy* graph reach s by a single edge so it only remains to determine which components are reached from s.

Definition 3.4.2 *The* index $I(v)$ *of a node* $v \in X \cup Y \cup Z$ *is defined as follows. For* $y_i \in Y$, $I(y_i) = i$. *For* $v \in X \cup Z$, $I(v) = i$ *where* y_i *is the matching mate of* v.

Definition 3.4.3 *Let* $P =< v_1, v_2, \ldots, v_p >$ *be a simple path in* $\vec{B}_f \setminus \{s\}$ *and for each* v_i *let* $Comp[v_i]$ *be the SCC of* $\vec{B}_f \setminus \{s\}$ *that* v_i *belongs to. The* signature $S(P)$ *of* P *is the sequence of components visited by* P. *That is,* $S(P) =< Comp[v_1], Comp[v_2], \ldots, Comp[v_p] >$.

Assume that a path visits a node v, proceeds to a node u with $Comp[u] \neq Comp[v]$ and later visits a node w with $Comp[w] = Comp[v]$. Since v and w are in the same SCC, there is a path from v to w which visits nodes only inside this SCC. Hence, we will assume w.l.o.g. that a path does not return to a component that it has already visited and left. This means

```
(* Assumption: $X$ and $Z$ are sorted according to $\underline{D}$. *)
$P_x \leftarrow []$ (* priority queue containing $x$ nodes sorted by $\overline{D}$ *)
$P_z \leftarrow []$ (* priority queue containing $z$ nodes sorted by $\overline{D}$ *)
$j \leftarrow 0$
for $i = 1$ to $n'$ do
    forall $x_h$ with $\underline{D}(x_h) = i$ do $P_x.\mathit{Insert}\ x_h$
    forall $z_h$ with $\underline{D}(z_h) = i$ do $P_z.\mathit{Insert}\ z_h$
    (* Assume that $\mathit{MinPriority}$ of an empty queue is $\infty$ *)
    while $P_z.\mathit{MinPriority} = i$ do
        if $P_x$.IsEmpty then report failure
        $x \leftarrow P_x.\mathit{ExtractMin}$; match $x$ with $y_i$
        $z \leftarrow P_z.\mathit{ExtractMin}$; match $z$ with $y_i$
    end while
    while $P_x.\mathit{MinPriority} = i$ do
        $x \leftarrow P_x.\mathit{ExtractMin}$; match $x$ with $y_i$
        if $P_z$.NotEmpty then
            $z \leftarrow P_z.\mathit{ExtractMin}$; match $z$ with $y_i$
        endif
    end while
endfor
report success
```

Figure 3.14: Algorithm to find a z-parity matching in a convex graph.

that in the signature of a path, $Comp[v_i] = Comp[v_j]$ implies $Comp[v_\kappa] = Comp[v_i]$ for all $i < \kappa < j$.

Definition 3.4.4 *Let $P =< v_1, v_2, \ldots, v_p >$ be a simple path in $\vec{B}_f \setminus \{s\}$ and for each v_i let v'_i be the first node on P such that $Comp[v_i] = Comp[v'_i]$. Then P is* left-monotonous *if $I(v'_\kappa) \geq I(v'_{\kappa+1})$ for all $1 \leq \kappa < p$. It is* right-monotonous *if $I(v'_\kappa) \leq I(v'_{\kappa+1})$ for all $1 \leq \kappa < p$.*

We next show that if there is a path from a component C to another component C' then there is also a path from C to C' which is either left-monotonous or right-monotonous.

Lemma 32 *If there is a path $P =< v_1, v_2, \ldots, v_p >$ then there is also a path $P' =< u_1, u_2, \ldots, u_{p'} >$ such that $u_1 = v_1$, $u_{p'} = v_p$ and P' is either left-monotonous or right-monotonous.*

Proof By induction on p. For $p = 1$, the claim clearly holds with $P' = P$. Assume that it holds for all $\bar{p} < p$ but not for p. Let $P =< v_1, v_2, \ldots, v_p >$ be a path. By the induction hypothesis, there is a path $P'' =< u_2, \ldots, u_{p'} >$ such that $u_2 = v_2$, $u_{p'} = v_p$ and P'' is monotonous. Assume w.l.o.g. that it is left-monotonous (the right-monotonous case is symmetric). If P'' goes through a node u_i with $Comp[u_i] = Comp[v_1]$, then since there is a path Q from v_1 to u_i which lies in the same SCC, the claim holds with $P' = Q \circ < u_i, \ldots, u_{p'} >$. Otherwise, if $I(v_1) > I(v_2)$ then since $v_2 = u_2$ the claim holds with $P' =< v_1, u_2, \ldots, u_{p'} >$, which is left-monotonous.

If, on the other hand, $I(v_1) < I(v_2)$ then there are two cases. The first is that $I(v_1) < I(u_{p'})$. If the edge from v_1 to v_2 is an $x \to y$ edge then by convexity there is also an $x \to y$ edge from v_1 to all components on the path, and in particular to $Comp[u_{p'}]$. Then the claim holds with $P' =< v_1, y_{I(u_{p'})} >$, which is right-monotonous. If the edge from v_1 to v_2 is a $y \to z$ edge then by convexity there is also a $y \to z$ edge from $y_{I(u_{p'})}$ to v_2, which means that $v_2, \ldots, u_{p'}$ are on a cycle and hence belong to one SCC. Again, the claim holds with the right-monotonous path $P' =< v_1, y_{I(u_{p'})} >$. The second case is that $I(u_{i+1}) < I(v_1) < I(u_i)$ for some $1 < i < p$. Since there is a path from v_1 to u_i and we assume that $Comp[v_1] \neq Comp[u_i]$, the edge from u_i to u_{i+1} is a $y \to z$ edge, because otherwise convexity would imply that there is an $x \to y$ edge from u_{i+1} to v_1, placing them in the same SCC. Hence, by convexity there is also a $y \to z$

edge from $y_{I(v_1)}$ to u_{i+1}, so the claim holds with $P' =< v_1, \ldots, y_{I(v_1)}, u_{i+1}, \ldots, u_{p'} >$. ∎

As a corollary, we get that we can find all components that are reached from s with three passes over the y nodes as shown in Figure 3.15. The first pass identifies the components that are reached from s by a single edge. The second pass traverses the y nodes from right to left and identifies the components that are reached from s by a left-monotonous path originating in a component that was marked in the first pass. The third pass is symmetric to the second. It traverses the y nodes from left to right and finds the components that are reached from s by a right-monotonous path originating at a component that was marked in one of the previous passes. In the following, when we speak of iteration i' of any pass of the algorithm, we mean the iteration in which $i = i'$, and not the ith iteration.

Lemma 33 *The algorithm in Figure 3.15 correctly marks the components that are reached from s.*

Proof Let C be a component that is reached from s. By Lemma 32, we know that C is also reached from s by a left-monotonous or right-monotonous path. Let p be the length of this path and $v \in C$ be the first node on it. If $p = 1$ then there is an edge (s, y_i) to some $y_i \in C$ and C will be marked in iteration i of the first pass.

For $p > 1$, if the last edge on the path is within the same component, then this component is reached by the path of length $p - 1$ that is obtained by discarding this edge. Then our claim follows by the induction hypothesis. We can therefore assume that the last edge on the path is between two distinct components, and in particular that it is not in M.

There are two cases to consider. The first is that the path is left-monotonous. We show by induction on p that it will be marked not later than in iteration $I(v)$ of the second pass. If the last edge on the path is a $y \to z$ edge (y_i, z_j) then $Comp[i]$ is reached from s by a left-monotonous path of length $p - 1$ and by the induction hypothesis, it will be marked not later than during iteration i of the second pass. Then, $min\_y\_reached\_from\_s$ will be set to $\underline{D^x}[Comp[i]] \leq i$. In iteration $I(z_j)$, $\overline{D^z}[C] \geq i \geq min\_y\_reached\_from\_s$ so C will be marked.

If the last edge on the path is an $x \to y$ edge (x_j, y_i), then by the induction hypothesis $Comp[I(x_j)]$ will be marked by the end of iteration $I(x_j)$ of the second pass and then *min_y_reached_from_s* will be set to at least $\underline{D^x}(x_j) \leq i$. In iteration i, we have *min_y_reached_from_s* $\leq i$ so C will be marked.

The second case is that C is reached from s by a right-monotonous path. We can symmetrically show by induction that it will be marked in the third pass of the algorithm. ∎

3.5 Future directions

We have shown that fast flow-based bound consistency algorithms can be designed when complex variable-value graphs are involved. We are working on extending these ideas to solve variations on the *Same* and *UsedBy* constraints. For example, a *GCC*-like restriction on the *Same* constraint: We still require that the multiset of values assigned to $\{x_1, \ldots, x_n\}$ is the same as the multiset of values assigned to $\{z_1, \ldots, z_n\}$, and in addition we have for each value i a lower bound and an upper bound on the number of variables in $\{x_1, \ldots, x_n\}$ (and hence also $\{z_1, \ldots, z_n\}$) that can be assigned the value i.

```
(* Pass 1: Mark components that are reached from $s$ by a single edge *)
for $i = 1$ to $n'$ do
    if $\mathit{Reached\_from\_s}[i]$ then
        $\mathit{Reached\_from\_s}[\mathit{Comp}[i]] \leftarrow$ true
    endif
endfor
(* Pass 2: Mark components that are reached from $s$ by a left-monotonous path *)
$\mathit{min\_y\_reached\_from\_s} \leftarrow n' + 1$
for $i = n'$ to 1 do
    $C \leftarrow \mathit{Comp}[i]$
    if $\mathit{min\_y\_reached\_from\_s} \leq i$ then
        $\mathit{Reached\_from\_s}[C] \leftarrow$ true
    endif
    if $\overline{D^z}[C] \geq \mathit{min\_y\_reached\_from\_s}$ then
        $\mathit{Reached\_from\_s}[C] \leftarrow$ true
    endif
    if $\mathit{Reached\_from\_s}[C]$ then
        $\mathit{min\_y\_reached\_from\_s} \leftarrow \underline{D^x}[C]$
    endif
endfor
(* Pass 3: Mark components that reach $s$ by a right-monotonous path *)
$\mathit{max\_y\_reached\_from\_s} \leftarrow 0$
for $i = 1$ to $n'$ do
    $C \leftarrow \mathit{Comp}[i]$
    if $\mathit{max\_y\_reached\_from\_s} \geq i$ then
        $\mathit{Reached\_from\_s}[C] \leftarrow$ true
    endif
    if $\underline{D^z}[C] \leq \mathit{max\_y\_reached\_from\_s}$ then
        $\mathit{Reached\_from\_s}[C] \leftarrow$ true
    endif
    if $\mathit{Reached\_from\_s}[C]$ then
        $\mathit{max\_y\_reached\_from\_s} \leftarrow \overline{D^x}[C]$
    endif
endfor
```

Figure 3.15: Algorithm to mark the components that reach s.

Chapter 4

Online Topological Ordering

In this chapter, we study online algorithms to maintain a topological ordering of a directed acyclic graph. In a topological ordering, each node $v \in V$ of a given directed acyclic graph (DAG) $G = (V, E)$ is associated with a value $ord(v)$, such that for each directed edge $(u, v) \in E$, we have $ord(u) < ord(v)$ [29]. In the online variant of the topological ordering problem, the edges of the DAG are not known in advance but are given one at a time. Each time an edge is added to the DAG, we are required to update the mapping ord. One directly observes that there are two cases when an edge is added. Suppose we have a DAG and a valid topological ordering function ord, and that an edge (x, y) is added to this DAG. If $ord(x) < ord(y)$ then ord is a valid topological ordering function for the new DAG and no updating is necessary. Otherwise, the new edge (x, y) is said to *violate* the topological order ord. In this case, we need to find a new function ord' which is a valid topological ordering function for the new DAG.

The online topological ordering problem has several applications. It has been studied in the context of compilation [14, 18] where dependencies between modules are maintained to reduce the amount of recompilation performed when an update occurs, source code analysis [19] where the aim is to statically determine the smallest possible target set for all pointers in a program and as a subroutine of an algorithm for incremental evaluation of computational circuits [2].

4.0.1 Known Results

The trivial solution for the online problem is to compute a new topological order from scratch whenever an edge is inserted. Since this takes time $\Theta(n+m)$ for an n-node m-edge DAG, the complexity of inserting m edges is $\Theta(m^2)$[1].

Marchetti-Spaccamela et al. [15] gave an algorithm that can insert m edges in $O(mn)$ time, giving an amortized time bound of $O(n)$ per edge instead of the trivial $O(m)$. This is the best amortized result known so far. Pearce and Kelly [19] propose a different algorithm and show experimentally that it achieves a speedup on sparse inputs, although its worst case running time is slightly higher. It was shown [12] that the algorithm by Pearce and Kelly is worst-case optimal in terms of the number of node relabelling operations it performs.

Alpern et al. [2] designed an algorithm which runs in time $O(\|\delta\| \log \|\delta\|)$ in the *bounded incremental computation* model [24]. In this model, the parameter $\|\delta\|$ measures, upon an edge insertion, the size (number of nodes and edges) of a minimal subgraph that we need to update in order to obtain a valid topological order ord' for the modified graph. Since $\|\delta\|$ can be anywhere between 0 and $\Theta(m)$, the bounded complexity result does not provide much information about the cost of a sequence of updates. All it guarantees is that each individual update could not have been performed much faster.

The only non-trivial lower bound for online topological ordering is due to Ramalingam and Rep [23], who show that an adversary can force any algorithm to perform $\Omega(n \log n)$ node relabelling operations while inserting $n-1$ edges (and creating a chain).

4.0.2 Our Results

In Section 4.1 we show that a slight variation on the algorithm by Alpern et al. can handle m edge insertions in $O(\min\{m^{3/2}\log n, m^{3/2} + n^2 \log n\})$ time. This implies an amortized time of $O(\min\{\sqrt{m}\log n, \sqrt{m} + \frac{n^2 \log n}{m}\})$ per edge; an improvement over the previous result. We then show that this analysis is almost tight. That is, for any $m \leq n(n-1)/2$, there is an input of m edges which will take the algorithm $\Omega(m^{3/2})$ time to process.

For $m = n-1$ our upper bound is $O(n^{3/2}\log n)$, which is far from the lower bound of $\Omega(n \log n)$. However, the worst case input, on which the algorithm performs $\Omega(n^{3/2})$

[1]Here and in the rest of the chapter we assume $m = \Omega(n)$.

work, contains bipartite cliques. In Section 4.2 we analyze the complexity of the same algorithm on structured graphs, and show that it has an implementation which is optimal on trees (assuming that the running time is at least proportional to the number of relabelling operations performed) and in general runs in time $O(mk \log^2 n)$, where k is the treewidth of the input DAG. While much of the current results that exploit the treewidth of graphs show that problems can be solved faster when the treewidth of the input graph is bounded, it is interesting to note that the notion also has other uses. For our result, we do not need to assume some bound on the treewidth, and we analyze an algorithm that does not use the treewidth or a tree decomposition of the given graph. Also, little research has so far been done on the effect of treewidth on online algorithms.

4.1 An $O(\min\{\sqrt{m}\log n, \sqrt{m} + \frac{n^2 \log n}{m}\})$ Amortized Upper Bound

Figure 4.3 shows a slight variation on the algorithm by Alpern et al. [2]. This variation works as follows. The *ord* labels of the nodes are maintained by an *Ordered List* data structure *ORD*, which is a data structure that allows to maintain a total order over a list of items and to perform the following operations in constant amortized time [4, 7]: *InsertAfter*(x, y) (*InsertBefore*(x, y)) inserts the item x immediately after (before) the item y in the total order, *Delete*(x) removes the item x, the query *Order*(x, y) determines whether x precedes y or y precedes x in the total order and *Next*(x) (*Prev*(x)) returns the item that appears immediately after (before) x in the total order. These operations are implemented by associating integer labels with the items in the list such that the label associated with x is smaller than the label associated with y iff x precedes y in the total order.

Initially, no edges exist in the graph so the nodes are inserted into *ORD* in an arbitrary order. Then, whenever an edge (*Source*, *Target*) is inserted into the graph, the function *AddEdge* is called. When this function returns, the total order *ORD* is a valid topological order for the modified graph.

It remains to describe how *AddEdge* works. Given a total order *ord* on the nodes and the edge (*Source*, *Target*) which is to be inserted, we define the *affected region AR* to be the set of nodes $\{v | ord(Target) \leq ord(v) \leq ord(Source)\}$. The set *FromTarget* contains all the nodes of *AR* which are reachable from *Target* by a path in the DAG and the set *ToSource* is the set of nodes in AR from which *Source* can be reached by a path. In the example in Figure 4.1, the nodes appear by the order of their *ord* labels and the dashed edge (I, C) is inserted. There, $AR = \{C, D, E, F, G, H, I\}$, *FromTarget* $= \{C, D, F\}$ and *ToSource* $= \{E, H, I\}$.

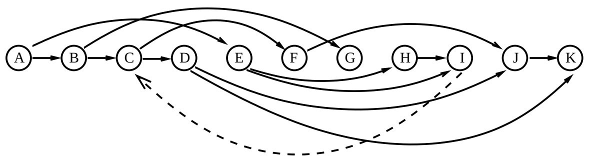


Figure 4.1: The dashed edge violates the order *ord*.

The algorithm relabels the nodes of a subset of *FromTarget* $\cup$ *ToSource*. It begins to explore the nodes in *FromTarget* and *ToSource*, the first by starting at *Target* and advancing by order of the *ord* labels of the nodes, and the second by starting at *Source* and advancing in reverse order of the *ord* labels. It builds two sets of nodes, *ToS* $\subseteq$ *ToSource* and *FromT* $\subseteq$ *FromTarget*, such that the desired order ord' can be obtained by relabelling the nodes in *ToS* $\cup$ *FromT*. The intuition behind the construction of these sets is that for each pair of nodes u, v such that $ord(u) > ord(v)$ and the new edge introduces a path from u to v, at least one of these nodes is in *ToS* $\cup$ *FromT*. The proof of correctness of this algorithm appears in [2].

Two priority queues are used for this search: *SourceQueue* for the nodes that can reach nodes in *ToS* and *TargetQueue* for the nodes that are reachable from nodes in *FromT*. In each iteration, there is one node s which is a candidate for insertion into *ToS* (the node with maximal *ord* label which reaches a node in *ToS* but is not in *ToS*) and one node t which is a candidate for insertion into *FromT* (the node with minimal *ord* label which can be reached from a node in *FromT* but is not in *FromT*). The algorithm adds at least one of them (possibly both) to the relevant set. The way in which it decides which candidate(s) to add aims at balancing the number of edges outgoing from nodes in *FromT* and the number of edges incoming to nodes in *ToS*. For each node $v \in V$, let *InDegree*$[v]$ be the number of edges incoming into v and *OutDegree*$[v]$ be the number of edges outgoing from v. Once a node becomes a candidate for insertion into *ToS* or *FromT*, it remains the candidate for insertion into its set until it is inserted or the function *AddEdge* terminates. The algorithm maintains two counters, one for the current s and the other for the current t. When a node v becomes a candidate for insertion into *ToS* (*FromT*), the relevant counter is set to

InDegree$[v]$ (*OutDegree*$[v]$). Whenever the two candidates are considered, the value of the smaller counter is subtracted from both counters. Then, each candidate is inserted into its respective set if the value of its counter is 0. This means that for every edge outgoing from a node in *FromTarget*, there is an edge incoming into a node in *ToSource*, and vice versa. When a node is inserted into *ToS* (*FromT*), its incoming (outgoing) edges are traversed and each of its predecessors (successors) is inserted into *SourceQueue* (*TargetQueue*) if it is not already there. It is important to note that the time spent by the algorithm is proportional to the number of edges adjacent to nodes that were inserted into *ToS* and *FromT*, and does not depend on the degrees of the last candidates for insertion into these sets.

The construction of the sets *ToS* and *FromT* ends when $ord(s) < ord(t)$ or $ToS = ToSource$ or $FromT = FromTarget$. Then, the nodes in *ToS* are deleted from *ORD* and are reinserted, in the same relative order among themselves, before all nodes in $FromTarget \setminus FromT$ and the nodes in *FromT* are deleted and reinserted in the same relative order among themselves after all nodes in *ToSource*. Finally, the edge (*Source*, *Target*) is inserted into the DAG. In the example in Figure 4.1, $FromT = \{C, D\}$, $ToS = \{E, H, I\}$ and after relabelling the nodes appear in *ORD* in the order shown in Figure 4.2.

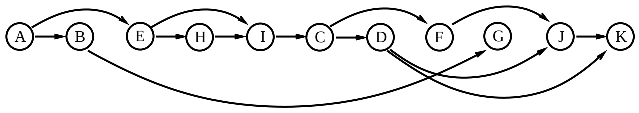


Figure 4.2: The graph of Figure 4.1 after relabelling the nodes and inserting the edge (I, C).

Remark. We have mentioned above that the algorithm we describe is a variation on the one by Alpern et al. The first change we made is that the *SourceDegree* and *TargetDegree* counters are initialized to the indegree or outdegree of a node. Alpern et al. initialize these values to the total number of edges incident on the node. In the bounded incremental complexity model, their definition is appropriate because it is the sum of the total degrees that should be minimized. For our analysis, however, this change is necessary. In addition, we have simplified the relabelling phase, compared to what was proposed by Alpern et al. Their relabelling phase minimizes the number of different labels used. While this gives a worse asymptotic complexity, they claim that it speeds up the operations on the Ordered List data structure in practice.

```
Function AddEdge (Source , Target )
    $SourceQueue \leftarrow []$; $TargetQueue \leftarrow []$; $SourceStack \leftarrow []$ ; $TargetStack \leftarrow []$
    $s \leftarrow Source$ ; $t \leftarrow Target$ ; $SourceDegree \leftarrow InDegree[s]$; $TargetDegree \leftarrow OutDegree[t]$

    (* Discovery phase: decide which nodes should be relabelled. *)
    while $(ord(s) > ord(t))$ do
        $d \leftarrow \min\{SourceDegree, TargetDegree\}$
        $SourceDegree \leftarrow SourceDegree - d$ ; $TargetDegree \leftarrow TargetDegree - d$
        if $SourceDegree = 0$ then
            foreach $(w, s) \in E$ do $SourceQueue.Insert(w)$
            $SourceStack$.Push($s$)
            if $SourceQueue$.NotEmpty then $s \leftarrow SourceQueue.ExtractMax$
            else $s \leftarrow Target$ (* Then we are done *)
            $SourceDegree \leftarrow InDegree[s]$
        endif
        if $TargetDegree = 0$ then
            foreach $(t, w) \in E$ do $TargetQueue.Insert(w)$
            $TargetStack$.Push($t$)
            if $TargetQueue$.NotEmpty then $t \leftarrow TargetQueue.ExtractMin$
            else $t \leftarrow Source$ (* Then we are done *)
            $TargetDegree \leftarrow OutDegree[t]$
        endif
    end while

    (* Relabel the nodes *)
    if $s = Target$ then $s \leftarrow ORD.Prev(Target)$
    while $SourceStack$.NotEmpty do $s' \leftarrow SourceStack$.Pop ; $ORD.Delete(s')$; $ORD.InsertAfter(s', s)$
    if $t = Source$ then $t \leftarrow ORD.Next(Source)$
    while $TargetStack$.NotEmpty do $t' \leftarrow TargetStack$.Pop ; $ORD.Delete(t')$; $ORD.InsertBefore(t', t)$

    (* Insert the edge *)
    $E \leftarrow E \cup \{(Source, Target)\}$
    $InDegree[Target] \leftarrow InDegree[Target] + 1$
    $OutDegree[Source] \leftarrow OutDegree[Source] + 1$
end
```

Figure 4.3: A variation on the algorithm by Alpern et al.

4.1.1 Complexity Analysis

The complexity analysis consists of two parts. In the first part, we show that the time spent on m edge insertions is $O(m^{3/2}\log n)$. In the second part, we show that this time is $O(m^{3/2}+n^2\log n)$. The combination of both results gives:

Theorem 4.1.1 *The topological order of the nodes of a DAG can be maintained while inserting m edges in total time $O(\min\{m^{3/2}\log n, m^{3/2}+n^2\log n\})$.*

Let $E=\{e_1,\ldots,e_m\}$ be the edges of the DAG, sorted by the order in which they are inserted. After inserting $e_1,\ldots,e_k$, for any $1\le i,j\le m$ we say that the pair of edges $e_i=(x_i,y_i), e_j=(x_j,y_j)$ is *ordered* if there is a path from y_i to x_j or from y_j to x_i that uses only edges from $\{e_1,\ldots,e_k\}$. That is, if the edges that were already inserted to the DAG form a path that determines the relative positions of e_i and e_j in the topological order. Otherwise, we say that the edges are *unordered*. Similarly, we say that a pair of nodes u,v is ordered if there is a path from u to v or from v to u, and unordered otherwise.

We denote by $\mathcal{U}_e\subseteq E\times E$ the set of unordered pairs of edges and by $\mathcal{U}_v\subseteq V\times V$ the set of unordered pairs of nodes.

Upon the insertion of an edge, let n_t (n_s) be the number of nodes in *FromT* (*ToS*). Let m_t (m_s) be the number of edges outgoing from (incoming to) nodes in *FromT* (*ToS*). Finally, let ℓ_t (ℓ_s) be the number of edges outgoing from (incoming to) the node that was candidate for insertion into *FromT* (*ToS*) when the algorithm terminated. If this node does not exist, i.e., *TargetQueue* (*SourceQueue*) is empty at the end, then $\ell_t=0$ ($\ell_s=0$).

4.1.2 An $O(m^{3/2}\log n)$ Upper Bound

We define a potential function $\Phi=|\mathcal{U}_e|$. I.e., the potential is equal to the number of unordered pairs of edges. Initially, all pairs of edges are unordered, so $\Phi_0=m(m-1)<m^2$.

Lemma 34 *The algorithm spends on an insertion $O(1+\max\{m_t,m_s\}\log n)$ time.*

Proof The time that the algorithm spends on the insertion is a constant amount c, plus $O(m_t\log m_t+m_s\log m_s)$ to construct the sets *ToS* and *FromT* and $O(n_t+n_s)$ for the relabelling. With $n_i\le m_i$ for $i\in\{s,t\}$, we have that the time spent is $O(1+m_t\log m_t+$

$m_s \log m_s) = O(1 + (m_t + m_s) \log m) = O(1 + \max\{m_t, m_s\} \log n)$. ∎

Lemma 35 *For every edge insertion, (1) $m_t \le m_s + \ell_s$ and (2) $m_s \le m_t + \ell_t$.*

Proof We show by induction on the number of iterations of the while loop which constructs the sets *FromT* and *ToS*, that after each iteration, $m_t + \ell_t - \mathit{TargetDegree} = m_s + \ell_s - \mathit{SourceDegree}$. Since at all times $0 \le \mathit{TargetDegree} \le \ell_t$ and $0 \le \mathit{SourceDegree} \le \ell_s$, (1) and (2) follow for the final as well as the intermediate values of m_t, ℓ_t, m_s, ℓ_s.

Initially, $m_t = m_s = 0$, $\mathit{TargetDegree} = \ell_t$ and $\mathit{SourceDegree} = \ell_s$, so the equality holds. Assume that it holds up to iteration $i-1$ of the discovery loop. We will show that it holds also after iteration i. For any value $\mathit{val} \in \{m_t, \ell_t, m_s, \ell_s, \mathit{TargetDegree}, \mathit{SourceDegree}\}$, let val denote this value after iteration $i-1$ and val' the same value after iteration i.

Since the algorithm did not terminate after iteration $i-1$, we know that there is a candidate for insertion into each of *FromT* and *ToS* at the beginning of iteration i. Assume, w.l.o.g., that $\mathit{SourceDegree} \le \mathit{TargetDegree}$. Then s will be inserted into *ToS*, so $m'_s = m_s + \ell_s$ and $\mathit{SourceDegree}' = \ell'_s$. We get that $m'_s + \ell'_s - \mathit{SourceDegree}' = m_s + \ell_s$.

For t, there are two cases. The first case is that $\mathit{TargetDegree} = \mathit{SourceDegree}$ so t will be inserted into *FromT*. Similar to the above, we get that $m'_t + \ell'_t - \mathit{TargetDegree}' = m_t + \ell_t$. Since $\mathit{SourceDegree} = \mathit{TargetDegree}$, we know from the induction hypothesis that $m_t + \ell_t = m_s + \ell_s$, which means that $m'_s + \ell'_s - \mathit{SourceDegree}' = m'_t + \ell'_t - \mathit{TargetDegree}'$.

The second case is that $\mathit{TargetDegree} > \mathit{SourceDegree}$ so t will not be inserted into *FromT*. This implies that $m'_t = m_t$, $\ell'_t = \ell_t$ and $\mathit{TargetDegree}' = \mathit{TargetDegree} - \mathit{SourceDegree}$. We get that $m'_t + \ell'_t - \mathit{TargetDegree}' = m_t + \ell_t + \mathit{TargetDegree} - \mathit{SourceDegree}$. By the induction hypothesis, this is equal to $m_s + \ell_s$, which is equal to $m'_s + \ell'_s - \mathit{SourceDegree}'$. ∎

For $0 \le i \le m$, let Φ_i be the value of the potential function after the insertion of $e_1, \ldots, e_i$ to the graph. For $1 \le i \le m$, let $\Delta\Phi_i = \Phi_i - \Phi_{i-1}$ be the change in potential due to the insertion of e_i.

Lemma 36 *When the algorithm handles an insertion, $\Delta\Phi_i \le -\max\{m_t^2, m_s^2\}$. That is, the potential decreases by at least $\max\{m_t^2, m_s^2\}$.*

Proof Let $e_t = (x_t, y_t)$ be one of the $m_t + \ell_t$ edges outgoing from a node in *FromT* or the last candidate for insertion into *FromT* and let $e_s = (x_s, y_s)$ be one of the $m_s + \ell_s$ edges incoming into a node in *ToS* or the last candidate for insertion into *ToS*. We will show that e_t and e_s were unordered before the insertion and ordered after the insertion. After the insertion, y_t is reachable from *Target* by a path P_t that uses e_t and *Source* is reachable from x_s by a path P_s that uses e_s. Concatenating the paths with the inserted edge between them gives the path $P_s \circ (\mathit{Source}, \mathit{Target}) \circ P_t$ from x_s to y_t that uses both edges. Hence, the edges are ordered such that $\mathit{ord}'(y_s) < \mathit{ord}'(x_t)$.

Assume that they are ordered before the insertion as well. If they are ordered with $\mathit{ord}(y_t) < \mathit{ord}(x_s)$ then the insertion introduces a cycle, so the graph is not a DAG. So they are ordered such that $\mathit{ord}(y_s) < \mathit{ord}(x_t)$. If y_s and x_t were candidates for insertion into their sets in the same iteration of the discovery phase when the algorithm processed the insertion of the edge $(\mathit{Source}, \mathit{Target})$, then their relative order in *ORD* would imply that the algorithm terminates without adding either one of them, a contradiction. Assume, w.l.o.g., that y_s was inserted before x_t became the candidate. Then when x_t became the candidate for insertion into *FromT*, the candidate for insertion into *ToS* was a node v such that $\mathit{ord}(v) < \mathit{ord}(y_s) < \mathit{ord}(x_t)$. So the algorithm should have terminated without adding x_t.

We get that the potential decreased by at least $(m_t + \ell_t)(m_s + \ell_s)$, which by Lemma 35 is at least $\max\{m_t^2, m_s^2\}$. ▮

Theorem 4.1.2 *The algorithm in Figure 4.3 needs $O(m^{3/2} \log n)$ time to insert m edges into an initially empty n-node graph.*

Proof By Lemma 34, the algorithm spends $O(1 + \max\{m_t, m_s\} \log n)$ time on an insertion, while by Lemma 36, the potential decreases by at least $\max\{m_t^2, m_s^2\}$.

For $i = 1, \ldots, m$, let x_i be the value of $\max\{m_t, m_s\}$ upon the insertion of the ith edge. We get that the total time spent on the m insertions is $O(\sum_{i=1}^{m}(1 + x_i \log n)) = O(m + \sum_{i=1}^{m} x_i \log n)$ and the potential decreased by a total of $\Phi_m - \Phi_0 = \sum_{i=1}^{m} \Delta\Phi_i \leq -\sum_{i=1}^{m} x_i^2$. Since $\Phi_0 < m^2$ and $\Phi_m \geq 0$, we have $\Phi_m - \Phi_0 > -m^2$ which implies $\sum_{i=1}^{m} x_i^2 < m^2$.

Cauchy's inequality states that

$$\left(\sum_{i=1}^{m} a_i b_i\right)^2 \leq \left(\sum_{i=1}^{m} a_i^2\right)\left(\sum_{i=1}^{m} b_i^2\right).$$

By substituting $a_i = x_i$ and $b_i = 1$ for all $1 \leq i \leq m$, we get that $\sum_{i=1}^{m} x_i < m^{3/2}$, so the total time spent by the algorithm on the m edge insertions is $O(m + \sum_{i=1}^{m} x_i \log n) = O(m^{3/2} \log n)$. ∎

4.1.3 An $O(m^{3/2} + n^2 \log n)$ Upper Bound

In this section we change two aspects of the analysis, compared to the previous section. The first is the potential function - it now counts not only unordered pairs of edges but also unordered pairs of nodes: $\Psi = |\mathcal{U}_e| + |\mathcal{U}_v|$.

The second change is in the analysis of the actual time spent by the algorithm when it processes the insertion of an edge. We assume that the priority queues are implemented by a data structure that supports insertions in constant amortized time and extractions in $O(\log n)$ amortized time (e.g., Fibonacci Heaps [8]). Note that the number of items inserted into *SourceQueue* can be as large as m_s, but the number of items extracted from it is bounded by n_s. This implies that the time spent on identifying the set *ToS* during the discovery phase is $O(m_s + n_s \log n)$, which is a tighter bound than the one we had in the previous section, of $O(m_s \log n)$. Similarly, the time spent on identifying the set *FromT* is $O(m_t + n_t \log n)$. Thus, we have shown:

Lemma 37 *The algorithm spends* $O(1 + \max\{m_t, m_s\} + \max\{n_t, n_s\} \log n)$ *time on an insertion.*

We now analyze the change in potential due to an insertion.

Lemma 38 *When the algorithm handles an insertion,* $\Delta\Psi_i \leq -\max\{m_t^2, m_s^2\} - \max\{n_t, n_s\}$. *I.e., the potential decreases by at least* $\max\{m_t^2, m_s^2\} + \max\{n_t, n_s\}$.

Proof We have shown in the proof of Lemma 36 that $|\mathcal{U}_e|$ decreases by at least $\max\{m_t^2, m_s^2\}$. In addition, for each $v \in$ *FromT* ($v \in$ *ToS*), the pair $\langle Source, v\rangle$ ($\langle v, Target\rangle$) was ordered after the insertion. If it was also ordered before the insertion, then the new edge introduces a cycle in the graph, contradicting our assumption that it is a DAG. So $|\mathcal{U}_v|$ decreases by at least $|FromT| + |ToS| = n_t + n_s \geq \max\{n_t, n_s\}$. ∎

Theorem 4.1.3 *The algorithm in Figure 4.3 needs $O(m^{3/2}+n^2 \log n)$ time to insert m edges into an initially empty n-node DAG.*

Proof By Lemma 37, an insertion takes $O(1+\max\{m_t, m_s\}+\max\{n_t, n_s\}\log n)$ time while by Lemma 38 the potential decreases by at least $\max\{m_t^2, m_s^2\}+\max\{n_t, n_s\}$, with a decrease of at least $\max\{m_t^2, m_s^2\}$ in the term $|\mathcal{U}_e|$ and a decrease of at least $\max\{n_t, n_s\}$ in the term $|\mathcal{U}_v|$.

For $i = 1, \ldots, m$, let x_i be the value of $\max\{m_t, m_s\}$ and y_i be the value of $\max\{n_t, n_s\}$ upon the insertion of the ith edge. We get that the total time spent on the m insertions is $O(\sum_{i=1}^m (1 + x_i + y_i \log n)) = O(m + \sum_{i=1}^m x_i + \log n \sum_{i=1}^m y_i)$ where $\sum_{i=1}^m x_i^2 \leq m^2$ and $\sum_{i=1}^m y_i \leq n^2$. As in the proof of Theorem 4.1.2, $\sum_{i=1}^m x_i \leq m^{3/2}$ so the total time spent is $O(m + \sum_{i=1}^m x_i + \sum_{i=1}^m y_i \log n) = O(m^{3/2} + n^2 \log n)$. ∎

4.1.4 Almost Tightness of the Analysis

We now show that our analysis is almost tight. That is, we show that for each $m \leq n(n-1)/2$ there is an input with n nodes and m edges that will take the algorithm in Figure 4.3 $\Omega(m^{3/2})$ time to process, which is equal to the upper bound for dense graphs and is merely a log factor away for sparse graphs.

Let $k = \sqrt{m}/4$ and let $\{v_1, \ldots, v_n\}$ be the nodes of the DAG sorted by their initial *ord* order. The first part of the input consists of the edge (v_i, v_j) for all $i \in \{1, \ldots, k\}, j \in \{2k+1, \ldots, 3k\}$ and $i \in \{k+1, \ldots, 2k\}, j \in \{3k+1, \ldots, 4k\}$.

In the second part, an edge is introduced from v_i to v_j for each $i \in \{2k+1, \ldots, 3k\}$ and $j \in \{k+1, \ldots, 2k\}$, in the following order: For all $i \in \{2k+1, \ldots, 3k-1\}$, all edges outgoing from v_i appear before all edges outgoing from v_{i+1} and for all $j \in \{k+2, \ldots, 2k\}$, the edge (v_i, v_j) appears before the edge (v_i, v_{j-1}).

So far, the input consisted of $3k^2 = 3m/16$ edges. In the third part, we complete this to m by inserting any $13m/16$ edges that are not already in the DAG.

We show that the second part will cause the algorithm to perform $\Omega(m^{3/2})$ work. Every edge (v_i, v_j) that is inserted in the second part is from a node v_i to a node v_j which is immediately before it in *ORD*. Since *InDegree*$[v_i] \geq k$ and *OutDegree*$[v_j] \geq k$, the algorithm will insert v_i into *ToS* or v_j into *FromT* (possibly both) and will traverse at least k incident edges. Then, it will reverse the order of *Source* and *Target* in *ORD*. Thus, it will spend $\Omega(k)$ time for each edge insertion and since the number of edges inserted in the second part is k^2, we get a total complexity of $\Omega(k^3) = \Omega(m^{3/2})$.

4.2 The Complexity on Structured Graphs

The only non-trivial lower bound for online topological ordering is due to Ramalingam and Rep [23], who show that an adversary can force any algorithm to perform $\Omega(n \log n)$ node relabelling operations while inserting $n-1$ edges (creating a chain). The upper bound we have shown for $m = n-1$ is $O(n^{3/2} \log n)$ time. We have shown that for any m we can construct an input on which the algorithm performs $\Omega(m^{3/2})$ work, but this input contains bipartite cliques.

In this section we show that the algorithm performs much better when the graph is structured. In particular, we show that for any DAG G, the algorithm runs in time $O(mk \log^2 n)$ where k is the treewidth of G. In addition, we show that the algorithm can be implemented such that on a tree it spends a total of $O(n \log n)$ time, i.e., it is optimal. The notion of treewidth was introduced by Robertson and Seymour [27]. We start with giving the definition of treewidth.

Definition 4.2.1 *A* tree decomposition *of a graph $G = (V, E)$ is a pair $(\{X_i \mid i \in I\}, T = (I, F))$ with T a tree, and $\{X_i \mid i \in I\}$ a family of subsets of V, such that $\bigcup_{i \in I} X_i = V$, for all $\{v, w\} \in E$, there is an $i \in I$ with $v, w \in X_i$, and for all $v \in V$, the set of nodes $T_v = \{i \in I \mid v \in X_i\}$ induces a connected subgraph (subtree) of T.*

The width *of a tree decomposition $(\{X_i \mid i \in I\}, T)$ is $\max_{i \in I} |X_i| - 1$. The treewidth of a graph G is the minimum width of a tree decomposition of G. The treewidth of a directed graph G is the treewidth of the underlying undirected graph of G.*

Trees have treewidth one. For an overview of the treewidth of several classes of graphs, see e.g., [5].

Lemma 39 *The algorithm can be implemented such that if G is a tree, it performs a total of $O(n \log n)$ work.*

Proof sketch Let T_v be the tree that node v belongs to. When an edge (u, v) is inserted, two trees T_u and T_v are merged into one tree. Assume, w.l.o.g., that $|T_u| \leq |T_v|$. Then the algorithm can be implemented such that it relabels $O(|T_u|)$ nodes and both the discovery and relabelling phases take $O(|T_u|)$ time. To do this, we need to give up the priority queues used in the discovery phase. Going back from *Source* is easy, because each node has indegree at most 1 so at all times there is a single candidate for insertion to *ToS*. The descendants of *Target* will be traversed in DFS order and will appear in the new topological order according to their relative postorder DFS numbers. Since we do not examine the nodes of *FromTarget* in increasing topological order, the discovery phase must continue until all of the nodes of either *ToSource* or *FromTarget* have been traversed.

Define the potential function $f = \sum_{v \in V} \log |T_v|$. When an edge (u, v) is inserted and the algorithm performs $O(|T_u|)$ work, the potential increases by at least $|T_u|$. The lemma follows because the potential is initially 0 and it is never larger than $n \log n$. ▮

We now turn to the case where $k > 1$. We need a simple lemma, which is a small variant of a well known result. (Compare, e.g., [27].) The proofs of the three lemmas below follow standard techniques.

Lemma 40 *Let T be a tree with ℓ leaves. Then there is a node v in T such that each connected component of $T - v$ contains at most $\ell/2$ leaves from T.*

Proof Suppose the lemma does not hold for tree T. For each node v from T, there is then exactly one subtree of $T - v$ with more than $\ell/2$ leaves. Build a directed graph H, by taking for each node v a directed edge to its neighbor in this subtree. As each node in H has outdegree one, H must contain a cycle, hence two neighboring nodes v_0, v_1 with edges (v_0, v_1) and (v_1, v_0) in H. The subtree of $T - v_0$ that contains v_1 is disjoint from the subtree of $T - v_1$ that contains v_0, and both these subtrees contain more than $\ell/2$ leaves, so T has more than ℓ leaves, contradiction. ▮

Lemma 41 *Let $G = (V, E)$ be a graph with treewidth k. There is a set $S \subseteq V$ such that $|S| \leq k + 1$, and for each connected component of $G - S$ there are at most $m/2$ edges with at least one endpoint in the component.*

Proof Take a tree decomposition $(\{X_i \mid i \in I\}, T = (I, F))$ of G of width at most k. For each edge $e = \{v, w\} \in E$, add one node i_e to the tree decomposition with $X_{i_e} = \{v, w\}$, and make i_e adjacent in T to an (old) node $i' \in I$ with $v, w \in X_{i'}$. Let $(\{X_i \mid i \in I'\}, T' = (I', F'))$ be the resulting tree decomposition. We have $I' = I \cup \{i_e \mid e \in E\}$. While T' has one or more leaf nodes that belong to I, i.e., do not correspond to an edge, remove such leaf nodes. Let $(\{X_i \mid i \in I'\}, T'' = (I'', F''))$ be the resulting tree decomposition. (One can verify that this is a tree decomposition of G of width at most k.) There is a one-to-one correspondence between the edges of E and the leaves of T''.

Let i^0 be the node in T'' such that each subtree of $T'' - i_0$ contains at most $m/2$ nodes that were a leaf in T'', i.e., correspond to an edge in E. Set $S = X_{i_0}$. Clearly, $|S| \leq k + 1$. Consider a connected component of $G - S$. By the properties of tree decomposition, there can be only one subtree of $T - i_0$ whose sets X_i, i in the subtree, can contain nodes from the component. In particular, for each edge $e = \{x, y\}$ with one endpoint in the component, we have that i_e belongs to that subtree. So, all edges with one endpoint in the component correspond to a node in the subtree that is a leaf in T''. So, the component has at most $m/2$ edges with at least one endpoint in it. ∎

Lemma 42 *Let $G = (V, E)$ be a graph with treewidth k. There is partition of V in three disjoint subsets A, B, and S, such that $|S| \leq k + 1$, no node in A is adjacent to a node in B, the number of edges with at least one endpoint in A is at most $2m/3$, and the number of edges with at least one endpoint in B is at most $2m/3$.*

Proof Let S be as in Lemma 41. If $G - S$ has a connected component with at least $m/3$ edges, then put the nodes of this connected component in A, and the nodes of the other connected component in B, and we are done. Otherwise, put the nodes of the components one by one in A, until the components in A together have at least $m/3$ edges. Put the nodes of the remaining components in B. Now, there are at least $m/3$ and at most $2m/3$ edges with at least one endpoint in A, and hence also at most $2m/3$ edges with at least one endpoint in B. ∎

Except in some degenerate cases, the set S from Lemma 42 is a separator. Let S be the set as in Lemma 42. Now define the potential function

$$\Psi = \sum_{v \in S} \sum_{e \in E} I(v, e)$$

where $I(v, e)$ is 1 if v and e are unordered and 0 otherwise. That is, the potential function counts the number of unordered pairs of an edge from the graph and a node from S. Clearly, $\Psi_0 = 0$ and at all times $\Psi \leq |S||E| \leq (k+1)m$.

The edges can be partitioned into three sets: the sets E_A (E_B) of edges whose insertions cause only nodes from A (B) to be relabelled, and the set $E_M = E \setminus (E_A \cup E_B)$. Let $T(m)$ be the time spent during all m edge insertions. Then the insertion of all edges from E_A (E_B) occur within the subgraph induced by the nodes in A (B). The time spent on the insertion of the edges in E_A can thus be bounded by $T(|E_A|)$ ($T(|E_B|)$).

While handling the insertion of an edge $e \in E_M$, the algorithm relabels at least one node v from S. Assume, w.l.o.g., that $v \in$ *ToSource*. Then by considerations similar to the ones used in the proof of Lemma 36, there are $(m_t + \ell_t)$ edges outgoing from nodes in *FromTarget* that were not ordered with v before the insertion of e and are ordered afterwards. Hence, $\Delta\Psi \geq m_t + \ell_t$, which by Lemma 35 is at least $\max\{m_s, m_t\}$. Since the time spent is $O(\max\{m_s, m_t\} \log n)$, we obtain that the insertion of all edges from E_M requires a total of $O(mk \log n)$ time. Hence, we have for the total time that $T(m) \leq mk \log n + T(|E_A|) + T(|E_B|)$, and as $|E_A| + |E_B| \leq m$, and $|E_A| \leq 2m/3$, $|E_B| \leq 2m/3$, it follows that $T(m) = O(mk \log^2 n)$. So we have shown:

Theorem 4.2.2 *The topological order of the nodes of a DAG with treewidth k can be maintained while inserting m edges into an initially empty n-node DAG in total time $O(mk \log^2 n)$. For the special case of trees, this problem can be solved in $O(n \log n)$ time, which is optimal.*

4.3 Conclusion

We have shown that the online topological ordering problem can be solved in $O(\min\{\sqrt{m} \log n, \sqrt{m} + \frac{n^2 \log n}{m}\})$ amortized time per edge, an improvement over the previous result of $O(n)$ time per edge. We have also shown that for any $m \leq n(n-1)/2$ there is an input with m edges on which the algorithm performs $\Omega(m^{3/2})$ work, indicating that our analysis of the algorithm's running time is almost tight if only the number of nodes and number of edges is known.

We then analyze the algorithm's complexity on structured graphs. We show that the algorithm has an optimal implementation on trees and that in general there is a correlation between the treewidth of the graph and the algorithm's complexity. There is here still a gap between upper and lower bound. Observe that for trees ($k = 1$), the general bound gives $O(n \log^2 n)$ and not the $O(n \log n)$ result that we have obtained separately. It is an open problem whether a different analysis or implementation can yield $O(mk \log n)$ running time.

Chapter 5

Dynamic Heaviest Paths in DAGs with Arbitrary Edge Weights

The *Heaviest Paths* (HP) problem is as follows. Given a Directed Acyclic Graph (DAG) $G = (V, E)$ with a weight $w(e)$ for each edge e, compute for each node $v \in V$ the weight of the heaviest path from the source of G to v, where the weight of a path is the sum of the weights of its edges. The *Dynamic Heaviest Paths* (DHP) problem is to efficiently update this information when a small change is performed on G. Here "efficient" means that the running time is proportional to the size of the portion of the graph that is affected by the change [24].

Formally, for each $v \in V$ denote by $\hbar(v)$ the weight of the heaviest path in G from the source of G to v. Let G' be the graph obtained by performing an operation on G (such as adding or deleting an edge) and let $\hbar'(v)$ be the weight of the heaviest path to v in G'. We define $\delta = \{v | v \in V \wedge \hbar(v) \neq \hbar'(v)\}$ to be the set of nodes that were affected by the operation. That is, the nodes for whom the weight of the heaviest path changed. We define $|\delta|$ to be the number of nodes in δ and $||\delta||$ to be $|\delta|$ plus the number of edges that are adjacent to at least one node in δ. Assuming that any algorithm would have to do something for each node in δ and to examine each edge adjacent to a node $v \in \delta$, we use $|\delta|$ and $||\delta||$ as the parameters for the complexity of an update.

Michel and Van Hentenryck [17] recently studied the DHP problem. Their research was motivated by scheduling applications, where one wishes to represent the current solution by a DAG and to perform tasks such as to evaluate the makespan and update it upon a small change to the DAG. They present algorithms that solve the DHP problem on DAGs with strictly positive edge weights. Their algorithms run in time $O(\|\delta\| + |\delta|\log|\delta|)$ for an edge insertion and $O(\|\delta\|)$ for an edge deletion. They show that it is possible to replace edges with zero or negative weights by edges with positive weights and then apply their algorithm. However, the number of edges added may be large. They conclude by asking whether there is an efficient algorithm that can handle arbitrary edge weights without graph transformations.

We answer their question by showing such an algorithm. In fact, their algorithm for updating the graph upon the deletion of an edge works also in the presence of non-positive edge weights. So what we need to show is an algorithm for edge insertion that can handle arbitrary weights. Our solution has the same asymptotic complexity as theirs and is not more complicated to implement.

In addition, we discuss the case in which the edge weights are integers and show an algorithm that runs in time $O(\|\delta\| + |\delta|\log\log\min\{|\delta|, \Delta_{\max}+1\})$ where $\Delta_{\max} = \max_{v\in V}\{\hbar'(v) - \hbar(v)\}$ is the maximum change to the heaviest path value of a node due to the edge insertion.

5.1 Fibonacci Heaps

Our algorithms use *Fibonacci Heaps* [8]. A min-sorted (max-sorted) Fibonacci Heap is a data structure that supports the following operations:

- *Insert*(i,p) inserts the item i with priority p.
- *UpdatePriority*(i,p): If i is not in the heap, performs *Insert*(i,p). If i is in the heap and has priority p', its priority is updated to $\min\{p,p'\}$ ($\max\{p,p'\}$).
- *ExtractMin* (*ExtractMax*) returns the item with minimal (maximal) priority and removes it from the heap.

Any sequence of n_i *Insert* operations, n_u *UpdatePriority* operations and n_e *ExtractMin*(*ExtractMax*) operations on an initially empty Fibonacci Heap can be performed in time $O(n_i + n_u + n_e\log n)$, where n is the maximal number of items in the heap at any point in time. Certainly, $n \le n_i + n_u$.

5.2 Intuition

Figure 5.1 shows the impact of adding the dashed edge to the DAG. The nodes in δ are the ones inside the rectangle and the changes to their $\hbar$ values are shown. Clearly, these changes propagate along paths in the graph. That is, if we add an edge (x, y) then $\hbar(v)$ for a node $v \neq y$ can only change if $\hbar(u)$ changed for a predecessor u of v. Furthermore, the changes propagate in a monotonous manner. That is, the change to $\hbar(v)$ is not larger than the maximum of the changes at v's predecessor.

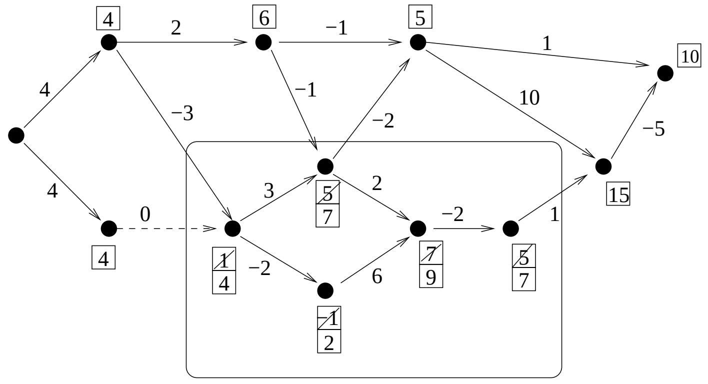


Figure 5.1: The impact of inserting the dashed edge to the DAG.

The algorithm in [17] would proceed from y in topological order and update the $\hbar$ value for each node only after the updated $\hbar$ values for all of its predecessors are known. This is possible when edge weights are strictly positive because the previous $\hbar$ values provide us with the relative topological order of the nodes. When zero or negative edge weights exist in the graph, it may be the case that $\hbar(v) \leq \hbar(u)$ while u precedes v in the topological order (see [17], Section 6 for an example). In other words, we do not know the topological order of the nodes so we need a different method to traverse them, which would still guarantee that we do not traverse the same subgraph more than once.

The rule we use is as follows. We begin by updating $\hbar(y) \leftarrow \max\{\hbar(y), \hbar(x) + w(x, y)\}$. This is the final value for $\hbar(y)$ because it can only change due to the edge (x, y), so we label y as *processed*. Then we go over the edges outgoing from y and for each such edge $e = (y, v)$, we insert the node v into a set which we call the *frontier* and which contains the nodes which are not processed but have a processed predecessor. In addition, we compute the change $\Delta^{(y,v)}$ that would occur to $\hbar(v)$ by advancing along the edge e: $\Delta^{(y,v)} = \hbar(y) + w(y, v) - \hbar(v)$.

We select v for which $\Delta^{(y,v)}$ is maximal, update $\hbar(v) \leftarrow \hbar(v) + \Delta^{(y,v)}$, remove v from the frontier and mark it as processed. As we will show later, we have found the final value for $\hbar(v)$. We then continue in the same manner: for each successor u of v, we insert u into the frontier if it is not already there and compute $\Delta^{(v,u)}$. Since u may have already been in the frontier, we set $\Delta(u) = \max\{\Delta^{(u',u)} | u' \text{ is processed}\}$. We then select the node v' from the frontier with maximal $\Delta(v')$ value and advance on an edge (w, v') such that $\Delta(v') = \Delta^{(w,v')}$. We repeat this until there is no node v in the frontier with $\Delta(v) > 0$.

The only point to add is that when we process a node v and compute the value Δ^e for an edge $e = (v, u)$, u might already belong to the frontier. In that case, we do not want to add it again but rather to update $\Delta(u)$ if necessary. For this reason, we use a max-sorted Fibonacci Heap for the nodes in the frontier, where the priority of a node u is $\Delta(u)$.

5.3 The Algorithm

Figure 5.2 shows the algorithm for updating the heaviest path values of the affected nodes upon insertion of an edge to the DAG. It receives a DAG $G = (V, E)$, the function $\hbar$ for G, an edge (x, y) which is to be inserted to G and the weight $w(x, y)$ of this edge. It updates the function $\hbar$ for the graph $G' = (V, E \cup \{(x, y)\})$.

Initially, it inserts the node y into the Fibonacci Heap Fh with priority equal to $\Delta_y = \hbar(x) + w(x, y) - \hbar(y)$. It then enters the while loop, which continues as long as Fh is not empty. It extracts from Fh a node u with maximal priority and updates $\hbar(u)$ for this node. Then it traverses the edges outgoing from u and updates the Δ values for the target of each of these edges. Recall that the operation *Fh.UpdatePriority*(v, Δ) inserts v to Fh if it is not already there, and otherwise sets its priority to the maximum among Δ and its previous priority.

5.3.1 Example

Before turning to the correctness proof, we show how this algorithm operates on the example shown in Figure 5.1. Figure 5.3 reproduces the part of the graph that the algorithm explores, with labels on the nodes.

Initially, y is inserted to Fh with priority $\Delta = \hbar(x) + w(x, y) - \hbar(y) = 4 + 0 - 1 = 3$.

```
**Function** *Insert*$(G, (x,y), \hbar, w(x,y))$
    $Fh \Leftarrow []$ (* Empty Fibonacci Heap. *)
    $\Delta = \hbar(x) + w(x,y) - \hbar(y)$
    **if** $\Delta > 0$ **then**
        $Fh.Insert(y, \Delta)$
    **endif**
    **while** $Fh$.NotEmpty **do**
        $(u, \Delta) \Leftarrow Fh.ExtractMax$
        $\hbar(u) \Leftarrow \hbar(u) + \Delta$ (* Update $\hbar(u)$ *)
        **foreach** $e = (u,v) \in E$ **do**
            $\Delta \Leftarrow \hbar(u) + w(u,v) - \hbar(v)$
            **if** $\Delta > 0$ **then**
                $Fh.UpdatePriority(v, \Delta)$
            **endif**
        **endfor**
    **end while**
**end**
```

Figure 5.2: Algorithm for updating $\hbar$ values upon an edge insertion.

Then the algorithm enters the while loop and performs five iterations.

Iteration 1: $(u, \Delta) \leftarrow (y, 3)$ and $\hbar(y) \leftarrow \hbar(y) + \Delta = 1 + 3 = 4$. Next, the successors of y are checked. a is inserted to Fh with priority $\Delta_a = \hbar(y) + w(y,a) - \hbar(a) = 4 + 3 - 5 = 2$ and b is inserted with priority $\Delta_b = \hbar(y) + w(y,b) - \hbar(b) = 4 - 2 + 1 = 3$.

Iteration 2: $(u, \Delta) \leftarrow (b, 3)$, $\hbar(b) \leftarrow \hbar(b) + \Delta = -1 + 3 = 2$. c is inserted to Fh with priority $\Delta_c = \hbar(b) + w(b,c) - \hbar(c) = 2 + 6 - 7 = 1$.

Iteration 3: $(u, \Delta) \leftarrow (a, 2)$, $\hbar(a) \leftarrow \hbar(a) + \Delta = 5 + 2 = 7$. The priority of c is updated from 1 to $\hbar(a) + w(a,c) - \hbar(c) = 7 + 2 - 7 = 2$ and $\Delta_f = \hbar(a) + w(a,f) - \hbar(f) = 7 - 2 - 5 = 0$ so f is not inserted to Fh.

Iteration 4: $(u, \Delta) \leftarrow (c, 2)$, $\hbar(c) \leftarrow \hbar(c) + \Delta = 7 + 2 = 9$. d is inserted to Fh with priority $\Delta_d = \hbar(c) + w(c,d) - \hbar(d) = 9 - 2 - 5 = 2$.

Iteration 5: $(u, \Delta) \leftarrow (d, 2)$, $\hbar(d) \leftarrow \hbar(d) + \Delta = 5 + 2 = 7$. $\Delta_e = \hbar(d) + w(d,e) - \hbar(e) = 7 + 1 - 15 = -7$ so e is not inserted to Fh.

Now, Fh is empty so the algorithm terminates.

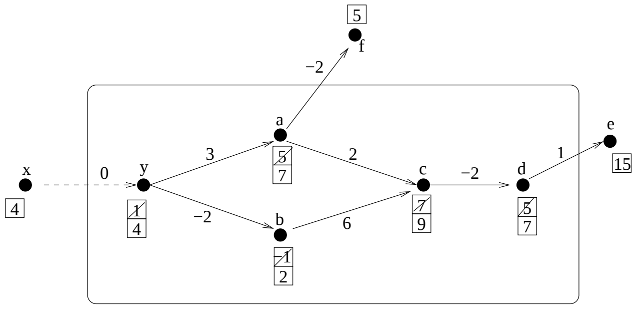


Figure 5.3: The portion of the graph of Figure 5.1 that the algorithm explores.

5.3.2 Proof of correctness

For a node $v \in V$, let $\hbar(v)$ be the weight of the heaviest path to v in the input DAG G and let $\hbar'(v)$ be the weight of the heaviest path to v in the DAG G' that we get by adding the edge (x, y) to G. Let $\bar{\Delta}(v) = \hbar'(v) - \hbar(v)$ be the change that occurs in the heaviest path value for v due to the insertion of this edge. To show that the algorithm computes the correct value for $\hbar'(v)$ for all $v \in V$, we will show that (1) If $\bar{\Delta}(v) > 0$ then v is inserted to *Fh* and when it is extracted from *Fh*, its priority is equal to $\bar{\Delta}(v)$. (2) If $\bar{\Delta}(v) \leq 0$ then v is not inserted to *Fh*.

It is easy to see that if (1) and (2) hold then the algorithm computes the correct value of $\hbar'(v)$ for all nodes.

Lemma 43 *For all $v \in V$, if $\bar{\Delta}(v) > 0$ then v is inserted into Fh and when it is extracted from Fh, its priority is equal to $\bar{\Delta}(v)$.*

Proof Assume the converse and let v be minimal w.r.t. the topological order such that $\bar{\Delta}(v) > 0$ and the claim does not hold for v. By construction, the priority of a node in *Fh* can never be higher than its $\bar{\Delta}$ value. So we need to show that v is inserted into the queue and that its priority when it is extracted is not less than $\bar{\Delta}(v)$.

By definition of $\hbar'$, there must be a predecessor u of v such that $\hbar'(v) = \hbar'(u) + w(u, v)$. There are two cases.

Case 1: $u = x$ and $v = y$ and (u, v) is the edge that was inserted to the graph. Then v is inserted into *Fh* at the beginning with priority $\bar{\Delta}(v)$ and is extracted immediately afterwards.

Case 2: $v \neq y$. Then there is a predecessor u of v such that $\hbar'(v) = \hbar'(u) + w(u, v)$. Since the edge (u, v) was in the graph before the insertion, $\hbar'(u) > \hbar(u)$, hence $\bar{\Delta}(u) > 0$. Since u

precedes v in the topological order, and by the minimality of v w.r.t. the topological order, we know that u was inserted to *Fh* and was extracted with priority equal to $\bar{\Delta}(u)$. If v was not extracted from *Fh* before u, then after u was extracted, v was either inserted with priority $\bar{\Delta}(v)$ (if it was not already in the queue) or its priority was updated to $\bar{\Delta}(v)$ (if it was already in *Fh*). Hence, v was inserted to *Fh* and when it was extracted, its priority was $\bar{\Delta}(v)$.

Assume that v was extracted before u. Let $P =< z_1, \ldots, z_n >$ be a heaviest path in G' from y to v through u. That is, $z_1 = y$, $z_{n-1} = u$ and $z_n = v$ and for each $1 < i \leq n$, $\hbar'(z_i) = \hbar'(z_{i-1}) + w(z_{i-1}, z_i)$. Clearly, for all $1 \leq i < n$, $\bar{\Delta}(z_i) \geq \bar{\Delta}(z_{i+1}) > 0$. Let i be maximal such that z_i was extracted from *Fh* before v. Since z_i precedes v in the topological order and by the minimality of v w.r.t. the topological order, we know that z_i was extracted when its priority was $\bar{\Delta}(z_i)$. Hence, after it was extracted, z_{i+1}'s priority became $\hbar(z_i) + \bar{\Delta}(z_i) + w(z_i, z_{i+1}) - \hbar(z_{i+1}) = \hbar'(z_i) + w(z_i, z_{i+1}) - \hbar(z_{i+1}) = \hbar'(z_{i+1}) - \hbar(z_{i+1}) = \bar{\Delta}(z_{i+1})$. Since v was extracted before z_{i+1} but when z_{i+1} was in *Fh* with priority $\bar{\Delta}(z_{i+1})$, the priority of v when it was extracted was at least $\bar{\Delta}(z_{i+1}) \geq \bar{\Delta}(v)$. ∎

Lemma 44 *For all $v \in V$, if $\bar{\Delta}(v) = 0$ then v is not inserted to Fh.*

Proof If $\bar{\Delta}(v) = 0$, this means that for every predecessor u of v, $\hbar'(u) + w(u, v) \leq \hbar(v)$. If none of the predecessors of v were inserted to *Fh* then v was never a candidate for insertion. If there are predecessors of v which were inserted to *Fh*, then whenever one of them was extracted and the edge leading from it to v was examined, $\Delta \leq 0$ so v was not inserted. ∎

Corollary 3 *The algorithm in Figure 5.2 correctly updates $\hbar(v)$ for all $v \in V$.*

5.3.3 Complexity analysis

By Lemma 43, when a node v is extracted from *Fh*, its priority is equal to $\bar{\Delta}(v)$ so $\hbar(v)$ is updated to its final value. This implies that v will never be inserted to *Fh* again; whenever another of its predecessors will be extracted from *Fh*, Δ will not be positive. In addition, each edge outgoing from a node in δ is examined once to determine whether its target should be inserted into *Fh* (or its priority updated if it is already there). We get that throughout the algorithm, *Fh* needs to support a total of $|\delta|$ insertions, $|\delta|$ extractions and at most $\|\delta\|$ *UpdatePriority* operations. The total running time is therefore $O(\|\delta\| + |\delta| \log |\delta|)$.

5.4 Integer Edge Weights

In this section we show an alternative algorithm which works for general inputs, but its advantages come to play when all edge weights are integers.

Again, for every node v let $\hbar(v)$ be the heaviest path value of v before the change to the graph and $\hbar'(v)$ the value after the edge insertion. Let $\bar{\Delta}(v) = \hbar'(v) - \hbar(v)$ be the change that occurred at v.

For every edge $e = (u, v)$, let $\Delta\bar{\Delta}(u, v) = \hbar(u) + w(u, v) - \hbar(v)$. Note that $\Delta\bar{\Delta}(u, v) \leq 0$. Intuitively, $\Delta\bar{\Delta}(u, v)$ measures by how much the change in the $\bar{\Delta}$ values decreases as the update propagates along the edge e. Formally,

Lemma 45 *For each node v, $\bar{\Delta}(v) = \max_{u \in \mathrm{pred}(v)}\{\bar{\Delta}(u) + \Delta\bar{\Delta}(u, v)\}$.*

Proof By definition, $\hbar'(v) = \max_{u \in \mathrm{pred}(v)}\{\hbar'(u) + w(u, v)\}$. So $\hbar'(v) - \hbar(v) = \max_{u \in \mathrm{pred}(v)}\{\hbar'(u) - \hbar(u) + \hbar(u) + w(u, v) - \hbar(v)\} = \max_{u \in \mathrm{pred}(v)}\{\bar{\Delta}(u) + \Delta\bar{\Delta}(u, v)\}$. ∎

We define a new weight function w_ϵ over the edges of the DAG where $w_\epsilon(u, v) = -\Delta\bar{\Delta}(u, v)$. This function enables us to prove Lemma 46 which characterizes the set δ. In the following a *path* is a directed path in the DAG and for a path $P =< v_1, \ldots, v_n >$, $w_\epsilon(P) = \sum_{i=1}^{n-1} w_\epsilon(v_i, v_{i+1})$ is the weight of the path with respect to w_ϵ. For a pair of nodes u, v, $w_\epsilon^P(u, v)$ is the minimal $w_\epsilon(P)$ over all paths P from u to v.

Lemma 46 *Assume that an edge (x, y) was inserted into G. A node v is in δ iff $w_\epsilon^P(y, v) < \bar{\Delta}(y)$. That is, there is a path P from y to v such that $w_\epsilon(P) < \bar{\Delta}(y)$.*

Proof Assume that there is such a path $P =< y = v_1, \ldots, v_n = v >$. By definition, for each $1 \leq i < n$ we have $\bar{\Delta}(v_{i+1}) \geq \bar{\Delta}(v_i) - w_\epsilon(v_i, v_{i+1})$. So $\bar{\Delta}(v_n) \geq \bar{\Delta}(y) - w_\epsilon(P) > 0$, which implies $v_n \in \delta$.

For the other direction, we show by induction that if $v \in \delta$, which means that $\bar{\Delta}(v) > 0$, then there is a path P from y to v with $w_\epsilon(P) = \bar{\Delta}(y) - \bar{\Delta}(v)$. By definition we know that there is a predecessor u of v such that $\bar{\Delta}(v) = \bar{\Delta}(u) - w_\epsilon(u,v)$. If $u = y$ then since $\bar{\Delta}(v) > 0$ we have that the edge (u,v) is a path from y to v such that $w_\epsilon(u,v) = \bar{\Delta}(u) - \bar{\Delta}(v) = \bar{\Delta}(y) - \bar{\Delta}(v)$.

Assume that $u \neq y$. Then by the induction hypothesis, there is a path P_u from y to u such that $w_\epsilon(P_u) = \bar{\Delta}(y) - \bar{\Delta}(u)$. Let P_v be the path from y to v that we get by appending the edge (u,v) to P_u. Then $w_\epsilon(P_v) = w_\epsilon(P_u) + w_\epsilon(u,v) = \bar{\Delta}(y) - \bar{\Delta}(u) + w_\epsilon(u,v) = \bar{\Delta}(y) - \bar{\Delta}(u) + \bar{\Delta}(u) - \bar{\Delta}(v) = \bar{\Delta}(y) - \bar{\Delta}(v)$. ∎

We now know how to identify the nodes of δ. We begin at y and compute shortest paths w.r.t. the weight function w_ϵ to nodes that are reached from y, but only as long as the length of the path is less than $\bar{\Delta}(y)$. The following lemma states that once we have found the length of the shortest path from y to v, we also know $\bar{\Delta}(v)$ and can update $\hbar(v)$.

Lemma 47 *Assume that an edge (x,y) was inserted into G. Let v be a node in δ and let $w = w_\epsilon^P(y,v)$. Then $\bar{\Delta}(v) = \bar{\Delta}(y) - w$.*

Proof We have shown in the proof of Lemma 46 that if $v \in \delta$ then there is a path P from y to v with $w_\epsilon(P) = \bar{\Delta}(y) - \bar{\Delta}(v)$. Assume that it is not minimal. That is, there is a path $Q =< y = v_1, \ldots, v_n = v >$ with $w_\epsilon(Q) < \bar{\Delta}(y) - \bar{\Delta}(v)$. For all $1 \leq j \leq n$, let $Q_j =< v_1, \ldots, v_j >$ and let i be minimal such that $w_\epsilon(Q_i) < \bar{\Delta}(y) - \bar{\Delta}(v_i)$. Note that $i > 1$ because $w_\epsilon(Q_1 =< v_1 >) = \bar{\Delta}(y) - \bar{\Delta}(v_1) = 0$. $w_\epsilon(Q_{i-1}) = \bar{\Delta}(y) - \bar{\Delta}(v_{i-1})$, so $w_\epsilon(v_{i-1}, v_i) = w_\epsilon(Q_i) - w_\epsilon(Q_{i-1}) < \bar{\Delta}(v_{i-1}) - \bar{\Delta}(v_i)$. By substituting $\hbar(v_i) - \hbar(v_{i-1}) - w(v_i, v_{i-1})$ for $w_\epsilon(v_{i-1}, v_i)$ and $\hbar'(v_{i-1}) - \hbar(v_{i-1}) - \hbar'(v_i) + \hbar(v_i)$ for $\bar{\Delta}(v_{i-1}) - \bar{\Delta}(v_i)$ we get that $\hbar'(v_{i-1}) + w(v_{i-1}, v_i) < \hbar'(v_i)$, contradicting the definition of $\hbar'$. ∎

This leads to the algorithm in Figure 5.4 as an alternative to the one in Figure 5.2. In this version, the nodes of the frontier are inserted into a min-sorted Fibonacci Heap where the priority of a node at any point in time is the length of the minimal path leading to it which was discovered so far.

We now prove correctness of this algorithm.

Lemma 48 *For all $v \in V$, if $w_\epsilon^P(y,v) < \bar{\Delta}(y)$ then v is inserted into Fh and when it is extracted from Fh, its priority is equal to $w_\epsilon^P(y,v)$.*

```
Function $Insert(G, (x,y), \hbar, w(x,y))$
    $Fh \Leftarrow []$ (* Empty Fibonacci Heap. *)
    $\bar{\Delta}_y = \hbar(x) + w(x,y) - \hbar(y)$
    if $\bar{\Delta}_y > 0$ then
        $Fh.Insert(y, 0)$ (* The minimum path from $y$ to $y$ has length 0. *)
    endif
    while $Fh$.NotEmpty do
        $(u, P) \Leftarrow Fh.ExtractMin$
        $\hbar(u) \Leftarrow \hbar(u) + \bar{\Delta}_y - P$ (* Update $\hbar(u)$ *)
        foreach $e = (u,v) \in E$ do
            $P' \Leftarrow P + w_\epsilon(u,v)$
            if $P' < \bar{\Delta}_y$ then
                $Fh.UpdatePriority(v, P')$
            endif
        endfor
    end while
end
```

Figure 5.4: Alternative algorithm for updating $\hbar$ values upon an edge insertion.

Proof Assume the converse and let v be minimal w.r.t. the topological order such that $w_\epsilon^P(y,v) < \bar{\Delta}(y)$ and the claim does not hold for v. By construction, the priority of a node in *Fh* can never be lower than $w_\epsilon^P(y,v)$. So we need to show that v is inserted into *Fh* and that its priority when it is extracted is not higher than $w_\epsilon^P(y,v)$.

Since y was extracted with priority $0 = w_\epsilon^P(y,y)$, $v \neq y$. By definition of w_ϵ, there must be a predecessor u of v such that $w_\epsilon^P(y,v) = w_\epsilon^P(y,u) + w_\epsilon(u,v)$. Since u precedes v in the topological order, and by the minimality of v w.r.t. the topological order, we know that u was inserted to *Fh* and was extracted with priority equal to $w_\epsilon^P(y,u)$. If v was not extracted from *Fh* before u, then after u was extracted, v was either inserted with priority $w_\epsilon^P(y,v)$ (if it was not already in *Fh*) or its priority was updated to $w_\epsilon^P(y,v)$ (if it was already in *Fh*). Hence, v was inserted to *Fh* and when it was extracted, its priority was $w_\epsilon^P(y,v)$.

Assume that v was extracted before u. Let $P =< z_1, \ldots, z_n >$ be a path from y to v through u such that $w_\epsilon(P) = w_\epsilon^P(y,v)$. That is, $z_1 = y$, $z_{n-1} = u$ and $z_n = v$ and for each $1 < i \leq n$, $w_\epsilon^P(y,z_i) = w_\epsilon^P(y,z_{i-1}) + w_\epsilon(z_{i-1},z_i)$. Clearly, for all $1 \leq i < n$,

$w_\epsilon^P(y,z_i) \le w_\epsilon^P(y,z_{i+1}) \le w_\epsilon^P(y,v) < \bar{\Delta}(y)$. Let i be maximal such that z_i was extracted from *Fh* before v. Since z_i precedes v in the topological order and by minimality of v w.r.t. the topological order, we know that z_i was extracted when its priority was $w_\epsilon^P(y,z_i)$. Hence, after it was extracted, z_{i+1}'s priority became $w_\epsilon^P(y,z_i) + w_\epsilon(z_i,z_{i+1}) = w_\epsilon^P(y,z_{i+1})$. Since v was extracted before z_{i+1} but when z_{i+1} was in *Fh* with priority $w_\epsilon^P(y,z_{i+1})$, the priority of v when it was extracted was at most $w_\epsilon^P(y,z_{i+1}) \le w_\epsilon^P(y,v)$. ∎

Lemma 49 *For all $v \in V$, if $w_\epsilon^P(y,v) \ge \bar{\Delta}(y)$ then v is not inserted to Fh.*

Proof If $w_\epsilon^P(y,v) \ge \bar{\Delta}(y)$, this means that for every predecessor u of v, $w_\epsilon^P(y,u)+w_\epsilon(u,v) \ge \bar{\Delta}(y)$. If none of the predecessors of v were inserted to *Fh* then v was never a candidate for insertion. If there are predecessors of v which were inserted to *Fh*, then by Lemma 48 whenever one of them (say u) was extracted from *Fh*, its priority was $w_\epsilon^P(y,u)$ so v was not inserted into *Fh*. ∎

Lemma 50 *The algorithm in Figure 5.4 correctly updates $\hbar(v)$ for all $v \in V$.*

Proof If $w_\epsilon^P(y,v) \ge \bar{\Delta}(y)$ then by Lemma 46, $v \notin \delta$, so $\hbar'(v) = \hbar(v)$, and by Lemma 49, v is never inserted to *Fh* so $\hbar(v)$ is never updated.

If $w_\epsilon^P(y,v) < \bar{\Delta}(y)$ then by Lemma 48, v is inserted into *Fh* and when it is extracted its priority is $w_\epsilon^P(y,v)$. So the algorithm sets $\hbar'(v)$ to $\hbar(v) + \bar{\Delta}_y - w_\epsilon^P(y,v)$, which by Lemma 47 is equal to $\hbar(v) + \bar{\Delta}_v$. ∎

5.4.1 Example

To illustrate how the algorithm works, we include here a trace of its execution on the example in Figure 5.3. Initially, $\bar{\Delta}(y) = 4+0-1 = 3$ is computed and y is inserted to *Fh* with priority 0. Then the algorithm enters the while loop and performs five iterations.

Iteration 1: $(u,P) \leftarrow (y,0)$ and $\hbar(y) \leftarrow \hbar(y) + \bar{\Delta}(y) - P = 1+3-0 = 4$. Next, the successors of y are checked. a is inserted to *Fh* with priority $P'_a = P + w_\epsilon(y,a) = 0+1 = 1$ and b is inserted with priority $P'_b = P + w_\epsilon(y,b) = 0+0 = 0$.

Iteration 2: $(u, P) \leftarrow (b, 0)$, $\hbar(b) \leftarrow \hbar(b) + \bar{\Delta}(y) - P = -1 + 3 - 0 = 2$. c is inserted to Fh with priority $P'_c = P + w_\epsilon(b, c) = 0 + 2 = 2$.

Iteration 3: $(u, P) \leftarrow (a, 1)$, $\hbar(a) \leftarrow \hbar(a) + \bar{\Delta}(y) - P = 5 + 3 - 1 = 7$. The priority of c is updated from 2 to $P + w_\epsilon(a, c) = 1 + 0 = 1$. $P'_f = P + w_\epsilon(a, f) = 1 + 2 = 3$ so f is not inserted to Fh.

Iteration 4: $(u, P) \leftarrow (c, 1)$, $\hbar(c) \leftarrow \hbar(c) + \bar{\Delta}(y) - P = 7 + 3 - 1 = 9$. d is inserted to Fh with priority $P'_d = P + w_\epsilon(c, d) = 1 + 0 = 1$.

Iteration 5: $(u, P) \leftarrow (d, 1)$, $\hbar(d) \leftarrow \hbar(d) + \bar{\Delta}(y) - P = 5 + 3 - 1 = 7$. $P'_e = P + w_\epsilon(d, e) = 1 + 9 = 10$ so e is not inserted to Fh.

Now, Fh is empty so the algorithm terminates.

5.4.2 Complexity analysis

When all edge weights are integers, we have that all w_ϵ values are non-negative integers and all priorities in the queue are in the interval $[0, \bar{\Delta}(y)]$. This means that we can use $\bar{\Delta}(y) + 1$ buckets for the nodes and place the buckets in the queue instead of the individual nodes. Since $\bar{\Delta}_y = \Delta_{\max}$ we get that if we use a Fibonacci Heap, the algorithm runs in time $O(\|\delta\| + |\delta| \log \min\{|\delta|, \Delta_{\max} + 1\})$.

If we use Thorup's integer priority queue [30] we can achieve an asymptotic running time of $O(\|\delta\| + |\delta| \log\log \min\{|\delta|, \Delta_{\max} + 1\})$. However, this priority queue is more complicated to implement.

Ausführliche Zusammenfassung

Constraint-Programming ist ein Programmierparadigma, bei dem der/die Programmierer(in) das zu lösende Problem durch Randbedingungen (engl.: *constraints*) an die Variablen des Problems modelliert. Das heißt, er/sie spezifiziert die Semantik des Problems, ohne ein Lösungsverfahren anzugeben. Dann bestimmt ein Constraint-Löser die Lösung durch Anwendung eines allgemeinen Lösungsalgorithmus in Kombination mit constraint-spezifischen Heuristiken.

Angenommen, ein Constraint-Löser wird auf eine endliche Menge von Randbedingungen über einer endlichen Menge von Variablen angesetzt, wobei jede Variable x einen endlichen Wertebereich $Dom(x)$ hat. Dann ist der Suchraum aller möglichen Variablenbelegungen endlich und kann erschöpfend durchsucht werden, indem man alle möglichen Belegungen aufzählt und für jede die Erfüllung der Randbedingungen prüft. Zur Beschleunigung verkleinern Constraint-Löser den Suchraum wiederholt durch Filterschritte, in denen Filteralgorithmen, die die Semantik der Randbedingungen kennen, Schlüsse wie den folgenden ziehen: “Es gibt keine Lösung, in der die Variable x den Wert $v \in Dom(x)$ annimmt”. Daraufhin kann v aus dem Wertebereich von x getilgt werden.

Als einfaches Beispiel dafür betrachte man die Randbedingung $X < Y$ mit $Dom(X) = \{2,3,5,6,8\}$ und $Dom(Y) = \{1,4,5,7\}$. Wie man leicht sieht, gibt es keine Lösung mit $X = 8$ und keine Lösung mit $Y = 1$. Daher können wir die Wertebereiche verkleinern auf $Dom(X) = \{2,3,5,6\}$ und $Dom(Y) = \{4,5,7\}$. Man bemerke, dass eine weitere Verkleinerung nicht möglich ist: Für jeden Wert im Bereich einer Variablen gibt es eine Lösung, in der die Variable diesen Wert annimmt. Mit anderen Worten, die Wertebereiche von X und Y sind jetzt *konsistent* mit der Randbedingung $X < Y$.

Zwei verschiedene Arten von Konsistenz spielen in der vorliegenden Arbeit eine Rolle.

Eine formale Definition wird in den entsprechenden Kapiteln gegeben, aber wir beschreiben sie nun kurz: Die erste Art von Konsistenz ist Kantenkonsistenz (*arc consistency*), die oben illustriert wurde: Gegeben eine Randbedingung C über einer Variablenmenge $X_1, \ldots, X_k$ heißen die Wertebereiche dieser Variablen kantenkonsistent, wenn es für jede Variable X_i und jeden Wert $v \in Dom(X_i)$ eine Lösung für C gibt, in der $X_i = v$ gilt. Die zweite Art von Konsistenz ist Schrankenkonsistenz (*bound consistency*). Hierbei nehmen wir an, dass jeder Wertebereich ein Intervall $Dom(X_i) = [\underline{X_i}, \overline{X_i}]$ ist. Wir nennen die Variablen schrankenkonsistent bezüglich einer Randbedingung C, wenn es für jede Variable X_i Lösungen für C gibt, so dass $X_i = \underline{X_i}$, bzw. $X_i = \overline{X_i}$ gilt. Von einem Filteralgorithmus sagen wir, dass er Kanten- bzw. Schrankenkonsistenz erreicht, wenn er kanten- bzw. schrankenkonsistente Wertebereiche für alle Variablen errechnet, ohne dass dabei eine Lösung verloren geht; d. h. der Algorithmus entfernt einen Wert v aus dem Wertebereich von X nur dann, wenn es keine Lösung der Randbedingung mit $X = v$ gibt.

Die Kapitel 2 und 3 behandeln Filteralgorithmen für zwei globale Randbedingungen, also solche, die auf eine große Anzahl von Variablen Bezug nehmen. Im Kapitel 2 beschreiben wir einen Schrankenkonsistenz-Algorithmus für die bekannte Randbedingung GCC. Zuvor waren nur effiziente Kantenkonsistenz-Algorithmen für diese Randbedingung bekannt. Zeitgleich mit unserer Arbeit hat eine andere Gruppe einen Algorithmus entworfen, der einem anderen Ansatz folgt und der Schrankenkonsistenz für einen Teil der Variablen erreichen kann [22]. In Kapitel 3 definieren wir eine neue Randbedingung *UsedBy* (und einen Spezialfall davon, den wir *Same* nennen), und wir zeigen, dass auch hierfür effiziente Kanten- und Schrankenkonsistenzberechnungen möglich sind. Unsere Filteralgorithmen folgen dem auf Flüssen und Matchings basierenden Ansatz, den erstmals Régin [25] in seinen Filteralgorithmen angewandt hat. Die Grundidee ist, die Randbedingung als einen Variablen/Werte-Graph zu modellieren, so dass jede Lösung einem zulässigen ganzzahligen Fluss in dem Graphen entspricht. Dann kann man zeigen, dass die starken Zusammenhangskomponenten des Residualgraphen angeben, welche Werte jeweils aus den Wertebereichen der Variablen entfernt werden können.

Kapitel 4 behandelt das Problem, eine topologische Sortierung der Knoten eines gerichteten kreisfreien Graphen aufrecht zu erhalten, der sich durch Kanteneinfügungen fortlaufend ändert. Wir zeigen, dass m Kanteneinfügungen in einen gerichteten kreisfreien Graphen mit n Knoten in Zeit $O(\min\{m^{3/2}\log n, m^{3/2} + n^2 \log n\})$ verarbeitet werden kann. Weiter untersuchen wir sodann die Komplexität des Verfahrens auf strukturierten Graphen. Wir zeigen, dass der Algorithmus Zeit $O(mk\log^2 n)$ benötigt, wobei k die *treewidth* des Graphen ist. Für den Spezialfall von Bäumen ($k = 1$) verbessern wir dies auf $O(n \log n)$, was optimal ist.

Kapitel 5 schließlich behandelt Algorithmen, die die Kenntnis gewichtsmaximaler Pfade in einem gerichteten kreisfreien Graphen unter Kanteneinfügungen und -löschungen aufrecht erhalten. Dieses Problem ist von Michel and Van Hentenryck [17] im Zusammenhang mit lokaler Suche (*local search*) für Constraint-Programming untersucht worden. Wie geben eine Verbesserung ihrer Algorithmen an, indem wir zeigen, wie man Graphen behandelt, die Kanten mit nicht-positiven Gewichten haben, und wie man verbesserte Komplexitätsschranken für den Fall erhält, dass alle Kantengewichte ganzzahlig sind.

Bibliography

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. Kenneth Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 32–42. Society for Industrial and Applied Mathematics, 1990.

[3] N. Beldiceanu, I. Katriel, and S. Thiel. Filtering algorithms for the Same constraint. In *Proceedings of International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR 2004)*, LNCS. Springer, 2004. to appear.

[4] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the 10th Annual European Symposium on Algorithms*, pages 152–164. Springer-Verlag, 2002.

[5] Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theor. Comp. Sc.*, 209:1–45, 1998.

[6] Y. Caseau, P.-Y. Guillo, and E. Levenez. A deductive and object-oriented approach to a complex scheduling problem. In *Deductive and Object-Oriented Databases*, pages 67–80, 1993.

[7] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 365–372. ACM Press, 1987.

[8] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. Assoc. Comput. Mach.*, 34:596–615, 1987.

[9] H.N. Gabow and R.E. Tarjan. A Linear-Time Algorithm for a Special Case of Disjoint Set Union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

[10] L. R. Ford Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, 1962.

[11] I. Katriel and S. Thiel. Fast bound consistency for the global cardinality constraint. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP 2003)*, volume 2833 of *LNCS*, pages 437–451, 2003.

[12] Irit Katriel. On algorithms for online topological ordering and sorting. Research Report MPI-I-2004-1-003, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, February 2004.

[13] A. Lopez-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, 2003.

[14] A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. On-line graph algorithms for incremental compilation. In *Proceedings of International Workshop on Graph-Theoretic Concepts in Computer Science (WG 93)*, volume 790 of *LNCS*, pages 70–86, 1993.

[15] A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. Maintaining a topological order under edge insertions. *Inf. Process. Lett.*, 59(1):53–58, 1996.

[16] K. Mehlhorn and S. Thiel. Faster Algorithms for Bound-Consistency of the Sortedness and the Alldifferent Constraint. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP 2000)*, volume 1894 of *LNCS*, pages 306–319, 2000.

[17] Laurent Michel and Pascal Van Hentenryck. Maintaining longest paths incrementally. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP 2003)*, volume 2833 of *LNCS*, pages 540–554, 2003.

[18] S.M. Omohundro, C. Lim, and J. Bilmes. The Sather language compiler/debugger implementation. Technical Report TR-92-017, International Computer Science Institute, Berkeley, March 1992.

[19] David J. Pearce and Paul H. J. Kelly. A dynamic algorithm for topologically sorting directed acyclic graphs. In *Proc. 3rd Int. Worksh. Efficient and Experimental Algorithms (WEA 2004)*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[20] J.-F. Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, pages 359–366, July 1998.

[21] C.-G. Quimper. Personal communication, October 2003.

[22] C.-G. Quimper, P. van Beek, A. Lopez-Ortiz, A. Golynski, and S. B. Sadjad. An efficient bounds consistency algorithm for the global cardinality constraint. In *Principles and Practice of Constraint Programming*, pages 600–614, 2003.

[23] G. Ramalingam and T. Reps. On competitive on-line algorithms for the dynamic priority-ordering problem. *Inf. Process. Lett.*, 51(3):155–161, 1994.

[24] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theor. Comput. Sci.*, 158(1-2):233–277, 1996.

[25] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 362–367, 1994.

[26] J.-C. Régin. Generalized Arc-Consistency for Global Cardinality Constraint. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 209–215, 1996.

[27] Neil Robertson and Paul D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *J. Algorithms*, 7:309–322, 1986.

[28] Médecins sans Frontières. http://www.doctorswithoutborders.org/.

[29] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.

[30] Mikkel Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. In *Proc. 35th ACM Symp. on Theory of Computing (STOC)*, pages 149–158, 2003.

[31] W.J. van Hoeve. The alldifferent constraint: A survey. In *Submitted manuscript. Available from http://www.cwi.nl/wjvh/papers/alldiff.pdf*, 2001.