On Minimum-Cost Assignments in Unbalanced Bipartite Graphs

Lyle Ramshaw, Robert E. Tarjan

HP Laboratories
HPL-2012-40

Keyword(s):
assignment problem; imperfect matching; minimum-cost matching; unbalanced bipartite graph; weight-scaling algorithm

Abstract:

Consider a bipartite graph $G = (X; Y ;E)$ with real-valued weights on its edges, and suppose that G is balanced, with $|X| = |Y|$. The assignment problem asks for a perfect matching in G of minimum total weight. Assignment problems can be solved by linear programming, but fast algorithms have been developed that exploit their special structure. The famous Hungarian Method runs in time $O(mn+n^2 \log n)$, where $n := |X| = |Y|$ and $m := |E|$. If the edge weights are integers bounded in absolute value by some constant C > 1, then algorithms based on weight scaling, such as that of Gabow and Tarjan, can lower the time bound to $O(m\sqrt{n} \log(nC))$.

But the graphs that arise in practice are frequently unbalanced, with $r := \min(|X|, |Y|)$ less than $n := \max(|X|, |Y|)$. Any matching in an unbalanced graph G has size at most r, and hence must leave at least $n - r$ vertices in the larger part of G unmatched. We might want to find a matching in G of size r and of minimum weight, given that size. We can reduce this problem to finding a minimum-weight perfect matching in a balanced graph G' built from two copies of G. If we use such a doubling reduction when $r << n$, however, we get no benefit from r being small.

We consider problems of this type in graphs G that are unbalanced. More generally, given any $s \leq r$, we consider finding a matching in G of size s and of minimum weight, given that size. The Hungarian Method extends easily to compute such a matching in time $O(mn+n^2 \log n)$. Note that all of the n's in the time bound for the balanced case have become either r's or s's, where $s \leq r \leq n$. But weight-scaling algorithms do not extend so easily. We introduce new machinery that enables us to compute a minimumweight matching of size s in time $O(m\sqrt{n} \log(nC))$ via weight scaling. Our techniques give some insight into the general challenge of designing efficient, matching-related algorithms.

This report's key algorithm is presented more concisely in HPL-2012-72 [18].

External Posting Date: June 6, 2012 [Fulltext] Approved for External Publication
Internal Posting Date: June 6, 2012 [Fulltext]

On Minimum-Cost Assignments in Unbalanced Bipartite Graphs

Lyle Ramshaw
HP Labs

Robert E. Tarjan
Princeton and HP Labs

Abstract

Consider a bipartite graph $G = (X, Y; E)$ with real-valued weights on its edges, and suppose that G is *balanced*, with $|X| = |Y|$. The *assignment problem* asks for a perfect matching in G of minimum total weight. Assignment problems can be solved by linear programming, but fast algorithms have been developed that exploit their special structure. The famous Hungarian Method runs in time $O(mn + n^2 \log n)$, where $n := |X| = |Y|$ and $m := |E|$. If the edge weights are integers bounded in absolute value by some constant $C > 1$, then algorithms based on *weight scaling*, such as that of Gabow and Tarjan, can lower the time bound to $O(m\sqrt{n}\log(nC))$.

But the graphs that arise in practice are frequently *unbalanced*, with $r := \min(|X|, |Y|)$ less than $n := \max(|X|, |Y|)$. Any matching in an unbalanced graph G has size at most r, and hence must leave at least $n - r$ vertices in the larger part of G unmatched. We might want to find a matching in G of size r and of minimum weight, given that size. We can reduce this problem to finding a minimum-weight perfect matching in a balanced graph G' built from two copies of G. If we use such a doubling reduction when $r \ll n$, however, we get no benefit from r being small.

We consider problems of this type in graphs G that are unbalanced. More generally, given any $s \leq r$, we consider finding a matching in G of size s and of minimum weight, given that size. The Hungarian Method extends easily to compute such a matching in time $O(ms + s^2 \log r)$. Note that all of the n's in the time bound for the balanced case have become either r's or s's, where $s \leq r \leq n$. But weight-scaling algorithms do not extend so easily. We introduce new machinery that enables us to compute a minimum-weight matching of size s in time $O(m\sqrt{s}\log(sC))$ via weight scaling. Our techniques give some insight into the general challenge of designing efficient, matching-related algorithms.

This report's key algorithm is presented more concisely in HPL-2012-72 [18].

Contents

Chapter 1

Introduction

Consider a bipartite graph $G = (V; E) = (X, Y; E)$, where the vertex set $V = X \cup Y$ is partitioned into two parts X and Y with $E \subseteq X \times Y$. We refer to the elements of X as *women* and the elements of Y as *men*.* The graph G is *balanced* when $|X| = |Y|$, so that there are the same number of women as men. Otherwise, G is *unbalanced*. (Some authors use the terms *symmetric* and *asymmetric* [3].)

For a balanced graph G, we follow tradition by using $n := |X| = |Y|$ for the number of vertices in each part and $m := |E|$ for the number of edges. For an unbalanced graph, we use $n := \max(|X|, |Y|)$ for the size of the larger part, while we introduce the symbol $r := \min(|X|, |Y|)$ for the size of the smaller part.† (Some authors use n_1 and n_2 for our r and n [1].) Note that the number of vertices in the larger part and the total number of vertices, n and $n + r$, differ by at most a factor of 2; so, in an asymptotic bound, we don't need to distinguish between them. But it can happen that r is much smaller than n. For example, $r = O(\sqrt{n})$ or $r = O(\log n)$ or even $r = O(1)$ are perfectly feasible. We call such graphs *asymptotically unbalanced* — that is, unbalanced by more than a constant factor, with $r = o(n)$. (Some authors reserve the term *unbalanced* for these graphs [1].)

Our bipartite graphs are *weighted*. So each edge in G has a weight, which is a real number that can be either positive, zero, or negative. We can interpret the weights either as *costs*, whose sums we try to minimize, or as *benefits*, whose sums we try to maximize. We can convert between those two points of view simply by negating all of the weights, so it makes little difference which point of view we adopt. And the existing literature includes both cost-minimizers and benefit-maximizers. To simplify comparisons with either camp, we introduce both a cost function $c\colon E \to \mathbb{R}$ and a benefit function $b\colon E \to \mathbb{R}$, requiring that $c(x, y) + b(x, y) = 0$ for each edge (x, y) in E. In this report, we talk mostly about minimizing cost, but with the understanding that maximizing benefit would be equivalent.

*To remember which is which, think about how sex chromosomes behave in mammals.

†As a mnemonic aid, think of n as the size of that part in which vertices are more $\underline{n}$umerous, while r is the size of that part in which vertices are more $\underline{r}$are.

A *matching* in the graph G is a set M of edges that don't share any vertices. The *size* (or *cardinality*) of a matching M is the number of edges: $s := |M|$. If (x, y) is an edge of a matching M, we refer to the woman x and the man y as *matched* or *married* in M. We define the *cost of a matching* to be the sum of the costs of its edges, and similarly for benefits:

$$c(M) := \sum_{(x,y)\in M} c(x,y) \qquad \text{and} \qquad b(M) := \sum_{(x,y)\in M} b(x,y) \tag{1-1}$$

We denote by $\nu(G)$ the maximum size of any matching in G. A balanced graph G has $\nu(G) = n$ just when matchings of size n exist in G. Such matchings are called *perfect*; they pair up each vertex in X with a vertex in Y and vice versa. An unbalanced graph G always has $\nu(G) \leq r$. If matchings of size r exist, we call them *one-sided perfect*: Every vertex in the smaller side of G is matched, but $n - r$ vertices in the larger side are left unmatched, left as either *maidens*[‡] or *bachelors*. A matching of size smaller than r is *imperfect*; it leaves both some maidens and some bachelors.

1.1 Three variants of the assignment problem

In an *assignment problem*, an output size s is somehow determined, and we then compute a matching in G of size s whose cost is minimum, among all matchings of that size. Note that we minimize cost only over the matchings of size s; matchings of other sizes are irrelevant.

We deal with three variants of the assignment problem:

Perfect Assignments (PerA) Let G be a balanced bipartite graph with edge weights. If $\nu(G) = n$, compute a min-cost perfect matching in G; otherwise, return the error code "infeasible".

Imperfect Assignments (ImpA) Let G be a bipartite graph with edge weights, either balanced or unbalanced, and let $t \geq 1$ be a target size. Compute a min-cost matching in G of size $s := \min(t, \nu(G))$.

Incremental Assignments (IncA) Let G be a bipartite graph with edge weights, either balanced or unbalanced. Compute min-cost matchings in G of sizes 1, 2, …, $\nu(G)$, presenting each in turn to the caller. The caller determines s by choosing to stop whenever satisfied.

For **ImpA** and **IncA**, our time bounds are functions of s, the size of the output matching, and are hence output sensitive. For simplicity in our time bounds, we assume that $s \geq 1$; and, when we introduce C below, we will assume that $C > 1$. We also assume that our bipartite graphs have no isolated vertices, so we have $r \leq n \leq m \leq rn$. It follows that m has to go to

[‡]English doesn't have a single word that means simply an unmarried woman; "maiden" and "spinster" both have irrelevant overtones, which are politically incorrect to boot. Indeed, when faced with this challenge, TV shows have often resorted to "bachelorette".

infinity along with n, as our graphs get large. But r can remain bounded; indeed, we can even have $r = 1$.

Historically, the problem **PerA** has been the most studied. If the graph G is both balanced and complete bipartite, then the costs of its edges can be conveniently represented using a square matrix. The resulting *linear-sum assignment problem* has been the object of much study, as described in a book by Burkard, Dell'Amico, and Martello [4].

1.2 Known algorithms for the balanced case

The algorithms for **PerA** that perform the best in practice are local methods. These algorithms maintain a matching and prices at the vertices, and their basic step is some flavor of local update, an update that changes the matching status of just one or two edges and the prices just at the vertices that those edges touch. Algorithms of this type were invented by several groups of researchers, under different names. Goldberg calls them *push-relabel* algorithms [12], and the term *preflow-push* is also used by this group; but Bertsekas calls them *auction* algorithms [2]. Even a cursory examination reveals that push-relabel and auction algorithms are similar; and Marcos Vargas analyzed a particular pair of such algorithms and showed them completely equivalent [20]. So we won't distinguish between those two families of algorithms, referring to them simply as local methods. While local methods perform the best in practice, achieving the best theoretical time bounds seems to require methods that are nonlocal, at least in part.

By the way, some simple local methods solve those instances of **PerA** in which $\nu(G) = n$ but would run forever if $\nu(G) < n$. Extra machinery is needed to guard against the lack of perfect matchings in those methods.

The granddaddy of polynomial-time algorithms for **PerA** is the famous Hungarian Method, from Kuhn in 1955 [16].[§] This method is purely global; it builds up its min-cost matching incrementally by augmenting along entire augmenting paths, paths from a maiden to a bachelor. While the Hungarian Method doesn't perform as well as local methods in practice, it is attractive from a theoretical perspective, with time bounds that have been improved by a slew of researchers over the years. Fredman and Tarjan [10] proposed using Fibonacci heaps, getting an algorithm that runs in space $O(m)$ and in time $O(mn + n^2 \log n)$. That's the current champion among strongly polynomial algorithms for **PerA**. If the weights are assumed to be integers, then Thorup [19] showed that the time can be reduced to $O(mn+n^2 \log\log n)$ by using the weights to compute the addresses of buckets.

Weight-scaling is a different way to achieve improved time bounds; like Thorup's technique, weight-scaling requires that the edge weights be integers and fails to be strongly polynomial. Using weight-scaling, we can solve the assignment problem in space $O(m)$ and in time $O(m\sqrt{n}\log(nC))$, where

[§]Going even further back, Carl Gustav Jacob Jacobi solved the assignment problem in polynomial time in the 19th century, published posthumously in 1890 in Latin.

$C > 1$ is a bound on the absolute values of the edge weights. This time bound is achieved by three different algorithms, due to Gabow and Tarjan [11], to Orlin and Ahuja [17], and to Goldberg and Kennedy [13]. Gabow-Tarjan is the most important of those three for our current purposes, and it is purely global; that is, like the Hungarian Method, Gabow-Tarjan builds its matchings by augmenting along entire augmenting paths. The other two algorithms are hybrids of local and global techniques.

A note about C: In this report, we use the symbol $\bar{C}$ to denote the maximum of the absolute values of the edge weights in the graph G:

$$\bar{C} := \max_{(x,y)\in E} |c(x,y)|. \tag{1-2}$$

Even a large graph G can have $\bar{C} = 0$, if all of the edge weights in G are precisely 0. It is this maximum $\bar{C}$ that is relevant, for example, in Section 3.8, where we show that the prices in the Hungarian Method always lie in the interval $[0\,..\,(2\ell+1)\bar{C}]$. When $\bar{C} = 0$, that interval collapses to $[0\,..\,0] = \{0\}$. In asymptotic time bounds, however, it is more convenient to work with some constant $C \geq \bar{C}$ that also has $C > 1$, so that $\log(C)$ and $\log(sC)$ are well defined and positive.

1.3 Reducing from unbalanced to balanced

The assignment problems that arise in practice are often unbalanced. One approach for solving such problems is to reduce them to balanced problems. The simplest reduction beefs up the smaller part of G to the same size as the larger part by adding $n - r$ new vertices to the smaller part and then adding zero-cost edges connecting each of those new vertices to each of the vertices in the larger part. But that involves adding $(n - r)n$ new edges, which is quadratically many. There is a better way.

A standard doubling technique lets us reduce an assignment problem for an unbalanced graph G to an assignment problem for a balanced graph G' of linear size — that is, with $n' = O(n)$ and $m' = O(m)$. We build G' by taking two copies of the input graph G and flipping one of them over, giving us a forward copy G_f and a backward copy G_b, each with the same edges and edge costs as G. We then add some *linking edges*, new edges that connect the two copies in G' of some of the vertices in G. The authors thank Marcos Vargas [20] for clarifying the roles played by these linking edges.

Suppose first that our graph G does have one-sided-perfect matchings and that our goal is to find such a matching that is min-cost; and suppose, for convenience, that X is the smaller part of G and Y is the larger part. As shown in Figure 1.1, we can then add, to G', just one linking edge for each vertex y in Y, connecting the copies of y in G_f and in G_b. And we give these large-to-large linking edges cost zero. The resulting balanced graph G' will have perfect matchings, and any such matching will consist of one-sided-perfect matchings in both G_f and G_b, with the unmatched vertices

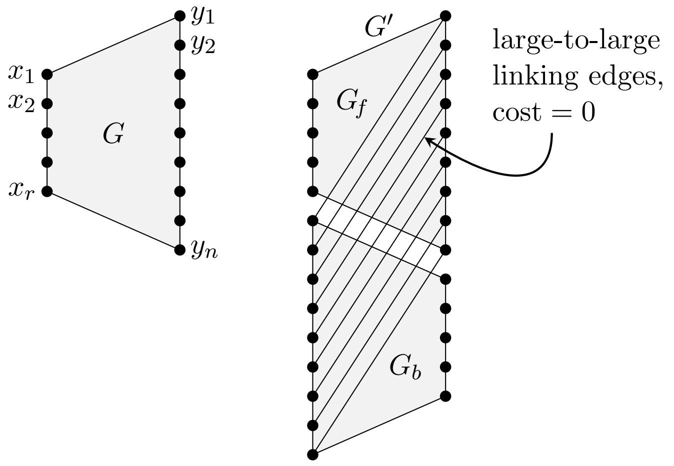

Figure 1.1: Reducing to **PerA** instances of **ImpA** with $t \geq \nu(G) = r$

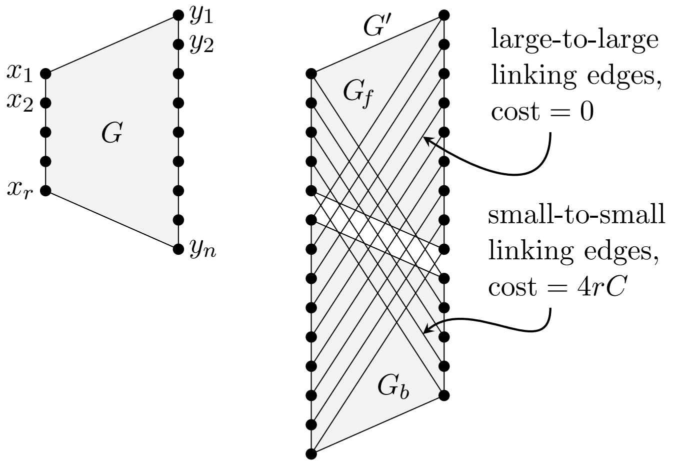
y_1
y_2
x_1
x_2
G
x_r
y_n
G'
G_f
G_b
large-to-large linking edges, cost = 0
small-to-small linking edges, cost = $4rC$

Figure 1.2: Reducing to **PerA** instances of **ImpA** with $t \geq \nu(G) < r$

paired up using $n - r$ of the large-to-large linking edges (so the matchings in G_f and G_b must match corresponding subsets of the larger part). This technique reduces to **PerA** those instances of **ImpA** in which $s = r$, that is, those instances in which $t \geq \nu(G)$ and $\nu(G) = r$.

Suppose next that we have an instance of **ImpA** with $t \geq \nu(G)$, but with $\nu(G) < r$. In addition to the large-to-large linking edges discussed above, we then also add small-to-small linking edges, as shown in Figure 1.2; but we give each of those edges a large positive cost — say, the cost $4rC$. (So the graph G' is now slightly more than linear, with $C' = O(rC)$ instead of $C' = O(C)$.) The resulting graph G' always has perfect matchings, which are of size $n' = n + r$. Indeed, all of the linking edges of both types constitute a perfect matching. But a perfect matching in G' of minimum cost must consist of matchings in G_f and G_b, each of size $\nu(G)$ and of minimum cost given that size, brought up to perfection by some $n - \nu(G)$ large-to-large linking edges and some $r - \nu(G)$ small-to-small linking edges. To see that the matchings in G_f and G_b will have size $\nu(G)$, note that the worst that can happen to the cost, when we increase the size of a matching in G by one, is that we lose $r - 1$ extremely attractive edges and replace them with r extremely expensive edges, thus increasing our cost by $(2r - 1)\bar{C}$. This could happen in both G_f and G_b; but we get to reduce by one the number of small-to-small linking edges that we need, which saves us $4rC$; so the transaction, as a whole, reduces our cost.

But these doubling reductions are unsatisfactory in various respects. They don't seem to help with instances of **ImpA** in which $t < \nu(G)$, since there is no obvious way to impose a bound less than $\nu(G)$ on the size of the matching in G that we extract from G'. And they don't seem to help with **IncA**. But the key problem with these doubling reductions, from our current perspective, is that we gain no speed advantage when $s \ll n$.

1.4 Tackling the unbalanced case directly

Rather than using some reduction, we can instead take an algorithm for the balanced case of the assignment problem and try to generalize it to handle the unbalanced case directly. Indeed, on the practical side, Bertsekas and Castañon [3] generalized an auction algorithm to work on unbalanced graphs directly.[¶] In this report, we explore that same direct approach from a theoretical perspective: We consider the time bounds for **PerA** algorithms and we try to replace as many of the n's in those time bounds as we can with r's or s's. Ahuja, Orlin, Stein, and Tarjan [1] replaced lots of n's with r's in the time bounds of network flow algorithms for bipartite graphs; but the corresponding challenge for assignment algorithms seems to be new.

The Hungarian Method is an easy success story; it generalizes nicely to handle the unbalanced case, with no new algorithmic ideas needed and with attractive resulting time bounds. Recall that the Hungarian Method, when implemented with Fibonacci heaps, solves **PerA** in time $O(mn + n^2 \log n)$. We show, in Section 3, that it solves **ImpA** in time $O(ms + s^2 \log r)$. Note that most of the n's have become s's, with a single r remaining in the logarithm that arises from the heap overhead. In fact, the Hungarian Method is incremental, so it even solves **IncA** in that same time bound. If the costs are integers and we use Thorup's technique, the $\log r$ factors in either of these bounds can be reduced to $\log \log r$.

Generalizing weight-scaling algorithms to the unbalanced case, however, turns out to be trickier. The algorithm of Goldberg and Kennedy [13] can compute imperfect matchings that are min-cost; but it isn't clear whether any of the n's in the resulting time bound can be replaced with r's or s's. Worse yet, a straightforward attempt to compute an imperfect matching with the algorithm of Gabow and Tarjan [11] or of Orlin and Ahuja [17] may result in a matching that fails to be min-cost. In Section 2.7, we derive the

[¶]We discuss an interesting aspect of their generalized algorithm in Section 4.5.

maiden and *bachelor bounds*, inequalities that relate the prices of unmatched vertices to those of matched vertices and thereby help to prove that an imperfect matching is min-cost. The Hungarian Method preserves these bounds naturally, as does Goldberg-Kennedy; but neither Gabow-Tarjan nor Orlin-Ahuja does so.

Our central result is *FlowAssign*, a weight-scaling algorithm that solves **ImpA** in time $O(m\sqrt{s}\log(sC))$; but *FlowAssign* is not incremental, so it doesn't solve **IncA**. Roughly speaking, *FlowAssign* is Gabow-Tarjan with dummy edges to a new source and sink added, to enforce the maiden and bachelor bounds. *FlowAssign* also simplifies Gabow-Tarjan in two respects. First, Gabow-Tarjan adjusts some prices as part of augmenting along an augmenting path. Those price adjustments turn out to be unnecessary, and we don't do them in *FlowAssign* (though we could, as Section 10.1 discusses). Second, a person posing an assignment problem sometimes wants prices that, through complementary slackness, prove that the output matching is indeed min-cost. Gabow and Tarjan compute such prices in a $O(m)$ postprocessing step. In *FlowAssign*, we compute such prices simply by rounding, to integers, the prices that we have already computed, that rounding taking time $O(n)$.

FlowAssign has an attractive theoretical time bound, and it deals with unbalanced graphs without the overhead of a doubling reduction. But it is purely global, building all of its matchings by augmenting along entire augmenting paths. As such, its performance in practice may not be all that attractive. Both Goldberg-Kennedy and Orlin-Ahuja are hybrid algorithms, using local techniques for many of their updates and mixing in only enough global updates to achieve good time bounds — in the balanced case. To find an algorithm for the unbalanced case that performs well in both theory and practice, it might be good to aim for such a hybrid algorithm.

1.5 The matching problem

Two problems related to the assignment problem are worth mentioning. In this section, we consider the *matching problem*, where we simply search for any matching of the relevant size s. We don't try for a matching that is min-cost, and, indeed, we ignore any edge weights that might exist.

We consider three variants of the matching problem, analogous to our three variants of the assignment problem:

Perfect Matchings (PerM) If G has any perfect matchings, return one; otherwise, return the error code "infeasible".

Imperfect Matchings (ImpM) Let G be a bipartite graph and let $t \geq 1$ be a target size. Compute a matching in G of size $s := \min(t, \nu(G))$.

Incremental Matchings (IncM) Let G be a bipartite graph. Compute matchings in G of sizes 1, 2, …, $\nu(G)$, presenting each in turn to the caller. The caller determines s by choosing to stop whenever satisfied.

The most famous algorithm for the matching problem was proposed by Hopcroft and Karp [14]. As published, Hopcroft-Karp computes a matching of size $\nu(G)$ in time $O(m\sqrt{\nu(G)})$, thus solving those instances of **ImpM** that have $t \geq \nu(G)$. We show, in Section 5, that Hopcroft-Karp actually solves all instances of **ImpM**, and even **IncM** as well, in time $O(m\sqrt{s})$. Our algorithm *FlowAssign* exploits this by using Hopcroft-Karp to solve an instance of **ImpM** as part of its initialization.

Consider computing a max-size matching — that is, a matching of size $\nu(G)$ — in a balanced graph G, and in a context where output-sensitive time bounds aren't interesting; Hopcroft-Karp does this in time $O(m\sqrt{n})$. If the graph G is sufficiently dense, then Feder and Motwani [9] show how to improve on Hopcroft-Karp by a factor of as much as $\log n$. In particular, they compute a max-size matching in time $O(m\sqrt{n}\log(n^2/m)/\log n)$. If m is nearly n^2, then the log factor in the numerator is small and we are essentially dividing by $\log n$. But the improvement drops to a constant as soon as m is $O(n^{2-\epsilon})$, for any positive ϵ.

We won't be applying the Feder-Motwani technique in this report. But it would be interesting to generalize their technique to the unbalanced case. By using a doubling reduction, their algorithm can find a max-size matching in an unbalanced graph in the time bound given above. But it isn't clear whether that time could be improved by tackling the unbalanced case directly — perhaps improved to $O(m\sqrt{r}\log(rn/m)/\log n)$.

1.6 Maximum-weight matchings

We now return to bipartite graphs with weights on their edges, and we tackle the *maximum-weight matching* problem **MWM**: the problem of computing a matching that has the maximum possible benefit (and hence the minimum possible cost), among all matchings of any size whatever. Duan and Su [8] recently found a weight-scaling algorithm for the balanced case of **MWM** that runs in space $O(m)$ and in time $O(m\sqrt{n}\log C)$. Thus, they managed to reduce the logarithmic factor of the typical weight-scaling bound from $\log(nC)$ to $\log C$. They discuss the intriguing open question of whether a similar reduction can be achieved for the assignment problem, where the optimization is over matchings of some fixed size.

Like most researchers in this area, Duan and Su did not consider the asymptotically unbalanced case, so they made no attempt to replace n's with r's. Their algorithm might generalize straightforwardly, thus solving the unbalanced case of **MWM** in time $O(m\sqrt{r}\log C)$; we leave that as another open question.

By the way, we can easily reduce from the unbalanced case of **MWM** to **ImpA**, with no doubling needed. As shown in Figure 1.3, we simply add r new vertices to the larger part of G, and we add r new, zero-weight edges that connect these new vertices to the vertices in the smaller part of G. The resulting graph G' is even more unbalanced than G, but it clearly

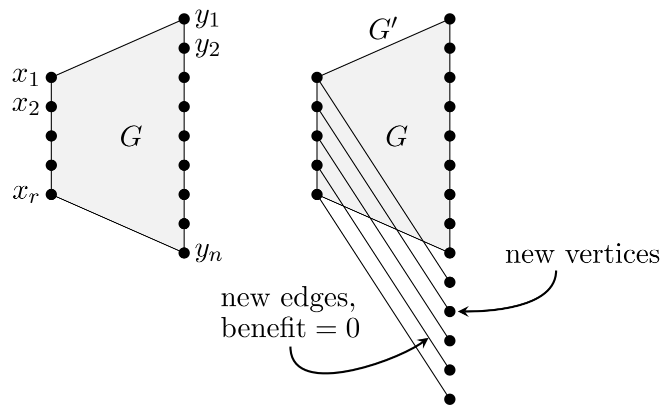

Figure 1.3: Reducing from **MWM** to **ImpA**

has one-sided-perfect matchings, and such a matching of maximum benefit gives us a max-weight matching in G. By using *Flow-Assign* to solve the resulting instance of **ImpA**, we can solve the unbalanced case of **MWM** in time $O(m\sqrt{r}\log(rC))$. Thus, we can reduce the $\sqrt{n}$ to a $\sqrt{r}$, but only at the price of putting the r factor back into the argument to the logarithm.

Finally, recall that Feder and Motwani showed how to speed up Hopcroft-Karp a bit, for quite dense graphs. Suppose that we have a fairly dense, balanced bipartite graph G with positive edge weights, but most of those weights are quite small; and we want to compute a max-weight matching in G. If all of the weights were precisely 1, then a max-weight matching would be the same as a max-size matching, so we could use Feder-Motwani. Kao, Lam, Sung, and Ting [15] showed that a similar improvement is possible as long as most of the edge weights are quite small. Assuming that the edge weights are positive integers and letting W denote the total weight of all of the edges in G, they compute a max-weight matching in time $O(\sqrt{n}\,W\log(n^2C/W)/\log n)$. When $C = O(1)$ and hence $W = \Theta(m)$, their bound matches that of Feder and Motwani. But they continue to achieve improved performance until W gets up around $m\log(nC)$, at which point we are better off reducing to an assignment problem. (We could reduce to **ImpA** as in Figure 1.3 and then apply *FlowAssign*; or we could reduce to **PerA** by using the doubling reduction in Figure 1.2, but with the weights of both the large-to-large and small-to-small linking edges set to zero.)

If someone manages to generalize Feder-Motwani to the asymptotically unbalanced case, it might then be worthwhile to consider similarly generalizing the Kao-Lam-Sung-Ting result. The main issue would be replacing their initial $\sqrt{n}$ with a $\sqrt{r}$.

Chapter 2

From matchings to flows

We begin by constructing a flow network N_G from the given bipartite graph G. This converts the problem of finding min-cost matchings in G to the problem of finding min-cost integral flows in N_G. This construction is quite standard, but our version has some wrinkles: We renounce a skew-symmetry that most authors adopt, and we introduce the new term "flux".

2.1 Building the flow network N_G

Figure 2.1 shows an example of how we construct, from an undirected graph G, a directed graph N_G that we call the *flow network* of G. We imagine shipping various quantities of some commodity — perhaps liters of water — along the various arcs of N_G. For example, we might ship $f(v, w)$ liters of water from node v to node w along the arc $v \to w$.

Each arc $v \to w$ in any of our flow networks will have a maximum capacity of one unit of flow and will have a per-unit-of-flow cost, which we denote $c(v, w)$. If we ship $f(v, w)$ units of flow along the arc $v \to w$, then the cost that we accrue from this arc is the product $f(v, w)c(v, w)$.

By the way, the per-unit-of-flow cost $c(v, w)$ is frequently positive; but one can imagine situations where it would be negative. For example, the node v might be at a higher altitude than the node w, and we might be able

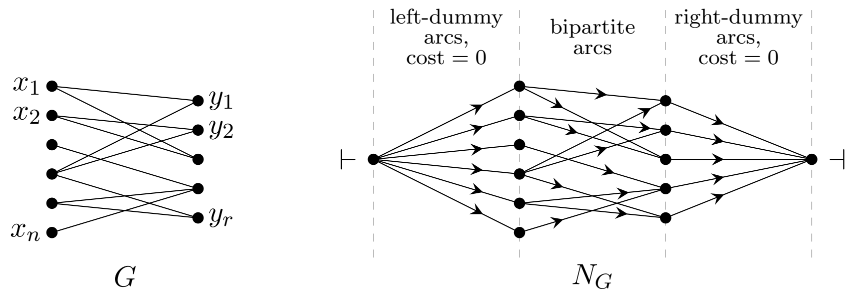

Figure 2.1: Converting a bipartite graph G into the flow network N_G

to generate hydroelectric power as a result of shipping a liter of water from v to w, thus leading to $c(v,w) < 0$.

The flow network N_G has one node for every vertex in G and has two special nodes called the *source* and the *sink*, which we denote by $\vdash$ and $\dashv$. Each edge (x,y) in G gives rise to a directed arc $x \to y$ in N_G, directed from x toward y. We refer to the arcs that arise in this way as *bipartite* arcs. The per-unit cost of the bipartite arc $x \to y$ is $c(x,y)$, the cost of the corresponding edge in G. In addition to the bipartite arcs, the network N_G includes *dummy* arcs. For each vertex x in X, there is a *left-dummy* arc $\vdash \to x$, directed from the source node $\vdash$ to the node x. The per-unit cost of a left-dummy arc is zero: $c(\vdash, x) := 0$. Symmetrically, for each vertex y in Y, the network N_G includes a *right-dummy* arc $y \to \dashv$, directed from the node y to the sink node $\dashv$ and of cost zero: $c(y, \dashv) := 0$.

Warning: Most authors set up their flow networks to include a backward version of each arc, along with the forward version. They then define the functions f and c that measure flow quantity and per-unit cost to be skew-symmetric, with $f(w,v) = -f(v,w)$ and $c(w,v) = -c(v,w)$. This approach has some advantages, but we do not adopt it. Instead, our flow networks have only forward arcs, oriented from left to right. Thus, if we ever talk about an arc $v \to w$ in a flow network N_G, we must have either:

- $v \to w$ is a left-dummy arc, with $v = \vdash$ and $w \in X$;
- or $v \to w$ is a bipartite arc, with $v \in X$ and $w \in Y$;
- or $v \to w$ is a right-dummy arc, with $v \in Y$ and $w = \dashv$.

We avoid backward arcs because *FlowAssign* quantizes the net costs of arcs in a one-sided manner, using "ceiling quantization", as we discuss in Section 6.1. If we included backward arcs, we would have to use floor quantization on them, to be consistent with the ceiling quantization that we use on our forward arcs. It seems simpler to avoid backward arcs entirely.

Of course, our augmenting paths will have to take both forward steps, adding a new edge to the matching, and backward steps, removing an edge from the matching. But our augmenting paths will be paths in an auxiliary graph called the *residual digraph*. To avoid confusion, we use different terms and notations for our three different levels of graphs. The original bipartite graph G has *vertices* and *edges*, where a typical edge is written (x,y). The flow network N_G has *nodes* and *arcs*, where a typical arc is written $v \to w$, and all arcs go forward. The residual digraph will have *nodes* and *links*, where a typical link is written $v \Rightarrow w$, and some links go forward while others go backward.

2.2 Of matchings and integral flows

There is no standard term for a function f that assigns some real-valued flow $f(v,w)$ to each arc $v \to w$ in a flow network, with no restrictions whatsoever. Let's call such a function f a *flux*.

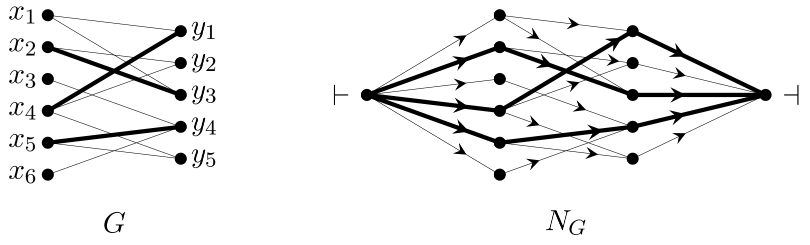

Figure 2.2: A matching M in a bipartite graph G and the corresponding integral flow f on the flow network N_G. The matching M has three edges and is imperfect, leaving x_1, x_3, and x_6 as maidens and leaving y_2 and y_5 as bachelors.

A *pseudoflow* is a flux in which the flow $f(v, w)$ along each arc $v \to w$ is nonnegative and satisfies the unit-capacity constraint: $0 \le f(v, w) \le 1$. If f is a pseudoflow, we refer to an arc $v \to w$ with $f(v, w) = 0$ as *idle* in f, to an arc with $f(v, w) = 1$ as *saturated* in f, and to an arc with $0 < f(v, w) < 1$ as having *fractional flow* in f.

A *flow* is a pseudoflow in which flow is conserved at all nodes except for the source and the sink; that is, the total flow entering each such node is the same as the total flow leaving it. The *value* of a flow f, denoted $|f|$, is the total flow out of the source, which is also the total flow into the sink – and, for that matter, the total flow over all of the bipartite arcs. ("Circulations" and "preflows" are other flavors of fluxes, for which we have no current need.)

A flux f is *integral* when, for every arc $v \to w$, the flow $f(v, w)$ over that arc is an integer. We will typically be dealing with fluxes, pseudoflows, and flows that are integral. If a pseudoflow on some flow network is integral, then every arc is either idle or saturated — no arcs have fractional flow.

Given any flux f in the flow network N_G, we define the *cost* of that flux to be the sum of the costs of its arcs:

$$c(f) := \sum_{v \to w \,\in\, N_G} f(v, w) c(v, w). \tag{2-1}$$

Recall that our flow network N_G has only forward arcs, so we have no need to divide by 2 in this formula, as the fans of skew-symmetry must do. Also, since all of the dummy arcs in N_G are zero-cost, only the bipartite arcs actually contribute to the sum in (2-1).

Prop 2-2. *Matchings M in the bipartite graph G naturally correspond to integral flows f in the flow network N_G. In this correspondence, the size of the matching is the value of the flow: $|M| = |f|$. And the cost of the matching is the cost of the flow: $c(M) = c(f)$. Thus, a min-cost matching of some size s corresponds to a min-cost integral flow of value s.*

Proof. Figure 2.2 shows an example. Given a matching, the edges in that matching tell us which bipartite arcs should be saturated in the corresponding flow, and then conservation of flow determines which dummy arcs have to be saturated. Conversely, given a flow, the saturated bipartite arcs of that flow become the edges of the matching. □

2.3 The dual variables as prices

With Prop 2-2 in mind, computing a min-cost matching of some size s in the bipartite graph G is the same as computing a min-cost integral flow of value s in the flow network N_G. If we drop the integrality constraint, the latter problem is a linear program. So we now appeal to concepts from linear programming, but stripped down for this situation. In particular, rather than saying that a flow f and prices p "satisfy complementary slackness", we will abbreviate by saying that the pair (f, p) is *proper*.

We invent a dual variable for each node in the flow network N_G, whose value we can interpret as the per-unit price of the commodity at that node. But here we face another choice of sign: Are we talking about the price that we would be charged to acquire a unit of the commodity at this node or the price that we would be charged, at this node, to dispose of a unit of the commodity? (The latter assumes, of course, that we dispose of our extra units of commodity in an ecologically responsible manner, which may involve some cost; we don't simply dump those units in a ditch on a dark night.) This choice of sign is analogous, in some ways, to the choice between costs and benefits; but the two choices are independent.

In this report, we are trying to accommodate both benefit-maximizers and cost-minimizers by introducing notations for both the benefit of an edge and its cost: $b(x, y) + c(x, y) = 0$. In an analogous way, let's introduce notations for both *acquire prices* and *dispose prices*.[*] So, for any node v in the network N_G, let $p_a(v)$ be the per-unit price to acquire the commodity at v, while $p_d(v)$ is the per-unit price to dispose of the commodity at v. We always have $p_a(v) + p_d(v) = 0$, for all nodes v. But both of these prices can have either sign; for example, gold typically has a positive acquire price and a negative dispose price, while asbestos has the reverse.

Warning: The distinction between the acquire price of a commodity and its dispose price is not at all the same as the distinction — say, for a precious metal — between its buying price and its selling price. The selling price is the acquire price: the money that you must pay to a dealer to acquire the commodity. But the buying price is the money that a dealer will pay you to dispose of the commodity for you; so the buying price is the negative of the dispose price. In our idealized economy, we assume that the acquire and dispose prices sum to zero, which means that the buying price and the selling price always coincide.

2.4 Net costs of arcs and flows

Suppose that we are given prices at all of the nodes in the flow network N_G; we'll refer to these prices as p, from which we can compute both $p_a(v)$ and $p_d(v)$ for any node v. It then makes economic sense to adjust the per-unit

[*]It is serendipitous that the words "cost", "benefit", "acquire", and "dispose" start with the first four letters of the alphabet.

cost $c(v, w)$ of each arc $v \to w$ to account for the difference in prices between the nodes v and w. We'll refer to this adjusted cost as the *net cost* of the arc $v \to w$, and we'll denote it as $c_p(v, w)$. Each arc $v \to w$ also has a *net benefit* $b_p(v, w)$, which satisfies $b_p(v, w) + c_p(v, w) = 0$. (The term *reduced cost* is often used, rather than *net cost*, both in assignment algorithms and in linear programming more generally. But "net cost" seems more apt, and is shorter to boot.) A moment of economic thought reveals the proper signs to use in the formulas that compute the net cost from the cost and the prices. There are four such formulas, since we might be keeping track of either costs or benefits and we might be using either acquire prices or dispose prices:

$$c_p(v, w) := c(v, w) + p_a(v) - p_a(w) = c(v, w) - p_d(v) + p_d(w) \tag{2-3}$$
$$b_p(v, w) := b(v, w) - p_a(v) + p_a(w) = b(v, w) + p_d(v) - p_d(w) \tag{2-4}$$

Warning: In work on the balanced case of the assignment problem, it has been traditional to compute net costs using a formula like $c_p(x, y) := c(x, y) + p(x) + p(y)$ or $c_p(x, y) = c(x, y) - p(x) - p(y)$, where the prices at x and at y enter with the same sign; and, indeed, the term "reduced cost" suggests a formula of this type. To get such a formula, acquire prices are used in one of the two parts of the bipartite graph G, while dispose prices are used in the other. The undirected graph underlying our network N_G is bipartite, so we could choose to continue this tradition. For example, we could use acquire prices at nodes in X and at the sink $\dashv$, but dispose prices at the nodes in Y and at the source $\vdash$. Our adjustment formulas would then have two prices with the same sign; but arcs of different types would have different adjustment formulas:

$$\begin{aligned} c_p(\vdash, x) &:= c(\vdash, x) - p_d(\vdash) - p_a(x) && \text{for a left-dummy arc } \vdash \to x \\ c_p(x, y) &:= c(x, y) + p_a(x) + p_d(y) && \text{for a bipartite arc } x \to y \\ c_p(y, \dashv) &:= c(y, \dashv) - p_d(y) - p_a(\dashv) && \text{for a right-dummy arc } y \to \dashv. \end{aligned}$$

It could be confusing to keep track of which formula applies in which cases. Once we add a source and sink to our network, it seems simpler to use either acquire prices throughout or dispose prices throughout.

But which prices shall we adopt: acquire or dispose? If we use dispose prices, it will turn out that the price changes in our Hungarian searches will be price increases, which are more familiar from real life than price decreases; so let's do that. From now on, when we say "price" without further specification, we mean "dispose price". So the adjustment formula that we will use most often is $c_p(v, w) = c(v, w) - p_d(v) + p_d(w)$.

In equation (2-1), we defined the cost of a flux f to be the sum of the costs of its arcs. Given any prices p on the nodes of the network N_G, we define the *net cost* of the flux f in the analogous way:

$$c_p(f) := \sum_{v \to w \,\in\, N_G} f(v, w) c_p(v, w). \tag{2-5}$$

If the flux f is actually a flow, then there is a simple relationship between its cost and its net cost. We have

$$\begin{aligned} c_p(f) &= \sum_{v \to w \,\in\, N_G} f(v,w) c_p(v,w) \\ &= \sum_{v \to w \,\in\, N_G} f(v,w)\big(c(v,w) - p_d(v) + p_d(w)\big) \\ &= c(f) - |f|\big(p_d(\vdash) - p_d(\dashv)\big). \end{aligned} \tag{2-6}$$

To justify that last step, note that the value $|f|$ is the total flow out of the source $\vdash$ and into the sink $\dashv$, while flow is conserved at all other nodes.

2.5 Proper arcs

The theory of linear programming gives us a concept and a result.

Definition 2-7. Let f be a pseudoflow on the network N_G and let p specify prices at the nodes of N_G. We refer to the pair (f, p) as *proper* when every arc $v \to w$ that has $c_p(v,w) > 0$ has $f(v,w) = 0$ and every arc $v \to w$ that has $c_p(v,w) < 0$ has $f(v,w) = 1$. In words, every arc with positive net cost is idle and every arc with negative net cost is saturated.

If a pseudoflow f and prices p form a proper pair (f, p), it is clear that f has the smallest net cost $c_p(f)$ that is possible for any pseudoflow under the prices p, since each arc's contribution to the sum in (2-5) is minimized. If f is a flow, however, the cost $c(f)$ is also minimum, in a certain sense.

Prop 2-8. *Let f be a flow on the flow network N_G and let p specify prices at the nodes of N_G. If the pair (f, p) is proper, then the cost $c(f)$ is minimum, among all flows of value $|f|$.*

In Appendix A, we prove Prop 2-8 as an instance of the duality of linear programming. For completeness, however, we prove it here directly.

Proof. Let f' be any flow in the network N_G with $|f'| = |f|$. We will show that $c(f') \geq c(f)$ by calculating the net cost $c_p(f' - f)$ of the difference flux $f' - f$ in two different ways. Note, by the way, that the flux $f' - f$ might not be a pseudoflow, since it might assign negative flow to some arcs.

We first calculate the net cost on an arc-by-arc basis:

$$c_p(f' - f) = \sum_{v \to w \,\in\, N_G} (f' - f)(v,w)\, c_p(v,w)$$

Each term in this sum is nonnegative: If $c_p(v,w) > 0$, properness implies that $f(v,w) = 0$, so $(f' - f)(v,w) \geq 0$; and, if $c_p(v,w) < 0$, properness implies $f(v,w) = 1$, so $(f' - f)(v,w) \leq 0$. It follows that $c_p(f' - f) \geq 0$.

Calculating in a different way, we can split up the flux $f' - f$ into its two constituent flows and use equation (2-6) on each:

$$\begin{aligned} c_p(f' - f) &= c_p(f') - c_p(f) \\ &= \big(c(f') - |f'|(p_d(\vdash) - p_d(\dashv))\big) - \big(c(f) - |f|(p_d(\vdash) - p_d(\dashv))\big) \end{aligned}$$

Since the flows f and f' have the same value, the correction terms cancel. So we have $c_p(f' - f) = c(f') - c(f) \geq 0$, and we conclude that $c(f') \geq c(f)$, as required. □

Definition 2-7 tests whether an arc is proper by using the arc's net cost to constrain its flow. We can turn this around, using the arc's flow to constrain its net cost. Taking contrapositives of the conditions in Definition 2-7, arcs that are not idle must have nonpositive net cost, while arcs that are not saturated must have nonnegative net cost. Combining these, it follows that arcs with fractional flow must have zero net cost. So here is an equivalent way to define what it means to be proper.

Definition 2-9. Given a pseudoflow f on the flow network N_G and prices p at the nodes of N_G, we define an arc that is idle in f to be *proper* when its net cost is nonnegative; we define an arc that is saturated in f to be *proper* when its net cost is nonpositive; and we define an arc that has fractional flow in f to be *proper* only when its net cost is zero. We say that the pair (f, p) is *proper* when all of the arcs in N_G — whether idle, saturated, or with fractional flow — are proper.

Note that, if the pseudoflow f is integral, as ours will typically be, then there are no arcs with fractional flow; so the third case doesn't arise.

Corollary 2-10. *Let f be an integral flow in the flow network N_G of value $s := |f|$ and suppose that we can find prices p at the nodes of N_G that make the pair (f, p) proper. The cost $c(f)$ is then minimum, among all integral flows of value s. It follows that the matching in G that corresponds to f is min-cost, among all matchings of size s.*

2.6 The perspective of linear programming

We now discuss some connections with the theory of linear programming.

Given the flow network N_G and a specified value s, it is a linear program to minimize the cost $c(f)$ of a flow f in N_G of value s. We refer to this as the *primal problem*. The decision variables of the primal problem are the flows $f(v, w)$ along the various arcs in N_G.

The *dual problem* has, as its decision variables, the prices at the nodes of N_G. (The dual problem also has decision variables associated with the arcs of N_G, but those variables play a minor role, as shown in Appendix A.) Given any prices at the nodes, the objective function of the dual gives a lower bound on the cost of any flow of value s. The goal of the dual problem is to maximize the resulting lower bound.

Let's now assume that $s \leq \nu(G)$, so that the primal is feasible. The theory of linear programming tells us that there will exist pairs (f, p) where f is a solution of the primal, p is a solution of the dual, and the primal objective at f matches the dual objective at p. In any such pair, f and p will satisfy a condition called *complementary slackness*. Furthermore, if complementary slackness does hold for some pair (f, p), then both the primal solution f and the dual solution p are optimal. Appendix A shows that complementary slackness is precisely the condition that we are calling properness.

A word about integrality: Even when all of the numbers that arise in specifying a linear program are integers, it may easily happen that the only optimal solutions are nonintegral. If that happened in our primal program, it would be bad, since we are hoping for an integral flow f in the network N_G, which we can then convert by Prop 2-2 into a matching M in the graph G. There is a high-powered theory to which we could appeal, to ensure that this bad thing won't happen. In particular, the coefficient matrix that arises in our primal program is totally unimodular, and the capacity constraints on the arcs in N_G are all integers — in fact, are all 1. This guarantees that, as long as any flows of value s exist, then min-cost flows of value s will exist that are integral. In this report, however, we don't need to appeal to that high-powered theory. Both of the algorithms that we consider — the Hungarian Method and our weight-scaling algorithm *FlowAssign* — compute an integral flow f and prices p that make the pair (f, p) proper. And those proper prices demonstrate that the integral f that we have computed is an optimal solution to the primal.

In fact, in *FlowAssign*, we are going to have integrality also for the dual. The coefficient matrix for the dual program is the transpose of the primal matrix, so the dual matrix is also totally unimodular. In addition, when we are doing weight-scaling, we require that the edge costs $c(x, y)$ all be integers. By that same high-powered theory, we can thus conclude that the dual program has optimal solutions that are integral. And, indeed, both the flow f and the prices p that *FlowAssign* computes will be integral.

2.7 The maiden and bachelor bounds

Let G be a weighted bipartite graph, let M be a matching of size $s := |M|$, and suppose that we hope to use Corollary 2-10 to show that the cost of M is minimum, among matchings of size s. Let f denote the integral flow of value $s = |f|$ that corresponds to M as in Prop 2-2; recall that Figure 2.2 gave an example. To apply Corollary 2-10, we must come up with prices p at the nodes of N_G that make all of the arcs of N_G proper. In particular, the left-dummy arcs must be proper and ditto for the right-dummy arcs. This turns out to boil down to two inequalities that we call the "maiden bound" and the "bachelor bound".

Having somehow chosen some prices p at the nodes of N_G, will the left-dummy arcs be proper? If x is a married woman in the matching M, then the left-dummy arc $\vdash \to x$ will be saturated in the flow f. For this arc to be proper, we must have $c_p(\vdash, x) = c(\vdash, x) - p_d(\vdash) + p_d(x) \leq 0$, which, since the cost $c(\vdash, x)$ is zero, means $p_d(x) \leq p_d(\vdash)$. In words, the (dispose) price at any married woman must be at most the (dispose) price at the source. On the other hand, if x is a maiden, then the left-dummy arc $\vdash \to x$ will be idle, so we must arrange that $p_d(\vdash) \leq p_d(x)$. In words, the price at the source must be at most the price at any maiden. Putting these together, we conclude that the left-dummy arcs will all be proper just when

$$\max_{x \text{ married}} p_d(x) \leq p_d(\vdash) \leq \min_{x \text{ maiden}} p_d(x). \tag{2-11}$$

To achieve this, it had better be the case that the maidens are the most expensive women. We refer to this inequality as the *maiden bound*:

$$\max_{x \text{ married}} p_d(x) \leq \min_{x \text{ maiden}} p_d(x). \tag{2-12}$$

If the maiden bound holds, then there is a closed interval in which we can choose the price at the source so as to make all left-dummy arcs proper.

The right-dummy arcs are a similar story. If y is a married man, then the right-dummy arc $y \to \dashv$ is saturated, and we must have $c_p(y, \dashv) = c(y, \dashv) - p_d(y) + p_d(\dashv) \leq 0$, which means $p_d(\dashv) \leq p_d(y)$. So the price at the sink must be at most the price at any married man. But the price at any bachelor must be at most the price at the sink, so the right-dummy arcs will all be proper just when

$$\max_{y \text{ bachelor}} p_d(y) \leq p_d(\dashv) \leq \min_{y \text{ married}} p_d(y). \tag{2-13}$$

To make this possible, it had better be the case that the bachelors are the cheapest men, as specified by the *bachelor bound*:

$$\max_{y \text{ bachelor}} p_d(y) \leq \min_{y \text{ married}} p_d(y). \tag{2-14}$$

When there are no maidens, choosing the price $p_d(\vdash)$ at the source to be anything large enough makes all left-dummy arcs proper. And, when there are no bachelors, choosing the price $p_d(\dashv)$ at the sink to be anything small enough makes all right-dummy arcs proper. So we can prove that a perfect matching is min-cost without worrying about the maiden and bachelor bounds. We do need to worry when our matching is less than perfect, however. In this sense, **ImpA** and **IncA** are somewhat harder problems than **PerA**, which might explain why **PerA** has been more studied.

Chapter 3

The Hungarian Method

Like many of the assignment algorithms that followed it, the Hungarian Method operates on the bipartite graph G itself, rather than on some flow network constructed from G. To ensure that its matchings are min-cost, the Hungarian Method computes prices that, from our perspective, make all of the bipartite arcs of the flow network N_G proper. The maiden and bachelor bounds are thus relevant. If they hold, then we can choose prices at the source and sink that make all of the dummy arcs proper as well, at which point Corollary 2-10 will guarantee that the matching is min-cost. Fortunately, if the Hungarian Method is implemented carefully, it naturally preserves the maiden and bachelor bounds; so it generalizes neatly to the unbalanced case. In fact, it solves **IncA**, which is the hardest of our three assignment variants, in space $O(m)$ and time $O(ms + s^2 \log r)$. Its high-level structure is shown in Figure 3.1.

```
HungarianMethod($G$)
    set $M$ to the empty matching;
    set prices at women to 0, at men to $\bar{C}$;
    for $s$ from 0 by 1 until stopped do
        announce($M$ is min-cost of size $s$);
        use Dijkstra to build a shortest-path forest
                with roots at all remaining maidens;
        if some bachelor $\beta$ was reached then
            raise prices to tighten the tree path to $\beta$;
            augment $M$ along that tight path;
        else
            announce($\nu(G) = s$);
            return($M$);
        fi;
    od;
```

Figure 3.1: Pseudocode for the Hungarian Method

Flipping the bipartite graph G over, swapping the roles of X and Y, doesn't change the assignment problem. But we get a better time bound for our implementation of the Hungarian Method if women are in the majority. Let's flip, if necessary, to ensure that $n = |X| \geq |Y| = r$. (So we would flip the graph in Figure 1.1, but not the one in Figure 2.1.)

3.1 Two invariants on the net costs of arcs

We view the Hungarian Method as taking place on the flow network N_G. But the source and sink nodes and the dummy arcs play no role until we prove, in Section 3.7, that the matchings computed by the Hungarian Method are indeed min-cost. Until then, only the bipartite arcs in N_G are relevant. For brevity, though, we will talk about the edges in G, rather than bipartite arcs in N_G. So we say that an edge in G is either *saturated* or *idle*, according as it does or does not belong to the current matching M.

The Hungarian Method associates prices with the nodes in X and Y, and we here assume that those prices are dispose prices. We maintain, as our first invariant, that every idle edge (x, y) in G has nonnegative net cost: $c_p(x, y) \geq 0$. We also maintain, as our second invariant, that every saturated edge (x, y) has zero net cost: $c_p(x, y) = 0$. Note that a saturated edge is proper as long as its net cost is nonpositive; so we are doing more than keeping our saturated edges proper. Edges with zero net cost are called *tight*; so the Hungarian Method keeps its saturated edges tight.

We establish these two invariants at the outset by starting with the empty matching, so all edges are idle. To guarantee that all edges have nonnegative net cost, we set $p_d(x) := 0$, for all x in X, and $p_d(y) := \bar{C}$, for all y in Y. Recall that $\bar{C} := \max_{(x,y)\in G} |c(x, y)|$. As a result, the net cost of the edge (x, y) is $c_p(x, y) = c(x, y) - p_d(x) + p_d(y) = c(x, y) + \bar{C} \geq 0$.

3.2 The residual digraph

The Hungarian Method builds up its matching by augmenting along tight augmenting paths, where these augmenting paths are paths in an auxiliary directed graph called the *residual digraph*.

Given a matching M in G, the *residual digraph* R_M has nodes and links that correspond precisely to the vertices and edges of G, but each saturated edge (x, y) becomes a backward-directed link $y \Rightarrow x$ in the residual digraph R_M, while each idle edge (x, y) becomes a forward-directed link $x \Rightarrow y$. That is, idle arcs become left-to-right links in R_M, while saturated arcs become right-to-left links. Figure 3.2 shows an example. Note that the residual digraph R_M depends only on the matching M, not on the prices p.

Prop 3-1. *In the residual digraph R_M corresponding to any matching M in the graph G, the in-degree of any married woman is* 1*, while the in-degree of any maiden is* 0*; symmetrically, the out-degree of any married man is* 1*, while the out-degree of any bachelor is* 0*.*

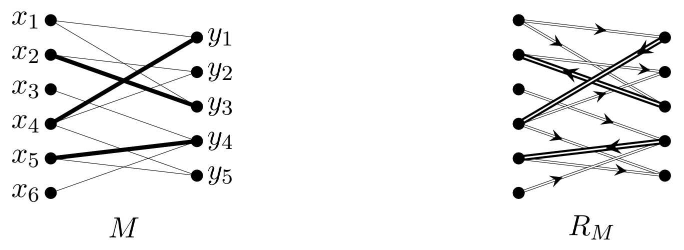

Figure 3.2: A matching M of size 3 and its residual digraph R_M

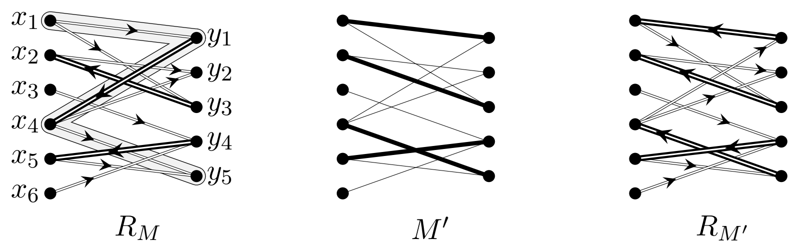

Figure 3.3: An augmenting path of length 3 from x_1 to y_5 in R_M, the new matching M' that results from augmenting along that path, and the residual digraph $R_{M'}$. (By the way, there are two augmenting paths of length nine in $R_{M'}$. Augmenting along either of them would marry off the final bachelor y_2, bringing the matching up to one-sided perfection and leaving either x_3 or x_6 as the sole maiden.)

Proof. Any link in R_M that arrives at a woman or leaves a man must be backward, arising from an edge in the matching M. There is precisely one such edge arriving at each married woman and leaving each married man; but no such edges arrive at any maiden or leave any bachelor. □

An *alternating path* is a path in the residual digraph R_M. Note that the links along an alternating path must alternate between forward and backward. An *augmenting path* is an alternating path that starts at a maiden and ends at a bachelor. So the first and last links of an augmenting path are both forward links. An augmenting path is *tight* when all of the edges that underlie its links are tight, that is, have zero net cost.

We augment along a tight augmenting path by swapping the status of the edges that underlie its links, saturating the idle edges and idling the saturated ones. This process increases the size of the matching by exactly 1. It marries off the maiden and bachelor at the ends of the augmenting path, thus reducing the number of places where future augmenting paths can start or end. Figure 3.3 shows an example. Note that every vertex that was married before the augmentation remains married after it, although the married vertices along the augmenting path, which are y_1 and x_4 in Figure 3.3, end up with new spouses. Note also that our invariants about the net costs of edges allow a tight edge to be either idle or saturated. Hence, augmenting along a tight augmenting path preserves those invariants.

3.3 Defining length in the residual digraph

How does the Hungarian Method find a tight augmenting path along which to augment? We define a notion of distance on the residual digraph, a notion in which every link has nonnegative length and the links of length zero are precisely those whose underlying edges are tight. Given this notion of distance, we use a variant of Dijkstra's algorithm to find an augmenting path that is as short as possible. We then raise prices so as to tighten all of the idle edges along that path, the saturated edges being already tight.

Let R_M be the residual digraph and let p be the prices that are in effect at some point during the Hungarian Method. If $x \Rightarrow y$ is a forward link in R_M, corresponding to an idle edge (x, y), we define the *length* of that link to be the net cost of the edge: $l_p(x \Rightarrow y) := c_p(x, y)$. Recall that this net cost is nonnegative. We define the *length* of any backward link $y \Rightarrow x$ to be zero: $l_p(y \Rightarrow x) := 0$. And we define the *length* of an alternating path to be the sum of the lengths of its links.

Of course, people often define the "length" of a path in a directed graph to be the number of links along that path, rather than the sum of the lengths of those links. In this report, when we are counting links, we will talk about the *link-count* of a path, rather than its *length*.

In the Hungarian Method, saturated edges are kept tight, so defining $l_p(y \Rightarrow x) := 0$ for a backward link $y \Rightarrow x$ has the same effect as defining $l_p(y \Rightarrow x) := c_p(x, y)$ or $l_p(y \Rightarrow x) := -c_p(x, y)$. For future reference, the last of these three is the choice that we will later adopt. In *FlowAssign*, saturated edges will be kept proper, but not necessarily tight. So, for a saturated edge (x, y), we will know only that $c_p(x, y) \leq 0$, and we will need to define $l_p(y \Rightarrow x) := -c_p(x, y)$, to keep our link lengths nonnegative. That is, for an alternating path A, we will define

$$l_p(A) := \sum_{x \Rightarrow y \text{ on } A} c_p(x, y) \; - \sum_{y \Rightarrow x \text{ on } A} c_p(x, y). \tag{3-2}$$

The clumsy subtraction in this definition is a price that we pay, in this report, for setting up our flow network N_G to have only forward arcs, not backward arcs. Recall that many authors define $c_p(y, x) = -c_p(x, y)$; but we leave $c_p(y, x)$ undefined.

By the way, we could model an alternating path A as a flux f_A on the network N_G by assigning flow $f_A(x, y) := 1$ for each forward link $x \Rightarrow y$ along A and $f_A(x, y) := -1$ for each backward link $y \Rightarrow x$. Equation (3-2) would then become a special case of our definition, in (2-5), of the net cost of a flux, with the clumsy -1 hidden in the definition of the flux f_A.

3.4 Building the shortest-path forest

The Hungarian Method chooses an augmenting path of minimal length as its next path along which to augment. We find that shortest path using a variant of Dijkstra's algorithm, but with two differences, as explained in Fredman and Tarjan [10].

First, Dijkstra's algorithm is often used to build a shortest-path tree, rooted at some single node. Instead, we build a shortest-path forest in the residual digraph R_M, with each remaining maiden as the root of a tree in that forest. Thus, the distance $\ell(v)$ that we will compute, for each vertex v in G, is the minimum length of an alternating path in R_M from some maiden to v. We initially set $\ell(\mu) := 0$ for each maiden μ, while we set $\ell(v) := \infty$ for all other vertices v. We lower the distance $\ell(v)$ to some finite value when we find an alternating path from some maiden to v.

Second, we can optimize our processing slightly by treating the vertices in X and in Y, the women and the men, differently. Prop 3-1 tells us that the in-degree in R_M of a married woman is 1. So any alternating path in R_M from a maiden to a married woman x must arrive at x by traversing the backward link $y \Rightarrow x$, where y is the husband of x. Thus, a shortest path to x must consist of a shortest path to y, with the link $y \Rightarrow x$ tacked on at the end. So it suffices to search for shortest paths to men, the shortest paths to their wives falling out with no extra work. Indeed, we have $l_p(y \Rightarrow x) = 0$, since backward edges are kept tight; so the shortest path to a married woman has the same length as the shortest path to her husband (though the link-count of the path to the woman is one greater).

Figure 3.4 sketches the code for building a shortest-path forest in the Hungarian Method. As Fredman and Tarjan suggest [10], we use a Fibonacci heap to store those men who have not yet joined the forest, but who have been reached by some alternating path from some maiden. The key of a man in the heap is the length of the shortest such path found so far.

Let x be some woman who is reachable from a maiden along an alternating path of length $\ell(x)$, and suppose that we know that no shorter path to x will be found. As part of adding x to the forest, we must *scan* her by considering each incident edge (x, y) that is idle — that is, each forward link $x \Rightarrow y$ in the residual digraph R_M. Using the link $x \Rightarrow y$, we have found an alternating path of length $L := \ell(x) + l_p(x \Rightarrow y) = \ell(x) + c_p(x, y)$ from some maiden to y. If $\ell(y) = \infty$, this is the first path that we've found reaching y. We set $\ell(y) := L$ and we insert y into the heap with key L. If $\ell(y)$ is finite, but $L < \ell(y)$, our new path is shorter than the former record-holder, so we set $\ell(y) := L$. The vertex y will be in the heap at this point, so we also decrease the key of y in the heap to the new, smaller value of $\ell(y)$. On the other hand, if $L \geq \ell(y)$, then our new path doesn't break any records, so we ignore it. (In this case, the man y might still be in the heap or he might have already moved from the heap to the forest.)

We begin by setting $\ell(\mu) := 0$ for every maiden μ and scanning each maiden in turn. We then enter the forest-building loop. We do a *delete-min* operation on the heap, thereby finding some reachable man y whose distance from a maiden is minimal, among all men not yet in the forest. We add y to the forest, confident that the distance $\ell(y)$ won't decrease any further. If y is married, we exploit the backward link $y \Rightarrow x$ that connects y to his wife

```
BuildForest()
    make-heap();
    for all nodes $v$, set $\ell(v) := \infty$;
    for all maidens $\mu$, set $\ell(\mu) := 0$ and ScanAndAdd($\mu$);
    while heap is nonempty do
        $y :=$ delete-min();
        add $y$ to the forest;
        if $y$ is married then
            $x :=$ wife of $y$;
            set $\ell(x) := \ell(y)$ and ScanAndAdd($x$);
        else
            exit(bachelor $\beta := y$ reached);
        fi;
    od;
    exit(no bachelor reached);

ScanAndAdd(x)
    for all forward links $x \Rightarrow y$ in $R_M$ do
        $L := \ell(x) + l_p(x \Rightarrow y)$; $L_{\text{old}} := \ell(y)$;
        if $L < L_{\text{old}}$ then
            set $\ell(y) := L$;
            if $L_{\text{old}} = \infty$ then insert($y, L$) else decrease-key($y, L$) fi;
        fi;
    od;
    add $x$ to the forest;
```

Figure 3.4: Building a shortest-path forest in the Hungarian Method

x by setting $\ell(x) := \ell(y)$, scanning x, and adding her to the forest; we then return to the top of the forest-building loop. Otherwise, y is a bachelor, so we have found an augmenting path of minimum length and we stop.

It might happen that, when we return to the top of the forest-building loop, ready to do a *delete-min* operation on the heap, the heap is actually empty. In that case, we have proven that no augmenting paths exist, so we stop without reaching a bachelor.

3.5 The time for building the forest

How much time does it take us to build this shortest-path forest? Recall that, in a Fibonacci heap, a *delete-min* operation costs $O(\log(\text{heap size}))$, while the other heap operations are $O(1)$.

We do one *make-heap* operation. We do at most r *insert* operations, since each man in Y is inserted at most once and we have arranged that Y is the smaller part of the bipartite graph G, with $|Y| = r$. So r is a bound on the size of the heap. We do at most m *decrease-key* operations, since each such operation is sparked by processing some edge — in fact, an idle edge. Finally, we do at most s *delete-min* operations. Each of these operations except perhaps the last deletes a man who turns out to be married; and there are only s edges in the current matching, and hence only s married men. If we succeed in finding an augmenting path, then the final *delete-min* that we perform deletes a bachelor. But augmenting along the resulting augmenting path increases s by 1, so it is still true that, when the smoke has cleared, we have done at most s *delete-min* operations.

Thus, the overall cost for building the shortest-path forest that results in a matching of size s is $O(m + s \log r)$. The remaining operations in the main loop of the Hungarian method are all $O(m)$. The total running time, on the way to a matching of any final size s, is thus

$$O((m + \log r) + (m + 2\log r) + \cdots + (m + s\log r)) = O(ms + s^2 \log r).$$

A practical remark: We started out by flipping G around, if necessary, to make Y the smaller part, with $|Y| \leq |X|$. This reduced the worst-case time bound by making $\log |Y| = \log r$, rather than $\log |Y| = \log n$. In practice, though, it might be faster to flip G the other way, so that there are more men than women. There will then be fewer maidens, so the shortest-path forests may then be smaller — perhaps enough smaller to more than pay for doing heap operations on a potentially larger heap. Experiments will be needed to determine which strategy performs better in practice.

3.6 Raising prices to tighten the path

If an iteration of the Hungarian Method starts with the matching M already of size $s = \nu(G)$, then no augmenting paths can exist, so the building of the shortest-path forest doesn't stop until we are ready to do a *delete-min* operation on an empty heap. In that case, we announce that $s = \nu(G)$ and halt. In every other case, the building of the forest stops when we first add a bachelor to the forest. Let β denote the bachelor that we found, and let μ denote the maiden at the root of the shortest-path tree that β joins. Note that $\ell(\beta)$ is the length of the tree path A from μ to β, that path being an augmenting path of minimal length. Our next task is to raise prices on the nodes in the forest so as to make all of the edges on the path A tight.

Using p' to refer to the prices after we raise them, we reset the (dispose) price at each vertex v in the shortest-path forest by setting

$$p'_d(v) := p_d(v) + \ell(\beta) - \ell(v). \tag{3-3}$$

This change in price is nonnegative, that is, $\ell(\beta) - \ell(v) \geq 0$, because vertices are added to the shortest-path forest in nondecreasing order of their ℓ values. We have three things to check.

Consider first a saturated edge (x, y), that is, an edge in the current matching. It was tight before we raised our prices, and we must show that it remains tight afterward. Is the woman x in the forest? Note that x is married. The only way that a married woman ever enters the forest is as a side effect of her husband's entering the forest. So, if x is in the forest, then y is also in the forest, and we have $\ell(x) = \ell(y)$. Thus, we raise the prices at x and at y by the same amount, so the edge (x, y) remains tight. On the other hand, if x is not in the forest, then y can't be in the forest either, since x is added to the forest as part of adding y; so then we don't change the prices at either x or y.

Next, consider an idle edge (x, y). It had nonnegative net cost $c_p(x, y) = c(x, y) - p_d(x) + p_d(y) \geq 0$ before we raised our prices, and we must show that its net cost remains nonnegative afterward: $c_{p'}(x, y) = c(x, y) - p'_d(x) + p'_d(y) \geq 0$. If the price at x doesn't rise, a rise in the price at y can only help. So we can assume that the price at x does rise, which means that x belongs to the forest. When x entered the forest and was scanned, we considered the idle edge (x, y), and we lowered $\ell(y)$, if necessary, to ensure that $\ell(y) \leq \ell(x) + c_p(x, y)$. In subsequent processing, $\ell(y)$ may have been lowered even further; but $\ell(x)$ didn't change. So, when we finished building the forest, we had

$$\ell(y) \leq \ell(x) + c_p(x, y) = \ell(x) + c(x, y) - p_d(x) + p_d(y). \tag{3-4}$$

Since we are assuming that x did enter the forest, we have $p'_d(x) = p_d(x) + \ell(\beta) - \ell(x)$, so we have

$$\ell(y) \leq c(x, y) - p'_d(x) + p_d(y) + \ell(\beta). \tag{3-5}$$

Now, the vertex y may or may not have entered the forest. If y did enter the forest, we have $p'_d(y) = p_d(y) + \ell(\beta) - \ell(y)$. Substituting into (3-5), we find that $0 \leq c(x, y) - p'_d(x) + p'_d(y) = c_{p'}(x, y)$, which is our goal. If y did not enter the forest, we have $p'_d(y) = p_d(y)$, and substituting into (3-5) tells us $\ell(y) \le c_{p'}(x,y) + \ell(\beta)$. But we can also conclude that $\ell(\beta) \le \ell(y)$, since β was selected by doing a *delete-min* operation on a heap that included y. So we again have our goal.

One more thing to check: After we raise prices, we want all of the edges on the alternating path A from μ to β to be tight, so that we can augment along A. We have already checked that all saturated edges continue to have zero net cost after the repricing, including the saturated edges on A. It remains to consider one of the idle edges on A, say the forward link $x \Rightarrow y$. Both x and y belong to the forest, so we have $p'_d(x) = p_d(x) + \ell(\beta) - \ell(x)$ and $p'_d(y) = p_d(y) + \ell(\beta) - \ell(y)$. Furthermore, x is the predecessor of y on a shortest path from a maiden; so we have $\ell(y) = \ell(x) + c_p(x,y)$, which means that inequality (3-4) holds with equality. Substituting in, we find that $c_{p'}(x,y) = 0$, as we hoped.

3.7 Verifying optimality

We have shown that, for each s in turn, the Hungarian Method computes a matching M of size s in time $O(ms + s^2 \log r)$. And the invariants of the Hungarian Method ensure that all idle bipartite arcs will have nonnegative net cost and all saturated bipartite arcs will have nonpositive net cost — in fact, zero net cost. If the maiden and bachelor bounds hold, then we will be able to choose prices at the source and the sink that make all of the dummy arcs proper, thus ensuring that the matching M will be min-cost of its size s, by Corollary 2-10. We here establish those bounds, pulling out some parts of the argument as propositions for future reference.

Prop 3-6. *Let M be a matching computed by the Hungarian Method, let x_0 be some woman who is married in M, and let μ_0 be some woman who is still a maiden. During the running of the Hungarian Method so far, the price at μ_0 has increased by at least as much as the price at x_0.*

Proof. Each time that we build a shortest-path forest, we include all of the remaining maidens as tree roots, and each such maiden μ has $\ell(\mu) = 0$. When prices are raised based on this forest, we set $p'_d(\mu) := p_d(\mu) + \ell(\beta) - 0$ for each such μ. The prices at all remaining maidens are thus increased by the same amount, by $\ell(\beta)$, and that amount is the maximum price increase. It follows that the overall price increase so far at any remaining maiden μ_0 is at least as great as the corresponding increase at any other vertex, including at any married woman x_0. $\square$

In the Hungarian Method, every woman x starts out at the same price $p_d(x) := 0$. It follows from Prop 3-6 that the remaining maidens are always the most expensive women, which establishes the maiden bound (2-12).

Prop 3-7. *Let M be a matching computed by the Hungarian Method, let y_0 be some man who is married in M, and let β_0 be some man who is still a bachelor. During the running of the Hungarian Method so far, the price at β_0 has not increased at all, while the price at y_0 may have increased.*

Proof. As the Hungarian Method runs, the prices at some men are raised, because those men belong to a shortest-path forest. But the price at a bachelor is never raised. Indeed, each shortest-path forest that causes some prices to rise contains just one bachelor, the vertex β whose entry into that forest stopped its construction. And the price at β is raised by the formula $p'_d(\beta) := p_d(\beta) + \ell(\beta) - \ell(\beta)$, and hence is left unchanged. Thus, every man retains his original price for as long as that man remains a bachelor. $\square$

In the Hungarian Method, every man y starts out at the same price $p_d(y) := \bar{C}$. It follows from Prop 3-7 that the remaining bachelors are always the least expensive men, which establishes the bachelor bound (2-14). This completes our performance analysis of the Hungarian Method:

Prop 3-8. *The Hungarian Method computes min-cost matchings incrementally, thus solving **IncA** in space $O(m)$ and time $O(ms + s^2 \log r)$.*

The Hungarian Method generalizes neatly to the unbalanced case because it naturally preserves the maiden and bachelor bounds. But be warned that some people, when using the Hungarian Method to compute perfect matchings, find their augmenting paths by building a shortest-path tree starting from some single maiden, rather than a shortest-path forest starting from all remaining maidens. If a perfect matching exists, then we are guaranteed to find some augmenting path, even if we start looking from just one arbitrarily chosen maiden. But the resulting variant of the Hungarian Method does not preserve the maiden bound, and hence any matching that it constructs that leaves some women as maidens may fail to be min-cost.

Be warned as well that some people add a preprocessing phase to the Hungarian Method, a phase that consists of local updates to the prices. After setting the prices at all women to 0 and at all men to $\bar{C}$, they consider each vertex in turn, in some order. They raise the price at each woman as far as they can and they lower the price at each man as far as they can, while keeping the net costs of all bipartite arcs nonnegative. (All bipartite arcs are idle at this point, so their net costs must remain nonnegative to keep them proper.) These local updates can help the subsequent processing to go faster; but the resulting variant of the Hungarian Method doesn't preserve either the maiden bound or the bachelor bound.

3.8 Bounding the prices

Finally, let's verify that the prices that it computes don't get too big. This is interesting in its own right, and some bound of this type is also needed to show that the Hungarian Method is indeed strongly polynomial.

Prop 3-9. *For any $s \geq 1$, let $2l - 1$ be the link-count of the path A along which the Hungarian Method augments to bring its matching up to size s. So $1 \leq l \leq s$. The (dispose) prices that the Hungarian Method uses to demonstrate the optimality of its matching of size s are nonnegative real numbers bounded by $(2l + 1)\bar{C}$.*

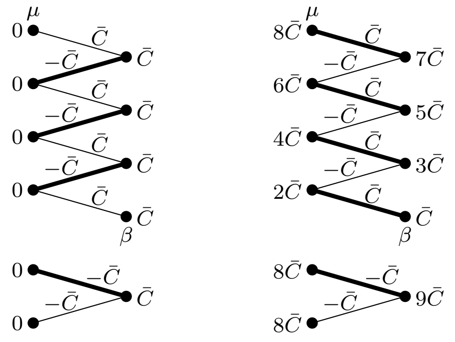

Figure 3.5: An example in which the Hungarian Method generates large prices; nodes are labeled with prices, edges with costs

Proof. All of the prices are initialized to nonnegative values, and they change only when they are increased; so nonnegativity is clear.

Let A be the path of length $2l-1$ along which we augment to bring our matching up to size s. We found the path A by building a shortest-path forest. Let β be the bachelor whose discovery stopped the construction of that forest, and let μ be the maiden at the root of the shortest-path tree that β joins. So A is an alternating path in R_M from μ to β, and augmenting along A results in marriages for both μ and β. Let p denote the prices after we raise them based on this forest, so the prices p are in effect when we augment along A. We intend to show that $p_d(x) \leq 2l\bar{C}$, for all women x, and that $p_d(y) \leq (2l+1)\bar{C}$, for all men y.

The price increases in the round that we are studying brought the net cost of every arc on the path A to zero. So the prices at the two endpoints of such an arc can't differ by more than $\bar{C}$. It follows that $p_d(\mu) \leq p_d(\beta) + (2l-1)\bar{C}$. But we have $p_d(\beta) = \bar{C}$, since the price at any bachelor remains at $\bar{C}$ until some round of price increases that happens after he is already married. So we have $p_d(\mu) \leq 2l\bar{C}$.

In each round of price increases, the prices at all of the remaining maidens increase by the same amount, and that amount is at least as large as any other price increase during that round. Since μ has been a maiden in every round of price increases so far and since the prices at all women are initialized to the same value — in fact, to 0 — we have $p_d(x) \leq p_d(\mu) \leq 2l\bar{C}$, for all women x. The same argument applies to the men, with the only difference being that they are initialized to $\bar{C}$, rather than to 0; so we have $p_d(y) \leq p_d(\mu) + \bar{C} \leq (2l+1)\bar{C}$, for all men y. □

Figure 3.5 shows an example in which these bounds are tight. The graph G consists of a path of length 7 (that is, of edge-count 7), together with a separate path of length 2. The costs along the path of length 7 alternate between $\bar{C}$ and $-\bar{C}$, with the first and last edges being cost $\bar{C}$. The costs of the two edges on the other path are both $-\bar{C}$. The Hungarian Method builds a matching of size 4 in this graph without any price increases, as shown on the left in Figure 3.5, with the matching edges drawn bold. (Some accident of ordering determines which of the two edges on the path of length 2 participates in this matching.) There are two maidens that remain, acting as roots in the next shortest-path forest. The last vertex to join that forest is the only remaining bachelor, labeled β, and it joins the tree rooted at the maiden labeled μ. The augmenting path A thus discovered is the length-7 portion of G, and we have $\ell(\beta) = 8\bar{C}$. We then raise prices and augment along A, leading to the state shown on the right in Figure 3.5.

Chapter 4

Weight-scaling

The Hungarian Method takes time $O(ms+s^2 \log r)$; but something more like $O(m\sqrt{s})$ would be better. The known algorithms that achieve this type of $\sqrt{s}$ performance use some version of weight-scaling. And the key to weight-scaling is defining what it means for arcs to be “approximately proper”.

4.1 Approximately proper

Recall that idle arcs are proper when their net costs are nonnegative, while saturated arcs are proper when their net costs are nonpositive. To make this concept approximate, one introduces a real parameter $\varepsilon > 0$. Roughly speaking, an arc is *ε-proper* when its net cost is within ε of being proper. But what happens in the boundary cases, when the net cost is precisely ε away from making the arc proper? In *FlowAssign*, we are going to resolve those boundary cases in a somewhat subtle, one-sided way, as discussed in Section 6.1. But we don't have to worry about that subtlety here.

For now, we say that an idle arc $v \to w$ is *ε-proper* when $c_p(v,w) > -\varepsilon$ and that it isn't *ε-proper* when $c_p(v,w) < -\varepsilon$; and we don't specify which happens in the boundary case, where $c_p(v,w) = -\varepsilon$. In a similar way, we say that a saturated arc $v \to w$ is *ε-proper* when $c_p(v,w) < \varepsilon$ and that it isn't *ε-proper* when $c_p(v,w) > \varepsilon$; and we don't specify which happens in the boundary case, where $c_p(v,w) = \varepsilon$.

To see how this approximate notion of properness could be helpful, note that ε-properness is easy to achieve when ε is large enough. In particular, when $\varepsilon > \bar{C}$, we can set all prices to zero and make all arcs ε-proper. On the other hand, when ε is small enough, then ε-properness can, by itself, suffice to demonstrate that a matching M is min-cost. If all costs are integers, for example, it turns out that $\varepsilon < 1/6s$ suffices, where $s := |M|$.

Prop 4-1. *Let G be a bipartite graph whose edge weights are all integers, let f be an integral flow on the flow network N_G of value $s := |f|$, and suppose that we can find prices p at the nodes of N_G that make all arcs of N_G be ε-proper, for some $\varepsilon < 1/6s$. The flow f is then min-cost, among all integral flows of value s.*

Proof. We begin, as in the proof of Prop 2-8, letting f' be any flow whose value is also $s = |f'|$ and considering the net cost of the difference flux $f' - f$:

$$c_p(f' - f) = \sum_{v \to w \, \in \, N_G} (f' - f)(v, w) c_p(v, w).$$

In Prop 2-8, we argued that $c_p(f' - f) \geq 0$ because each term in the sum was nonnegative. But we are now assuming, not that all arcs are proper, but only that they are ε-proper. On the other hand, we are here assuming that both of the flows f and f' are integral. Since the arcs $v \to w$ with $f(v, w) = f'(v, w)$ drop out of the sum, we thus have

$$c_p(f' - f) = \sum_{\substack{f(v,w)=0 \\ f'(v,w)=1}} c_p(v, w) - \sum_{\substack{f(v,w)=1 \\ f'(v,w)=0}} c_p(v, w).$$

In any integral flow of value s, there are exactly $3s$ arcs that have unit flow: s left-dummy arcs, s bipartite arcs, and s right-dummy arcs. Hence, there are at most $3s$ terms in each of these two sums. Furthermore, any arc $v \to w$ with $f(v, w) = 0$ must have $c_p(v, w) \geq -\varepsilon$ to be ε-proper; and any arc with $f(v, w) = 1$ must have $c_p(v, w) \leq \varepsilon$. Thus, while we can't conclude that $c_p(f' - f) \geq 0$, we do have $c_p(f' - f) \geq -6s\varepsilon > -1$.

Considering the costs and net costs of the flows f' and f separately, as in Prop 2-8, we have $c_p(f' - f) = c(f') - c(f)$, so $c(f') > c(f) - 1$. But both $c(f')$ and $c(f)$ must be integral, since all edge weights are integers, so we deduce that $c(f') \geq c(f)$. □

Note: In papers that focus on **PerA** and thus compute only perfect matchings, the analogs of this proposition get by with the weaker assumption that $\varepsilon < 1/2n$, rather than our $\varepsilon < 1/6s$. If f and f' are flows of value n in the flow network N_G of a balanced graph G, then both f and f' must saturate all n of the left-dummy arcs and all n of the right-dummy arcs. The argument above can thus consider only the bipartite arcs, of which there are most n in each sum, leading to the bound $c_p(f' - f) \geq -2n\varepsilon$.

4.2 The weight-scaling technique

Here is the rough structure of a typical weight-scaling algorithm. It computes flows and prices that are ε-proper, for various values of ε. It starts with $\varepsilon := C$, or thereabouts, to make the initialization straightforward. It then carries out a sequence of *scaling phases*, each of which divides ε by some factor $q > 1$, typically taken to be a constant. Let s denote the size of the output matching, the matching that we must show to be min-cost. The algorithm reduces ε from about C to about $1/s$, perhaps to $1/6s$, by carrying out about $\log_q(sC)$ scaling phases. At this point, some result like Prop 4-1 is invoked, to show that the output matching is indeed min-cost.

A scaling phase thus takes, as input, a flow of value s and prices that are $(q\varepsilon)$-proper. It produces, as output, a new flow of value s and new prices that are ε-proper. Thus, it improves the accuracy of the approximation to properness by a factor of q. A typical weight-scaling algorithm manages somehow to implement a scaling phase to run in time $O(m\sqrt{s})$, leading to an overall running time of $O(m\sqrt{s}\log(sC))$.

Various weight-scaling assignment algorithms have been published. They typically solve only **PerA**, and hence compute only perfect matchings. We now discuss three of them, commenting on how easy it might be to generalize them to compute imperfect matchings that are still min-cost.

4.3 Gabow-Tarjan and perfect matchings

We first consider the Gabow-Tarjan algorithm [11]. Like the Hungarian Method, Gabow-Tarjan works directly on the graph G, not on some flow network constructed from G, and it verifies that its matchings are min-cost by coming up with prices that make all bipartite arcs proper. That suffices, because Gabow-Tarjan computes only perfect matchings — so there are no maidens or bachelors to worry about.

Gabow-Tarjan has a straightforward weight-scaling structure. Each scaling phase builds a new matching from scratch, exploiting only the prices computed by the previous phase. The new matchings are computed using augmenting paths, very much as in the Hungarian Method. Indeed, Gabow-Tarjan builds shortest-path forests and does rounds of price increases, just like the Hungarian Method. But some things must be different, of course, since each scaling phase of Gabow-Tarjan has to run in time $O(m\sqrt{s})$, while the Hungarian Method takes time $O(ms + s^2\log r)$.

The biggest challenge is to reduce the ms term to $m\sqrt{s}$. Gabow-Tarjan does this using ideas like those used in the Hopcroft-Karp algorithm for the matching problem. We discuss how Hopcroft-Karp achieves its $\sqrt{s}$ performance in Section 5.2. Gabow-Tarjan and our new algorithm *FlowAssign* achieve their $\sqrt{s}$ performance in very similar ways, and we discuss how *FlowAssign* does so in Section 9.4.

Avoiding the factor of $\log r$ is a simpler matter. In the scaling phase that ends with prices that are ε-proper, Gabow and Tarjan constrain all prices to be multiples of ε, and they round the weights of all edges to be multiples of ε as well. They thus arrange that the keys associated with the nodes in their heap are all small integers. As we discuss in Section 8.1, this means that they can avoid any logarithmic heap overhead by using the Dial technique [7].

4.4 Gabow-Tarjan and imperfect matchings

Does Gabow-Tarjan generalize to compute imperfect matchings that are still min-cost? On the positive side, each scaling phase of Gabow-Tarjan is much like the Hungarian Method, and that method preserves the maiden and bachelor bounds. But it turns out that the maiden and bachelor bounds are lost in going from one scaling phase to the next.

The Hungarian Method preserves the maiden bound (2-12) because the prices at all remaining maidens increase by at least as much as the prices at any woman, as we showed in Prop 3-6. That property holds true, also, for each scaling phase of Gabow-Tarjan. In the Hungarian Method, all women start out at the same price, so the maidens end up as the most expensive women. Unfortunately, in the scaling phases of Gabow-Tarjan, the women start out at whatever prices were computed by the preceding phase. If all phases ended up choosing the same set of women to be their final maidens, we'd still be okay. The prices at those perpetual maidens would both start and end each phase at least as high as the prices at the other women. But each scaling phase in Gabow-Tarjan makes its own, independent decision about which women will be the final maidens of that phase. So Gabow-Tarjan does not preserve the maiden bound, and we cannot trust it to compute imperfect matchings that are min-cost.

The story for the bachelor bound (2-14) is similar. The Hungarian Method doesn't raise the price at any man until after that man is no longer a bachelor, as we showed in Prop 3-7. The same holds for each scaling phase of Gabow-Tarjan. In the Hungarian Method, all men start out at the same price, so the bachelors end up as the cheapest men. In Gabow-Tarjan, however, the men start each phase at whatever prices were computed by the preceding phase. If all phases chose the same men to end up as bachelors, we'd still be okay. But the phases make independent decisions about which men will end up as bachelors, so the bachelor bound is not preserved.

Our new algorithm is quite similar to Gabow-Tarjan; but, in *FlowAssign*, we work on the flow network N_G, rather than on G itself. In particular, we keep track of prices at the source and the sink throughout the algorithm, and we search for a flow and prices that make the dummy arcs proper, as well as the bipartite arcs. So we have no need to appeal to the maiden and bachelor bounds, after the fact, to establish that our matchings are min-cost.

FlowAssign also differs from Gabow-Tarjan in that more information is carried over from one scaling phase to the next. In both algorithms, each new phase starts off using the prices computed by the preceding phase. In *FlowAssign*, however, each new phase also remembers which vertices ended up matched versus unmatched, at the end of the preceding phase. The details of how the preceding phase paired up married women to married men are forgotten, as in Gabow-Tarjan. But the married-versus-unmarried status of each vertex is remembered. The new phase may choose to marry off different vertices; but any such choice is guided by the current prices and requires explicit bookkeeping.

4.5 Orlin-Ahuja

The Orlin-Ahuja algorithm [17] also has a straightforward weight-scaling structure, but the scaling phases in Orlin-Ahuja build their new matchings using a combination of local and global techniques. Each phase starts out working locally, building most of its new matching using auctions. But it finishes up by working globally, bringing its new matching up to perfection using augmenting paths. This hybrid structure may lead to performance that is good both in theory and in practice.

Orlin-Ahuja works directly on the graph G and computes prices that are make all bipartite arcs proper. As published, it computes only perfect matchings in balanced graphs, so there are neither maidens nor bachelors to worry about. If we tried to generalize it to compute imperfect matchings, we would have trouble for several reasons. First, like Gabow-Tarjan, Orlin-Ahuja carries over prices from one scaling phase to the next, and each phase makes independent decisions about which vertices end up matched. So it fails to preserve the maiden and bachelor bounds for the same reason that Gabow-Tarjan does. But Orlin-Ahuja has another problem as well. The auction-like updates at the start of each scaling phase make no attempt to preserve the maiden and bachelor bounds; so those bounds are lost immediately.

There are fancier versions of auction-like updates that are careful to preserve the maiden and bachelor bounds. For example, Bertsekas and Castañon [3] compute one-sided-perfect matchings in graphs with a majority of men, so their matchings leave some bachelors. Their matchings are min-cost despite these bachelors because they introduce an auction step that maintains a scalar λ that they call a *profitability threshold*. In our framework, λ serves as a fence that separates the prices at the bachelors from the prices at the married men, thus ensuring that the bachelor bound is preserved — and, in fact, we can think of λ as a proposed value for the price at the sink. Auction steps of these fancier types might be useful in designing an algorithm for **ImpA** that combines local and global techniques so as to perform well in both theory and practice, as we discussed in Section 1.4.

4.6 Goldberg-Kennedy

The Goldberg-Kennedy algorithm [13] uses weight-scaling, not to compute a min-cost matching in the given bipartite graph G directly, but instead to compute a min-cost circulation in a certain flow network.[*] As a result, their algorithm generalizes to the unbalanced case easily from a correctness point of view; but the performance of the result is unclear.

Goldberg and Kennedy first reduce **PerA** to the problem of finding a min-cost circulation in a flow network. Their network is quite close to our N_G, but with loop-back arcs added from the sink to the source, thereby enabling their network to support circulations. Because they use a network in this way, it should be straightforward to adjust their algorithm to compute imperfect matchings that are min-cost. That's the good news.

Having constructed their flow network, Goldberg and Kennedy then find a min-cost circulation in that network using a local, push-relabel algorithm,

[*]A *circulation* is like a flow, except that flow is conserved at all nodes; so the network has no source or sink.

supplemented by a subtle machinery of occasional global price updates. They show that their algorithm achieves the standard weight-scaling time bound of $O(m\sqrt{n}\log(nC))$. But their arguments are complex and are phrased entirely in terms of n, with r and s playing no role. To make Goldberg-Kennedy interesting from our perspective, we would have to replace some of the n's in this time bound with r's or s's. Doing so is an intriguing open problem.

Chapter 5

Hopcroft-Karp

The matching algorithm of Hopcroft and Karp [14] was the first algorithm in this area to achieve $\sqrt{s}$ performance. We now analyze Hopcroft-Karp because our new algorithm *FlowAssign* follows Gabow-Tarjan by achieving its $\sqrt{s}$ performance in a very similar way.

Keep in mind that Hopcroft-Karp solves matching problems, rather than assignment problems. So the bipartite graphs in this chapter don't have weights on their edges, and we have no costs, prices, or net costs to deal with. Also, in this chapter, we treat all edges as unit length, so the length of a path is the same as its link-count.

Figure 5.1 sketches the code of Hopcroft-Karp. As published, Hopcroft-Karp finds a matching of size $\nu(G)$ in a bipartite graph G in space $O(m)$ and time $O(m\sqrt{\nu(G)})$. We are going to call Hopcroft-Karp as part of the initialization of *FlowAssign*, and $O(m\sqrt{\nu(G)})$ is more time than we are going to be able to afford. Fortunately, Hopcroft-Karp builds its matching incrementally, and we will show that, if we stop it when the matching has reached size s, the time spent is $O(m\sqrt{s})$, which we can afford. So Hopcroft-Karp solves both **ImpM** and **IncM** in time $O(m\sqrt{s})$.

```
HopcroftKarp($G$)
    set $M$ to the empty matching;
    do
        find a maximal set $\boldsymbol{\mathcal{P}}$ of vertex-disjoint augmenting paths,
                for $M$, all of the minimum possible length;
        if $|\boldsymbol{\mathcal{P}}| = 0$ then announce($\nu(G) = |M|$); return; fi;
        for $P$ in $\boldsymbol{\mathcal{P}}$ do
            augment $M$ along $P$;
            announce($M$ is a matching);
        od;
    od;
```

Figure 5.1: Pseudocode for Hopcroft-Karp

5.1 The disjoint-path bound

The key to the $\sqrt{s}$ performance of Hopcroft-Karp is an inequality that we will call the *disjoint-path bound*, called that because the proof involves a collection of paths that are vertex-disjoint.

Prop 5-1. *In a bipartite graph G, let M_0 be a matching of size $s_0 := |M_0| \geq 0$, and suppose that every augmenting path for M_0 has length at least L. For any integer s_1 with $s_0 \leq s_1 \leq \nu(G)$, we then have the* disjoint-path bound*:*

$$(s_1 - s_0)(L + 1) \leq 2s_1. \tag{5-2}$$

Proof. By augmenting along augmenting paths, we could transform M_0 into larger and larger matchings, until reaching a matching of size $\nu(G)$. Let M_1 denote the matching of size s_1 that we would reach during this process.

Let $X_{M_1} \subseteq X$ consist of those vertices in X that are matched in M_1, and let Y_{M_1} denote the corresponding subset of Y. So we have $|X_{M_1}| = |Y_{M_1}| = s_1$. Since we can transform M_0 into M_1 by augmenting along augmenting paths, every vertex that is matched in M_0 is matched also in M_1, although the vertex to which it is matched may be different. Thus, each edge in M_0 must connect some vertex in X_{M_1} to some vertex in Y_{M_1}.

Consider the multigraph $M_0 \uplus M_1$ that results from assembling together all of the edges of M_0 and of M_1. If any edge appears both in M_0 and in M_1, we make that edge appear twice in $M_0 \uplus M_1$, and that's why $M_0 \uplus M_1$ may be a multigraph. But note that the multigraph $M_0 \uplus M_1$ contains only $2s_1$ vertices: the vertices in $X_{M_1} \cup Y_{M_1}$.

What are the connected components of $M_0 \uplus M_1$? Of the s_1 vertices in X_{M_1}, the s_0 that are matched in M_0 have degree 2, while the remaining $s_1 - s_0$ have degree 1; and the analogous claim holds for Y_{M_1}. Some of the connected components of $M_0 \uplus M_1$ may be even-length cycles, all of whose vertices have degree 2 and whose edges alternate between M_0 and M_1. For example, if M_0 and M_1 share any edge, the resulting double edge in $M_0 \uplus M_1$ is a cycle of this type of length 2. But any connected component of $M_0 \uplus M_1$ that isn't a cycle must be a path of odd length whose first and last edges belong to M_1. Furthermore, there must be precisely $s_1 - s_0$ such paths, matching up the degree-1 vertices in X_{M_1} with the degree-1 vertices in Y_{M_1}. And each such path is an augmenting path for M_0.

We are assuming that all augmenting paths for M_0 have length at least L, which means that they touch at least $L + 1$ vertices. And we have found $s_1 - s_0$ augmenting paths that fit, in a vertex-disjoint way, into the multigraph $M_0 \uplus M_1$, which has only $2s_1$ vertices. So we have $(s_1 - s_0)(L + 1) \leq 2s_1$. $\square$

By the way, if the matching M_0 was the current matching at some point during some run of Hopcroft-Karp and if M_1 was the output matching of that run, then Hopcroft-Karp might be lucky enough to transform M_0 into M_1 by augmenting along precisely the $s_1 - s_0$ augmenting paths that we constructed above. That would be efficient, in the sense that no edge would change status more than once. Each edge in $M_1 \setminus M_0$ would enter the matching at some point and each edge in $M_0 \setminus M_1$ would leave it, and there would be no other changes of status. In general, however, Hopcroft-Karp will choose paths along which to augment that cause some edges to change status more than once.

Corollary 5-3. *If every augmenting path for a matching M of size s in a bipartite graph G has length at least $L = 2k + 1$, then*

$$s \geq \left(1 - \frac{2}{L+1}\right) \nu(G) = \frac{k}{k+1}\, \nu(G). \tag{5-4}$$

Proof. We apply the disjoint-path bound (5-2) with $s_0 := s$ and $s_1 := \nu(g)$ and then do some easy algebra. $\square$

By the way, for any $k \geq 0$, it's easy to find a matching that makes inequality (5-4) tight: Take the even-numbered edges along a path of length $2k+1$. The whole path is the unique augmenting path for this matching, and we have $s = \frac{k}{k+1}\, \nu(G)$. (Examples like this come up in Appendix B, where we discuss graphs on which Hopcroft-Karp can experience a slow start.)

5.2 Showing square-root performance

The disjoint-path bound (5-2) is the key to the square root in the time bound for Hopcroft-Karp. We won't give the full proof here. In particular, we simply assume two claims, referring to the literature for their proofs [5, 14].

First, we assume that Hopcroft-Karp can be implemented so that each iteration of the outer loop takes time $O(m)$. The challenge in doing this is finding $\boldsymbol{\mathcal{P}}$, the maximal set of vertex-disjoint augmenting paths, all of the minimum possible length. It isn't hard, in $O(m)$ time, to determine the minimum possible length of an augmenting path and to find one augmenting path of that length. But it takes some care to continue to find additional augmenting paths of that same length, until we have built up a maximal set of them, all in time $O(m)$. Note that $\boldsymbol{\mathcal{P}}$ may well fail to be maximum, and that's okay. It suffices that, given the choices of minimum-length augmenting paths that we've already made, there is no additional augmenting path of that same length that is vertex-disjoint from our current paths.

Second, we assume that the minimum length of an augmenting path increases, from each iteration of the outer loop of Hopcroft-Karp to the next. Suppose that the shortest current augmenting paths have length L. If we compute a maximal set of vertex-disjoint augmenting paths of length L and we then augment along all of those paths, any augmenting paths that exist after we are done will have length greater than L. In fact, since lengths of augmenting paths are always odd, they will have length at least $L + 2$.

Assuming those two claims, we now consider the runtime of Hopcroft-Karp in the light of the disjoint-path bound (5-2).

Prop 5-5. *Hopcroft-Karp solves **IncM** in space $O(m)$ and time $O(m\sqrt{s})$.*

Proof. We are assuming that each iteration of the outer loop takes time $O(m)$ and increases the minimum length L of an augmenting path by at least 2. It's clear that each such iteration also increases $|M|$ by at least 1, since each iteration (except perhaps the last) finds at least one augmenting path and augments along it. Thus, each iteration makes progress on two different fronts, increasing both L and $|M|$.

For any $i \geq 1$, consider the state after the algorithm has completed i iterations of its outer loop; suppose that the matching is then M_i, of size s_i. We will then have $L > 2i$, since L increases by at least 2 per iteration. From the disjoint-path bound, we deduce that $s - s_i < s/i$, where s is the size of the output matching. Since $|M|$ increases by at least 1 in each iteration, an additional s/i iterations will finish the job, bringing our matching up from size s_i to size s. Thus, the total number of iterations is at most $i + s/i$. Setting $i := \sqrt{s}$ to balance the two terms in this upper bound, we deduce that $2\sqrt{s}$ iterations suffice, so the overall runtime is $O(m\sqrt{s})$. □

Let $c \leq 1$ be some constant, and suppose that we set $t = c\,\nu(G)$ in **ImpM**. Thus, we are looking for some matching that is within a factor of c of being maximal. Prop 5-5 shows that Hopcroft-Karp will find such a matching in time $O(m\sqrt{t}) = O(m\sqrt{\nu(G)})$. And that is the correct answer when $c = 1$. When $c < 1$, however, more is true.

Prop 5-6. *If the target size t in **ImpM** satisfies $t \leq c\,\nu(G)$ for some constant $c < 1$, then Hopcroft-Karp solves **ImpM** in time $O(m)$.*

Proof. Run the algorithm for $1/(1-c) = O(1)$ iterations of its main loop. At this point, we will have $L > 2/(1-c)$. Invoking Corollary 5-3, we find that $s > c\,\nu(G)$, so we are done in time $O(m)$. □

Chapter 6

Introducing *FlowAssign*

FlowAssign is our new, weight-scaling algorithm for computing imperfect assignments. To carry out a scaling phase, *FlowAssign* invokes a procedure called *Refine*. In this chapter, we discuss the high-level structure of *FlowAssign*, but we don't dive into *Refine* until Chapter 7.

The input to *FlowAssign* is a bipartite graph G with integral edge weights and a target size t. *FlowAssign* computes a min-cost matching in G of size $s := \min(t, \nu(G))$, along with integral prices that demonstrate that its output is indeed min-cost. It runs in space $O(m)$ and in time $O(m\sqrt{s}\log(sC))$, where $C > 1$ is a bound on the magnitude of any edge weight.

We first perform some initialization. Ignoring the edges weights, we use Hopcroft-Karp to look for some matching of size t. Hopcroft-Karp warns us if t exceeds $\nu(G)$ and gives us an initial matching of size $s := \min(t, \nu(G))$. Keep in mind that a matching of size s in the graph G corresponds to an integral flow f of value $|f| = s$ in the flow network N_G, and it is best to think, from here on, of working with such flows.

The core of *FlowAssign* is the procedure *Refine*, which carries out a single scaling phase. This procedure takes, as input, a flow f of value s and prices p that together make all arcs $(q\varepsilon)$-proper, where q is an integer parameter. *Refine* builds a new flow f', also of value s, and prices p' that together make all arcs ε-proper. Thus, it improves the quality with which we are approximating properness by a factor of q. We are going to end up choosing q to be a constant; $q = 8$ or $q = 16$ might be good choices. But we initially treat q as an independent parameter, allowing for such algorithm-design options as setting $q = \Theta(\log n)$.

FlowAssign calls *Refine* about $\log_q(sC)$ times, thus reducing the error parameter ε from about C to about $1/s$. At that point, our approximate prices are so close to being proper that, with a little care, we can simply round them to be proper, thereby proving that the matching produced by the final call to *Refine* is indeed min-cost of size s.

Recall that the Hungarian Method in incremental, solving **IncA** as well as **ImpA**; but be warned that *FlowAssign* is not. The procedure *Refine* does use augmenting paths to build up its approximately min-cost matching of size s, so it produces smaller matchings along the way that are also approximately min-cost. But small matchings that are precisely min-cost might not emerge from any calls to *Refine* until after lots of lousy matchings of the full size s have been computed. So *FlowAssign* is not usefully incremental.

6.1 Ceiling quantization

In *FlowAssign*, when we define what it means for an arc to be ε-proper, we are going to decide the boundary cases in a one-sided manner; and we are going to exploit that one-sidedness when rounding our prices to make them proper, at the end of *FlowAssign*.

Let ε be a positive real number. During the scaling phase in which *FlowAssign* is striving for ε-properness, how we treat an arc $v \to w$ will depend upon that arc's net cost $c_p(v, w)$ only through the quantity $\lceil c_p(v, w)/\varepsilon \rceil$. So what matters is which of the following intervals on the real line contains that arc's net cost:

$$\ldots,\ (-2\varepsilon \mathinner{..} -\varepsilon],\ (-\varepsilon \mathinner{..} 0],\ (0 \mathinner{..} \varepsilon],\ (\varepsilon \mathinner{..} 2\varepsilon],\ (2\varepsilon \mathinner{..} 3\varepsilon],\ \ldots \tag{6-1}$$

We have chosen these intervals to be left-open and right-closed, and we'll discuss why in a moment. We are following Gabow-Tarjan by quantizing our net costs to multiples of ε, but we are adding a new wrinkle by adopting this *ceiling quantization*.

Consider an integral pseudoflow f on the network N_G and prices p at the nodes of N_G. We define an idle arc $v \to w$ in the network N_G to be *ε-proper* when $c_p(v, w) > -\varepsilon$ and a saturated arc $v \to w$ to be *ε-proper* when $c_p(v, w) \le \varepsilon$. Note that a saturated arc $v \to w$ is allowed to have $c_p(v, w) = \varepsilon$ and still be ε-proper, but an idle arc $v \to w$ that wants to be ε-proper is not allowed to have $c_p(v, w) = -\varepsilon$. Those choices are dictated by our ceiling quantization.*

Prop 6-2. *Let $v \to w$ be an idle arc whose net cost is known to be a multiple of ε. If the arc $v \to w$ is ε-proper, then it is automatically proper.*

Proof. We must have $c_p(v, w) > -\varepsilon$, so we actually have $c_p(v, w) \ge 0$. $\square$

We don't get the analogous automatic properness for saturated arcs. But idle arcs are typically in the majority, so we prefer to deal with the idle arcs automatically and deal with the saturated arcs in some other way. That's why we quantize with ceilings, rather than floors.

Recall that an arc with net cost zero is often called *tight*. We say that an idle arc $v \to w$ is *ε-tight* when $-\varepsilon < c_p(v, w) \le 0$, while a saturated arc $v \to w$ is *ε-tight* when $0 < c_p(v, w) \le \varepsilon$. (Gabow and Tarjan refer to these arcs as *eligible*.) Note that ε-tight arcs are ε-proper, but just barely so, in the sense of "just barely" that our ceiling quantization allows. We also

*It follows that an idle arc $v \to w$ with $c_p(v, w) = 0$ is proper, but is not 0-proper; this is potentially confusing, so we will talk about arcs being γ-proper only when $\gamma > 0$.

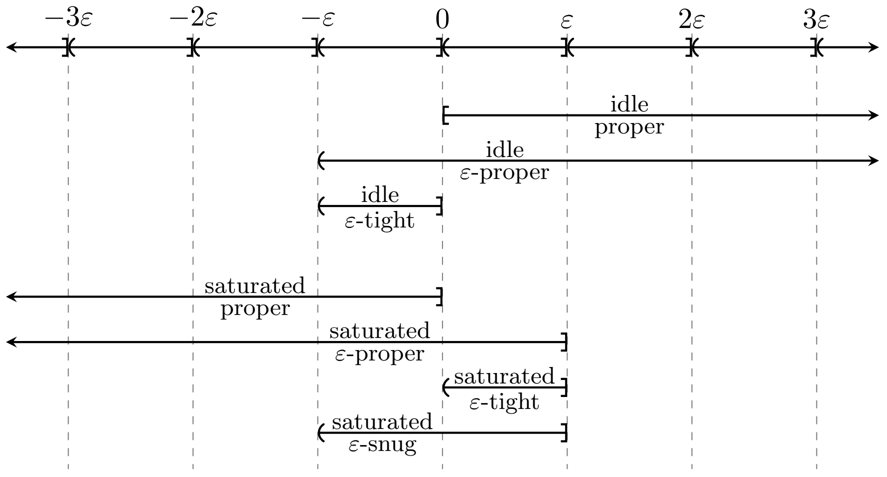

Figure 6.1: Classifying an arc based on its net cost

define a saturated arc $v \to w$ to be ε-*snug* when $-\varepsilon < c_p(v, w) \leq \varepsilon$, a weaker condition than ε-tight. Note that we define ε-snugness only for saturated arcs, not for idle arcs.

Figure 6.1 shows how idle and saturated arcs are classified into these various classes, based on their net costs. Note that all classes are right-closed and all classes except for the idle-and-proper class are left-open.

6.1.1 An aside on ceilings versus floors

Our choice of ceiling quantization could be viewed as unfortunate, since floors seem a bit simpler than ceilings. Of course, we have $\lceil t \rceil = - \lfloor -t \rfloor$, so ceilings are, at worst, only slightly more complicated than floors.

If we talked about maximizing benefit in presenting *FlowAssign*, rather than minimizing cost, then the formula for classifying an arc would be $\lfloor b_p(v, w)/\varepsilon \rfloor$, with a floor. But minimizing cost is convenient when discussing the Hungarian Method or similar algorithms. The net costs of proper, idle arcs are nonnegative, while the net costs of proper, saturated arcs are nonpositive. When we then compute the length of an augmenting path, we add the net costs of its idle arcs and subtract the net costs of its saturated arcs (since we back up along them); so our path lengths are nonnegative, which is a natural fit for a Dijkstra-like search. If we were dealing with net benefits, instead of net costs, it would be clumsy to end up with path lengths that were nonnegative. So we are sticking with minimizing cost.

It's not completely clear why floors are typically thought of as simpler than ceilings, by the way; but here is one relevant issue. Let k be some integer. Modern computers represent k using two's complement, with the digits 0 and 1. As a result, we can compute, say, $\lfloor k/16 \rfloor$ simply by shifting the representation of k four bits to the right. This works for $k \geq 0$ and also for $k < 0$, as long as a right shift of a negative value shifts in 1's at the left end. Computing $\lceil k/16 \rceil$ is a little clumsier.

```
FlowAssign$(G, t)$
    $(M, s) :=$ HopcroftKarp$(G, t)$;
    convert $M$ into an integral flow $f$ on $N_G$ with $|f| = s$;
    set $\varepsilon := \overline{\varepsilon}$ and, for all nodes $v$ in $N_G$, set $p_d(v) := 0$;
    while $\varepsilon > \underline{\varepsilon}$ do
        $\varepsilon := \varepsilon / q$;
        Refine$(f, p, \varepsilon)$;
    od;
    round prices to integers that make all arcs proper;
```

Figure 6.2: The high-level structure of *FlowAssign*

If we liked, we could instead represent integers using two's complement, but with the digits 0 and -1. In this scheme, a sign bit of -1 would mean a positive number, while a sign bit of 0 would mean either a negative number or 0. Thus, zero would become "the largest negative number" instead of, as we are used to, "the smallest positive number". Using this scheme, it would be $\lceil k/16 \rceil$ that could be computed simply by right-shifting; so we might well then think of ceilings as simpler than floors.

6.2 The high-level structure of *FlowAssign*

Figure 6.2 shows the high-level structure of the algorithm *FlowAssign*, but with the details about the scaling phase *Refine* elided for now.

The real parameter ε specifies how close our current prices come to being proper. We also introduce an integer $q \geq 2$. For now, we leave q as a free parameter, perhaps chosen to depend in some way upon the parameters n, m, r, and s. In the end, however, we will choose q to be a small constant. The integer q tells us the factor by which ε is reduced, in moving from one scaling phase to the next.

The primary state of *FlowAssign* consists of the flux f, the prices p, and the real number ε. Here are four invariant properties of that state:

I1 The flux f on N_G is a flow of value $|f| = s$.

I2 The prices at all nodes in N_G are multiples of ε.

I3 Every arc of N_G, idle or saturated, is ε-proper.

I4 Every saturated bipartite arc is ε-snug.

At the start of each scaling phase, ε is reduced by a factor of q. This reduction weakens **I2**, but strengthens **I3** and **I4**. The routine *Refine* begins with special actions that reestablish **I3** and **I4**, given the new, smaller ε.

In each call to *Refine*, we have $\varepsilon = q^e$ for some integer e. The initial value $\overline{\varepsilon} = q^{\overline{e}}$ is the smallest power of q that strictly exceeds C; so we set $\overline{e} := 1 + \lfloor \log_q C \rfloor$. The final $\underline{\varepsilon} = q^{\underline{e}}$ is the largest power of q that is strictly less than $1/(s+2)$; so we set $\underline{e} := -\bigl(1 + \lfloor \log_q (s+2) \rfloor\bigr)$. (We suggest pronouncing these as “e-up”, “e-down”, “ε-up”, and “ε-down”.) The number of calls to *Refine* is

$$\overline{e} - \underline{e} = O(\log_q(sC)) = O(\log(sC)/\log q). \tag{6-3}$$

The *early scaling phases* are those with $e \geq 0$, so that ε is a positive integer; the *late phases* are those with $e < 0$, so that ε is the reciprocal of an integer.

FlowAssign has to do arithmetic on costs, which are integers, and on prices and net costs, which are rational numbers. But the integer $1/\underline{\varepsilon} = q^{-\underline{e}}$ is a global common denominator — a common denominator for every cost, price, and net cost that ever arises. For simplicity, *FlowAssign* represents these quantities as rational numbers with this denominator, that is, as integer multiples of $\underline{\varepsilon}$. Corollary 9-5 shows that the prices remain $O(qsC)$. Since $1/\underline{\varepsilon} = O(qs)$, the numerators that we manipulate are $O(q^2s^2C)$, so, if we set q to be a constant, $q = O(1)$, then triple precision will suffice.

It might be more efficient in practice, rather than storing the prices as integer multiples of $\underline{\varepsilon}$, to store them as integer multiples of the current ε, since the resulting integers would be smaller. But doing this would require that we scale up all prices by a factor of q in moving from one phase to the next, to compensate for ε being scaled down by that factor. Using the current ε as our unit of measure would also require doing something special about the costs, since the costs may not be multiples of ε during the early phases. In any case, this optimization couldn't improve performance by more than a constant factor; so we stick to the simpler plan of representing all prices and costs, throughout, as multiples of $\underline{\varepsilon}$.

Returning to Figure 6.2, we begin the processing in *FlowAssign* by using Hopcroft-Karp, as described in Section 5, to compute some matching of size $s = \min(t, \nu(G))$, which takes time $O(m\sqrt{s})$. We convert the Hopcroft-Karp matching into an integral flow f in the network N_G, with $|f| = s$. Then we set $\varepsilon := \overline{\varepsilon}$ and we set the prices at all nodes of N_G to zero. This establishes all four of our invariants:

I1 The flux f is a flow of value $|f| = s$, as required.

I2 All prices are zero, so they are multiples of anything, including ε.

I3 Because the prices are zero, the net cost of every arc equals its cost; and the magnitude of every such cost is at most C. Since $\overline{\varepsilon} > C$, we conclude that $-\overline{\varepsilon} < c_p(x, y) < \overline{\varepsilon}$, for every bipartite arc $x \to y$. So all bipartite arcs are $\overline{\varepsilon}$-proper, whether they are idle or saturated. As for the dummy arcs, their net costs are all currently zero; so they are also $\overline{\varepsilon}$-proper, whether idle or saturated. In fact, they are currently proper; but that won't last.

I4 We have just seen that any saturated bipartite arc $x \to y$ must have $-\overline{\varepsilon} < c_p(x, y) < \overline{\varepsilon}$, and is hence ε-snug.

So the core of the algorithm can now commence, with repeated reductions of ε and calls to *Refine*. We'll analyze that core in later chapters.

6.3 Rounding the final prices

The final call to *Refine* makes all arcs ε-proper, where $\varepsilon = \underline{\varepsilon} < 1/(s+2)$. After that call, we round our prices to integers by computing

$$\widetilde{p}_d(v) := \lfloor p_d(v) + k\underline{\varepsilon} \rfloor, \tag{6-4}$$

for all nodes v in N_G, where k is a carefully chosen integer in the range $[0 \mathrel{..} 1/\underline{\varepsilon})$, the same k for rounding all prices. We now discuss choosing k so that the rounded prices make all arcs proper.

The rounding operation $u \mapsto \lfloor u + k\underline{\varepsilon} \rfloor$ is monotonic and commutes with integer shifts; so an arc that is proper before we round will remain proper afterward. For example, an idle arc $v \to w$ that is proper before we round has $c(v,w) + p_d(w) \geq p_d(v)$; this implies that $c(v,w) + \widetilde{p}_d(w) \geq \widetilde{p}_d(v)$, so the arc will be proper afterward as well. And the argument for saturated arcs is the same, but with the inequalities reversed.

We claim next that all of the idle arcs are proper before we round. Since all costs are integers and $\underline{\varepsilon}$ is the reciprocal of an integer, all costs are multiples of $\underline{\varepsilon}$. All prices are multiples of $\underline{\varepsilon}$ as well, by **I2**, so all net costs are multiples of $\underline{\varepsilon}$. All idle arcs, which are $\underline{\varepsilon}$-proper by **I3**, are then automatically proper by Prop 6-2.

So our goal is to find a k for which the rounding will take all of the saturated arcs that are improper before we round and convert them into proper arcs. To understand the issues involved, let's consider an example. Suppose that $\underline{\varepsilon} = 1/16$ and let $v \to w$ be a saturated arc. So the prices at v and at w are multiples of $1/16$, the cost $c(v,w)$ is some integer, and the arc $v \to w$ is $(1/16)$-proper, which means that

$$c_p(v,w) = c(v,w) - p_d(v) + p_d(w) \leq 1/16.$$

The arc $v \to w$ might already be proper. If not, the only possibility is that $c_p(v,w) = c(v,w) - p_d(v) + p_d(w) = 1/16$. For example, we might have $c(v,w) = 3$, $p_d(v) = 4\frac{5}{16}$, and $p_d(w) = 1\frac{3}{8}$.

When we round prices, replacing p with $\tilde{p}$, the price at v will become $\widetilde{p}_d(v) \in \{4,5\}$, while the price at w will become $\widetilde{p}_d(w) \in \{1,2\}$. Precisely what happens will depend upon the integer k, chosen from $[0 \mathrel{..} 16)$. If we choose k in $[0 \mathrel{..} 9]$, then both prices round down, with $\widetilde{p}_d(v) = 4$ and $\widetilde{p}_d(w) = 1$. As a result, the arc $v \to w$ becomes proper. If we choose k in $[11 \mathrel{..} 15]$, then both prices round up, with $\widetilde{p}_d(v) = 5$ and $\widetilde{p}_d(w) = 2$. And, again, the arc $v \to w$ becomes proper. A difficulty arises only in one case out of 16: If we choose $k = 10$, then the price at v rounds down to 4 while the price at w rounds up to 2, causing the arc $v \to w$ to be improper by the large margin of unity: $c_{\tilde{p}}(v,w) = 1$. We avoid this problem simply by being careful not to choose $k = 10$.

Each saturated arc $v \to w$ that is currently improper determines exactly one bad value for k in a similar way, and we can determine that bad value by looking at the fractional part of either $p_d(v)$ or $p_d(w)$. The bad choice is the largest k that causes $p_d(v)$ to round down, which is also the smallest k that causes $p_d(w)$ to round up. If an arc is $\underline{\varepsilon}$-proper but improper, then that arc must be saturated and those two values of k must coincide.

The current flow f in the network N_G has precisely $3s$ arcs that are saturated. Each of the s saturated bipartite arcs might rule out a distinct possibility for k. But note that, of the s saturated left-dummy arcs, all of the ones that are currently improper, however many of them there are, must rule out the same possibility for k, since all left-dummy arcs leave the same node: the source. In a similar way, of the s saturated right-dummy arcs, all of the ones that are currently improper must rule out the same possibility for k. As a result, at most $s+2$ possibilities are ruled out. Since $1/\underline{\varepsilon} \geq s+3$, we will be able to choose a k that is not bad for any arc. (More concretely, we mark each integer in the range $[0 \mathrel{..} 1/\underline{\varepsilon})$ as tentatively good. We then consider each saturated arc in turn and, if it determines a bad value for k, we clear the corresponding mark. Finally, we scan for an integer that is still marked good.) Rounding all prices using this good value for k generates integral prices that make all arcs proper, thus demonstrating that the matching M corresponding to the flow f is indeed min-cost.

Chapter 7

Introducing *Refine*

The routine *Refine*, shown in Figure 7.1, carries out a scaling phase similar to those in Gabow-Tarjan. As in the Hungarian Method, the main loop starts with a Dijkstra-like search to build a shortest-path forest, followed by a round of price increases. But then, as in Hopcroft-Karp, we augment, not just along the single length-0 augmenting path that our price increases have ensured, but along a maximal set of compatible such paths.

7.1 The pseudoflows in *Refine*

Refine deals with pseudoflows on the network N_G, rather than flows, but pseudoflows with a simple structure. If f is a pseudoflow on N_G, we define a *surplus of f* to be a node other than the sink at which the entering flow exceeds the leaving flow. And we define a *deficit of f* to be a node other

```
Refine($f, p, \varepsilon$)
    $S$ := {the $s$ women who are matched in $f$};
    $D$ := {the $s$ men who are matched in $f$};
    convert the $s$ bipartite arcs that are saturated in $f$ to idle;
    raise the prices $p$, as in Figure 7.4, to make all arcs $\varepsilon$-proper;
    int $h := s$;
    while $h > 0$ do
        build a shortest-path forest from the current surpluses $S$,
                stopping when a current deficit in $D$ is reached;
        raise prices at forest nodes by multiples of $\varepsilon$, shortening
                the discovered augmenting path to length 0;
        find a maximal set $\boldsymbol{\mathcal{P}}$ of length-0 augmenting paths
                that are compatible, as defined in Section 8.3;
        augment $f$ along each of the paths in $\boldsymbol{\mathcal{P}}$ in turn, thereby
                reducing $|S| = |D| = h$ by $|\boldsymbol{\mathcal{P}}|$;
    od;
```

Figure 7.1: The high-level structure of *Refine*

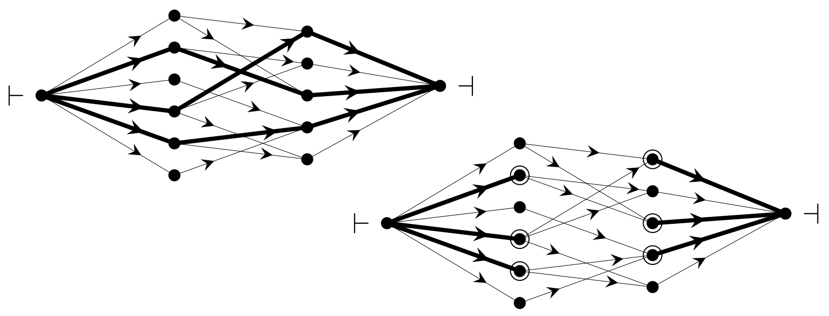

Figure 7.2: Converting the input flow into a pseudoflow at the start of *Refine*. Zeroing the flow on the three bipartite arcs that were saturated results in three surpluses and three deficits, which are circled.

than the source at which the leaving flow exceeds the entering flow. So a pseudoflow qualifies as a flow just when it has no surpluses and no deficits.

For a woman x in X, let the *left stub to x* be the pseudoflow that saturates the left-dummy arc $\vdash \to x$, but leaves all other arcs idle. Symmetrically, for a man y in Y, the *right stub from y* saturates only the right-dummy arc $y \to \dashv$. Any pseudoflow f that arises in *Refine* is the sum of some flow, some left-stubs, and some right-stubs. The flow component, which we denote $\hat{f}$, encodes the partial matching that *Refine* has constructed so far, during this scaling phase. We initialize $\hat{f}$ to zero, so this matching starts out empty. The left-stubs remember those women who were matched at the end of the previous phase and who have not yet been either matched or replaced during this phase. Those women are the surpluses of the pseudoflow f, and they constitute the set S. The right-stubs remember the previously matched men in a similar way. Those men are the deficits of f, and they constitute D.

During *Refine*, we replace the invariant **I1** with **I1′**:

I1′ The flux f on N_G is a pseudoflow consisting of an integral flow $\hat{f}$ of value $|\hat{f}| = s - h$ supplemented by left stubs to each of the women in S and by right stubs from each of the men in D, where $|S| = |D| = h$.

Note that **I1** is that special case of **I1′** in which $h = 0$, so there are no stubs and the flux $f = \hat{f}$ is itself an integral flow.

When *Refine* is called, f is an integral flow of value $|f| = s$. But *Refine* starts by altering f so as to zero the flow along the s bipartite arcs that were saturated. Figure 7.2 shows an example with $s = 3$. The initialization of *Refine* then raises prices so that every arc in N_G becomes ε-proper, for the resulting pseudoflow f and for the new, smaller value of ε.

The rest of *Refine*, its *main loop*, finds augmenting paths and augments along them, each such path joining a node in S to a node in D, that is, a surplus to a deficit. By augmenting along s such paths, we return f to being a flow once again, but now with all arcs ε-proper, rather than just $(q\varepsilon)$-proper. Unlike in Gabow-Tarjan, however, our augmenting paths are allowed to visit the source and the sink, as we discuss next.

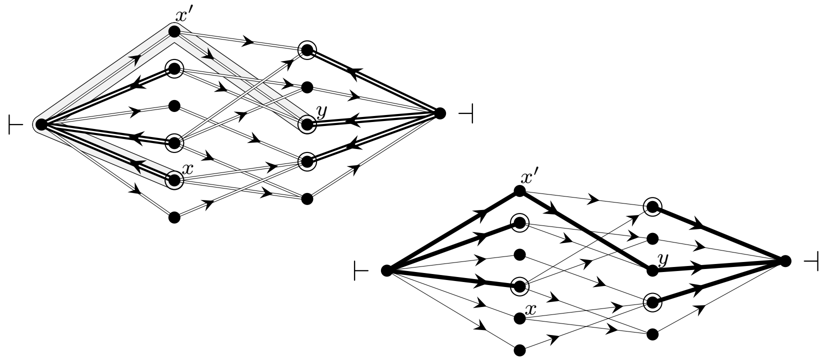

Figure 7.3: An augmenting path of length 3 in the residual digraph of the initial pseudoflow and the new pseudoflow that results from augmenting along that path

7.2 The residual digraph R_f

Given an integral pseudoflow f on the network N_G, we define the *residual digraph* R_f as follows: Each node in N_G becomes a node in R_f. Each idle arc $v \to w$ becomes a forward link $v \Rightarrow w$ in R_f; and each saturated arc $v \to w$ becomes a backward link $w \Rightarrow v$.[*] Note that, because our flow network N_G includes source and sink nodes, the links along a path in R_f need not alternate between forward and backward.

An *augmenting path* is a simple path in the residual digraph R_f that starts at a surplus and ends at a deficit. An augmenting path is allowed to visit either the source or the sink or both (in either order); but it can visit each of them at most once, since it is simple. Figure 7.3 shows an example of an augmenting path of length 3 that visits the source. This path starts at the surplus x, leaving that surplus along the backward link $x \Rightarrow \vdash$. It then follows the forward link $\vdash \Rightarrow x'$ and the forward link $x' \Rightarrow y$, arriving at the deficit y. When we augment along that path, the arcs underlying these three links reverse their idle-versus-saturated status, thus recording the fact that the set of women who are going to end up married has changed, with x' replacing x in that set.

The residual digraphs R_f that arise during *Refine* are more complex than the digraphs R_M that arose in the Hungarian Method. Despite this complexity, the simple rules from Prop 3-1 about the in-degrees of maidens and the out-degrees of bachelors in R_M have equally simple analogs for R_f — but now it's the in-degrees of surpluses and the out-degrees of deficits.

Prop 7-1. *In the residual digraph R_f for any pseudoflow f that arises during* Refine*, the in-degree of any non-surplus woman is* 1*, while the in-degree of*

[*]Recall the three levels of terminology that we first discussed in Section 2.1: The original graph G has vertices and edges, each edge (x, y) having a cost $c(x, y)$; the flow network N_G has nodes and arcs, each arc $v \to w$ going forward and having a net cost $c_p(v, w)$; and the residual digraph R_f has nodes and links, some links going forward and some backward, while each link $v \Rightarrow w$ will have an integral length $l_p(v \Rightarrow w) \geq 0$.

any surplus is 0*. Symmetrically, the out-degree of any non-deficit man is* 1*, while the out-degree of any deficit is* 0*.*

Proof. Consider a woman x, and consider a link in R_f that arrives at x. Such a link can arise in one of two ways: either because the left-dummy arc $\vdash \to x$ is idle, leading to the forward link $\vdash \Rightarrow x$, or because some bipartite arc leaving x, say $x \to y$, is saturated, leading to the backward link $y \Rightarrow x$. Appealing to **I1′**, any bipartite arc that is saturated in f must be saturated also in the flow $\hat{f}$, since no stub saturates any bipartite arcs. Since flow is conserved at x in the flow $\hat{f}$, there can't be more than one bipartite arc leaving x that is saturated, and there can't be even one such saturated arc unless the left-dummy arc $\vdash \to x$ that enters x is also saturated. So the in-degree of x in R_f is at most 1. Furthermore, the node x is a surplus just when the left-dummy arc $\vdash \to x$ is saturated and no bipartite arc leaving x is saturated; so the in-degree of x is then 0.

The argument for the out-degree of a man is symmetric. □

Corollary 7-2. *On any augmenting path that arises during* Refine*, the only surplus is the surplus at which it starts and the only deficit is the deficit at which it ends.*

7.3 The lengths of the links in R_f

We now associate a nonnegative length with each link in the residual digraph R_f. In the Hungarian Method, the lengths of links were real numbers; but they are integers in *Refine* because, as in Gabow-Tarjan, we quantize our net costs to multiples of ε. More precisely, we do ceiling quantization. A forward link $v \Rightarrow w$ in the residual digraph R_f arises from an idle arc $v \to w$, and we define the *length* of that forward link to be

$$l_p(v \Rightarrow w) := \left\lceil \frac{c_p(v,w)}{\varepsilon} \right\rceil . \qquad (7\text{-}3)$$

Since all arcs are maintained ε-proper by **I3**, the idle arc $v \to w$ will have $c_p(v,w) > -\varepsilon$, so we have $l_p(v \Rightarrow w) \geq 0$. A backward link $w \Rightarrow v$ arises from a saturated arc $v \to w$, and we define the *length* of that backward link to be

$$l_p(w \Rightarrow v) := 1 - \left\lceil \frac{c_p(v,w)}{\varepsilon} \right\rceil . \qquad (7\text{-}4)$$

A saturated arc $v \to w$ that is ε-proper has $c_p(v,w) \leq \varepsilon$, so the value of the ceiling is at most 1, and it could be large negative. Thus, the length of any backward link is also a nonnegative integer: $l_p(w \Rightarrow v) \geq 0$.

In the Hungarian Method, all saturated arcs were bipartite and were kept tight, so all backward links had length zero. In *Refine*, however, we have dummy arcs, some of which are saturated, and their net costs might be large negative. So we have to define the lengths of backward links in *Refine*, and the minus sign in front of the ceiling in equation (7-4) makes sense, since the backward link $w \Rightarrow v$ is the reverse of the saturated arc $v \to w$. Indeed, the presence of that minus sign simplifies the impact of a price increase on the lengths of the affected links.

Prop 7-5. *In a round of price increases in the main loop of* Refine*, raising the price $p_d(v)$ at some node v in N_G by ε lowers by* 1 *the length of any link in the residual digraph R_f that leaves v and raises by* 1 *the length of any link that enters v.*

Proof. A link $v \Rightarrow w$ leaving v is either forward or backward. If it is forward, the arc underlying it is the idle arc $v \to w$. Raising the price at v by ε lowers the net cost of this arc by ε, which, by equation (7-3), lowers the length of the link by 1. If the link $v \Rightarrow w$ is backward, the arc underlying it is the saturated arc $w \to v$. Raising the price at v by ε raises the net cost of this arc by ε, which, by equation (7-4), also lowers the length of the link by 1.

Links, either forward or backward, that enter v are a similar story. □

So the minus sign in equation (7-4) makes good sense; but what about the offset of $+1$? We obviously need some offset, since Dijkstra's algorithm for shortest paths requires our lengths to be nonnegative. One could argue that our offsets of 0 in equation (7-3) and $+1$ in equation (7-4) have somewhat the same effect that offsets of $-\frac{1}{2}$ and $+\frac{1}{2}$ would have, where those offsets could be justified as removing the upward bias of the ceiling function. But the real reason that we adopt the offsets that we do is the following.

Prop 7-6. *In* Refine*, a link in the residual digraph has length* 0 *just when the arc underlying it is ε-tight; so an augmenting path has length* 0 *just when all of its links are ε-tight. And a backward link has length at most* 1 *just when the saturated arc underlying it is ε-snug.*

Proof. An idle arc $v \to w$ is ε-tight when $-\varepsilon < c_p(v,w) \le 0$, in which case equation (7-3) gives us $l_p(v \Rightarrow w) = 0$. A saturated arc $v \to w$ is ε-tight when $0 < c_p(v,w) \le \varepsilon$, in which case equation (7-4) gives us $l_p(w \Rightarrow v) = 0$; and it is ε-snug when $-\varepsilon < c_p(v,w) \le \varepsilon$, in which case $l_p(w \Rightarrow v)$ must be either 0 or 1. □

Now that the links in R_f have lengths, we can add our final invariant. During *Refine*, we maintain **I1′**, **I2**, **I3**, and **I4**, which we here repeat for reference, and we add **I5**:

I1′ The flux f on N_G is a pseudoflow consisting of an integral flow $\hat{f}$ of value $|\hat{f}| = s - h$ supplemented by left stubs to each of the women in S and by right stubs from each of the men in D, where $|S| = |D| = h$.

I2 The prices at all nodes in N_G are multiples of ε.

I3 Every arc of N_G, idle or saturated, is ε-proper.

I4 Every saturated bipartite arc is ε-snug.

I5 The residual digraph R_f has no cycles of length zero.

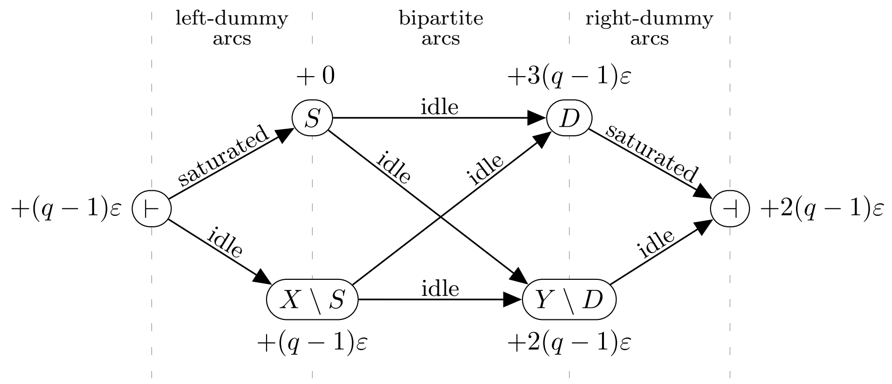

Figure 7.4: Price increases during the initialization of *Refine*

7.4 Before the main loop starts

As *Refine* begins, zeroing the flow along the s bipartite arcs that started out saturated leaves all bipartite arcs idle. This establishes **I1′** with $h = s$ and with $\hat{f} = 0$. It also establishes **I4** trivially. In addition, there are now no links $y \Rightarrow x$ in the residual digraph that go backward from the men's side to the women's side; hence, there can't be any cycles at all in the residual digraph, so **I5** holds, whatever the prices might be. As for **I2**, all prices start as multiples of $q\varepsilon$, and hence also multiples of ε. To establish the remaining **I3**, we must make all arcs ε-proper, which we do by raising prices as indicated in Figure 7.4. Note that these increases are all multiples of ε, so we don't invalidate **I2** in the process of establishing **I3**.

What can we say about the net costs of the various arcs, before we raise any prices? The saturated left-dummy arcs $\vdash \to x$ are $(q\varepsilon)$-proper, so they satisfy $c_p(\vdash, x) \le q\varepsilon$. The idle left-dummy arcs $\vdash \to x$ are $(q\varepsilon)$-proper, all of the prices are multiples of $q\varepsilon$, and the costs of all dummy arcs are zero. It follows, from Prop 6-2, that the idle left-dummy arcs are actually proper, with $c_p(\vdash, x) \ge 0$. Symmetrically, the saturated right-dummy arcs $y \to \dashv$ satisfy $c_p(y, \dashv) \le q\varepsilon$ and the idle right-dummy arcs $y \to \dashv$ are actually proper, with $c_p(y, \dashv) \ge 0$. The bipartite arcs $x \to y$ come in two flavors. Some of them were idle also in the flow that was in effect when *Refine* was called. Those arcs were idle and $(q\varepsilon)$-proper, so they satisfy $c_p(x, y) > -q\varepsilon$. The others are idle now, but they were saturated when *Refine* was called. By **I4**, we conclude that $c_p(x, y) > -q\varepsilon$ also for those arcs.

We now walk around the hexagon in Figure 7.4 counterclockwise, verifying that the indicated price increases leave all arcs ε-proper. Let's use p' to denote the prices after we have raised them.

Consider first a saturated left-dummy arc $\vdash \to x$. We start with

$$c_p(\vdash, x) = c(\vdash, x) - p_d(\vdash) + p_d(x) \le q\varepsilon.$$

The surplus x lies in S, so we leave the price at x unchanged: $p'_d(x) := p_d(x)$.

But we raise the price at the source: $p'_d(\vdash) := p_d(\vdash) + (q-1)\varepsilon$. So we have

$$c_{p'}(\vdash, x) = c(\vdash, x) - p'_d(\vdash) + p'_d(x) = c_p(\vdash, x) - (q-1)\varepsilon.$$

So $c_{p'}(\vdash, x) \leq \varepsilon$, and the saturated left-dummy arc $\vdash \to x$ is left ε-proper.

What about an idle left-dummy arc $\vdash \to x$? We start with $c_p(\vdash, x) \geq 0$. And we add $(q-1)\varepsilon$ to the prices at both $\vdash$ and at x; so we end with $c_{p'}(\vdash, x) \geq 0$. We need only $c_{p'}(\vdash, x) > -\varepsilon$ to leave the idle arc $\vdash \to x$ being ε-proper; but Prop 6-2 tells us that the only way to achieve $c_{p'}(\vdash, x) > -\varepsilon$ is to make $c_{p'}(\vdash, x) \geq 0$.

We consider the bipartite arcs $x \to y$ next. They are now all idle, and we have seen that $c_p(x, y) > -q\varepsilon$, whether the arc $x \to y$ was idle or saturated at the call to *Refine*. The price at x either stays the same or goes up by $(q-1)\varepsilon$, according as x does or does lie in S. So $p'_d(x) \leq p_d(x) + (q-1)\varepsilon$. The price at y goes up either by $3(q-1)\varepsilon$ or by $2(q-1)\varepsilon$, according as y does or does not lie in D. So $p'_d(y) \geq p_d(y) + 2(q-1)\varepsilon$. We thus have

$$\begin{aligned} c_{p'}(x, y) &= c(x, y) - p'_d(x) + p'_d(y) \\ &\geq c(x, y) - p_d(x) - (q-1)\varepsilon + p_d(y) + 2(q-1)\varepsilon \\ &\geq c_p(x, y) + (q-1)\varepsilon, \end{aligned}$$

from which it follows that $c_{p'}(x, y) > -\varepsilon$. Thus, all of the bipartite arcs are left idle and ε-proper.

The right-dummy arcs are similar to the left-dummy arcs. The idle ones start out proper and remain proper, since we raise the prices at both ends by the same amount. For the saturated ones, we raise the price at the left end by $(q-1)\varepsilon$ more than we raise the price at the right end. This ensures that the saturated right-dummy arcs are left ε-proper, so **Inv3** has been established, and we are ready for the main loop.

Chapter 8

The main loop in ***Refine***

Recall, from Figure 7.1, that the main loop of *Refine* iterates four steps. We now analyze each of those four steps in turn.

8.1 Building the shortest-path forest

We start the main loop of *Refine* by building a shortest-path forest, with the h surpluses remaining in S as the roots of the trees. This process is similar to what we did in the Hungarian Method: We are looking for an augmenting path whose links we can then bring to length 0 by raising prices appropriately. But there are some important differences.

- We need a path from a surplus to a deficit. In the Hungarian Method, we were satisfied with a path from any maiden to any bachelor.
- Our paths may visit the source and the sink; if they do so, then the links along them won't alternate between forward and backward.
- We use a different notion of path length. In the Hungarian Method, the length of a path was just the sum of the net costs of its forward links, all of its backward links having net cost zero. In *Refine*, both the forward and backward links can have positive length; and the lengths of all links are ceiling quantized to be multiples of the current ε.
- We don't need Fibonacci heaps in *Refine*. Since all path lengths are multiples of ε, we can follow Gabow-Tarjan in using the technique of Dial [7] to maintain our heap without any logarithmic overhead.
- *Refine* is also simpler in another way. When building a shortest-path forest in the Hungarian Method, we treated men and women differently, using our heap to find a shortest path to a man, but then deducing a shortest path to his wife as a corollary. In *Refine*, we treat all nodes the same: The women, the men, the source, and the sink all pass through the heap on their way into the forest. This means that r is no longer a bound on the size of the heap; but that doesn't matter, since we pay no logarithmic heap overhead.

```
BuildForest();
    make-heap();
    for all nodes $v$, set $\ell(v) := \infty$;
    for all surpluses $\sigma$ in $S$, set $\ell(\sigma) := 0$ and insert($\sigma, 0$);
    do  $v :=$ delete-min();
  scan: for all links $v \Rightarrow w$ leaving $v$ in $R_f$ do
              $L := \ell(v) + l_p(v \Rightarrow w)$; $L_{\text{old}} := \ell(w)$;
              if $L \leq \Lambda$ and $L < L_{\text{old}}$ then
                  set $\ell(w) := L$;
                  if $L_{\text{old}} = \infty$
                      then insert($w, L$)
                      else decrease-key($w, L$)
                  fi;
              fi;
        od;
        add $v$ to the forest;
    until $v$ is a deficit;
```

Figure 8.1: Building a shortest-path forest in *Refine*

- But be warned that *Refine* is also more complex in another respect. In the Hungarian Method, all women start out at price 0 and all men start out at price $\bar{C}$. In *Refine*, however, the prices at the nodes start out with whatever complicated structure has resulted from earlier calls to *Refine*. We will analyze the changes to those prices, proving analogs of Props 3-6 and 3-7 for *Refine*. But the prices in *Refine* don't start out with any particular uniformity.

We stop building our shortest-path forest when a deficit first joins it. We will prove a bound in Corollary 9-4 on how long a path we might need, to first reach a deficit. For now, we just assume that Λ is some such bound. That is, whenever we start building a shortest-path forest, we assume that some path in R_f from some surplus to some deficit will be found whose length is at most Λ.

We build the heap using a Dijkstra-like search, as shown in Figure 8.1. This code is somewhat shorter than the analogous code in Figure 3.4, because we no longer treat women and men differently. The value $\ell(v)$, when finite, stores the minimum length of any path in R_f that we've found so far from some surplus to v. We use a heap to store those nodes v with $\ell(v) < \infty$ until they join the forest. The key of a node v in the heap is $\ell(v)$. The commands *insert*, *delete-min*, and *decrease-key* operate on that heap.

Since our keys are small integers and our heap usage is monotone[*], we can use Dial [7] to avoid any logarithmic heap overhead. We maintain an array Q,

[*]Cherkassky, Goldberg, and Silverstein [6] call a pattern of heap usage *monotone* if, whenever k is the key of a node that was just returned by a *delete-min*, then the key parameter in any future call to *insert* or *decrease-key* will be at least k. The Dial technique depends upon this monotonicity.

where $Q[k]$ points to a doubly-linked list of those nodes in the heap that have key k. Exploiting our assumed bound Λ, we allocate the array Q as $Q[0\,..\,\Lambda]$. We ignore any paths we find whose lengths exceed Λ. We also maintain an integer B, which stores the value $\ell(v)$ for the node v that was most recently added to the forest. We add nodes v to the forest in nondecreasing order of $\ell(v)$, so B never decreases. To implement *insert*(v, k), we add v to the list $Q[k]$. To implement *delete-min*, we look at the lists $Q[B]$, $Q[B+1]$, and so on, removing and returning the first element of the first nonempty list we find. To implement *decrease-key*(v, k), we exploit the double linking to remove v from the list $Q[\ell(v)]$ and then add v to the list $Q[k]$.

By our assumption about Λ, some deficit δ with $\ell(\delta) \le \Lambda$ will eventually enter the forest, at which point we stop building it. The space and time that we spend building it are both $O(m + \Lambda)$, where the Λ term accounts for the space taken by the array Q and for the time taken to scan that array once while doing *delete-min* operations.

By the way, suppose that every individual link had length at most K, for some $K \ll \Lambda$. The only lists in the array Q that could then be nonempty would be the lists $Q[B]$, $Q[B+1]$, through $Q[B+K]$. Thus, we could reduce the storage needed for the array Q by treating its indices modulo $K + 1$. That idea is also part of the Dial technique. But we don't exploit that idea here, since we don't have any good bound on the lengths of individual links.

8.2 Raising the prices

The next step in the main loop of *Refine* is to raise prices. For each node v in the shortest-path forest, we set the new (dispose) price $p'_d(v)$ by

$$p'_d(v) := p(d) + (\ell(\delta) - \ell(v))\varepsilon, \tag{8-1}$$

where δ is the deficit whose discovery halted the growth of the shortest-path forest. This is essentially the same formula that we used in the Hungarian Method, back in equation (3-3), except for the multiplication by ε, which we need here because our path lengths are now measured as multiples of ε.

Let σ be the surplus at the root of the tree that δ joins. We have $p'_d(\sigma) = p_d(\sigma) + \ell(\delta)\varepsilon$, but $p'_d(\delta) = p_d(\delta)$. So Prop 7-5 tells us that our price increases shorten the path from σ to δ by $\ell(\delta)$ length units. Since $\ell(\delta)$ was the length of that path before our price increases, its length after the increases will be zero. If our invariants are preserved, all of the links along that path must end up of length 0, meaning that all of the underlying arcs are ε-tight. As for our invariants, it's clear that **I1′** and **I2** continue to hold. But establishing the other three invariants takes more work.

For each node v in the network N_G, let's define $i(v)$ to be the multiple of ε by which we raise the price $p_d(v)$. So, for nodes v in the forest, we set $i(v) := \ell(\delta) - \ell(v)$, while, for nodes v not in the forest, we set $i(v) := 0$. We then have the repricing formula $p'_d(v) = p_d(v) + i(v)\varepsilon$, for all nodes v. Using this repricing formula, we can express the impact of our repricings on an arc

$v \to w$ without needing to know whether or not the nodes v and w lie in the forest. We have

$$c_p(v,w) = c(v,w) - p_d(v) + p_d(w)$$
$$c_{p'}(v,w) = c(v,w) - p'_d(v) + p'_d(w),$$

so we have

$$c_{p'}(v,w) = c_p(v,w) + (i(w) - i(v))\varepsilon. \tag{8-2}$$

Lemma 8-3. *Suppose that, at some point during the construction of the shortest-path forest, the node v in N_G enters the forest and the link $v \Rightarrow w$ in the residual digraph is scanned. We can then conclude that, after the forest's construction, we will have*

$$\ell(v) + l_p(v \Rightarrow w) \geq \ell(\delta) - i(w). \tag{8-4}$$

Proof. Three cases can arise, as we consider the length $L := \ell(v) + l_p(v \Rightarrow w)$ of the new path that we have found from some surplus to w.

First, we might find that $L > \Lambda$, so that we ignore the new path completely. But we are assuming that some path to a deficit is eventually found of length at most Λ, which means that $\Lambda \geq \ell(\delta)$. So we have $L > \ell(\delta)$ in this case, which implies inequality (8-4).

If the first case does not pertain, we next consider $\ell(w)$, which might be infinite, finite but larger than L, or at most L. Whichever of these three subcases applies, our processing of the link $v \Rightarrow w$ reduces $\ell(w)$ enough so that $L \geq \ell(w)$ then holds. During the remainder of the forest construction, $\ell(v)$ does not change and $\ell(w)$ can only decrease further. So we still have $L \geq \ell(w)$ after the forest-building has ceased.

When the forest-building has ceased, it might be that w has not entered the forest; that constitutes our second top-level case. We then have $\ell(w) \geq \ell(\delta)$, since δ has entered the forest and nodes enter the forest in nondecreasing order of their ℓ values. So we have $L \geq \ell(w) \geq \ell(\delta)$ in this case, which also implies inequality (8-4).

Finally, in our third top-level case, w has joined v the forest at some point during its construction, along with v. We then have $i(w) = \ell(\delta) - \ell(w)$, so $L \geq \ell(w)$ implies $L \geq \ell(\delta) - i(w)$, and (8-4) holds for the third time. $\square$

8.2.1 Idle arcs are left ε-proper

We now establish **I3** for idle arcs. Consider an idle arc $v \to w$ in the network N_G, which gives rise to the forward link $v \Rightarrow w$ in the residual digraph R_f. This arc was ε-proper before we raised our prices, so we had $c_p(v,w) > -\varepsilon$. We must show that $c_{p'}(v,w) = c_p(v,w) + (i(w) - i(v))\varepsilon > -\varepsilon$.

If v does not belong to the forest, we have $i(v) = 0$. The net cost of the arc $v \to w$ couldn't then have decreased, so the arc $v \to w$ ends ε-proper because it started ε-proper. It remains to consider the case in which v does belong to the forest.

If v entered the forest, then we must have scanned v, which means that we considered the forward link $v \Rightarrow w$. We can deduce from Lemma 8-3 that

$$\ell(v) + l_p(v \Rightarrow w) \geq \ell(\delta) - i(w).$$

Since $v \Rightarrow w$ is a forward link and $u + 1 > \lceil u \rceil$ for all real u, we have

$$\ell(v) + \frac{c_p(v,w)}{\varepsilon} + 1 > \ell(v) + \left\lceil \frac{c_p(v,w)}{\varepsilon} \right\rceil \geq \ell(\delta) - i(w).$$

Multiplying through by ε and rearranging, we find that

$$c_p(v,w) + \big(i(w) - (\ell(\delta) - \ell(v))\big)\varepsilon > -\varepsilon.$$

Since v belongs to the forest, we have $i(v) = \ell(\delta) - \ell(v)$, so $c_{p'}(v,w) > -\varepsilon$. Thus, our price increases do indeed leave all idle arcs ε-proper.

8.2.2 Saturated arcs are left ε-proper

To finish verifying **I3**, we next consider a saturated arc $v \to w$ in the network N_G, which gives rise to the backward link $w \Rightarrow v$ in the residual digraph R_f. This arc was ε-proper before we raised prices, so we had $c_p(v,w) \leq \varepsilon$. Our goal is to show that $c_{p'}(v,w) = c_p(v,w) + (i(w) - i(v))\varepsilon \leq \varepsilon$.

If w does not belong to the forest, we have $i(w) = 0$, so the arc $v \to w$ ends ε-proper because it started ε-proper. On the other hand, if w belongs to the forest, we must have scanned w and considered the link $w \Rightarrow v$. Applying Lemma 8-3 with v and w reversed, we deduce that

$$\ell(w) + l_p(w \Rightarrow v) \geq \ell(\delta) - i(v).$$

Since $w \Rightarrow v$ is a backward link and $-u \geq -\lceil u \rceil$ for all real u, we have

$$\ell(w) + 1 - \frac{c_p(v,w)}{\varepsilon} \geq \ell(w) + 1 - \left\lceil \frac{c_p(v,w)}{\varepsilon} \right\rceil \geq \ell(\delta) - i(v).$$

Multiplying through by ε and rearranging, we find that

$$\varepsilon \geq c_p(v,w) + \big((\ell(\delta) - \ell(w)) - i(v)\big)\varepsilon.$$

Since w belongs to the forest, we have $i(w) = \ell(\delta) - \ell(w)$, so $c_{p'}(v,w) \leq \varepsilon$ as we hoped. Our price increases thus preserve **I3**.

8.2.3 Saturated bipartite arcs are left ε-snug

To show that **I4** is preserved, we consider a saturated, bipartite arc $x \to y$; we must show that $c_{p'}(x,y) = c_p(x,y) + (i(y) - i(x))\varepsilon > -\varepsilon$, which is the lower-bound part of ε-snugness. Since we are now dealing with a lower bound on the net cost of a saturated arc, though, our argument will have to differ significantly from our arguments for the two halves of **I3** above.

If x does not belong to the forest, we have $i(x) = 0$, so the arc $x \to y$ ends ε-snug because it started ε-snug. We henceforth assume that x belongs to the forest.

But how did x get into the forest? Since x is the tail of the saturated bipartite arc $x \to y$, the node x isn't a surplus; so x wasn't put into the forest initially. Since the arc $x \to y$ is saturated, the corresponding link in the residual digraph is the backward link $y \Rightarrow x$, which arrives at x. By Prop 7-1, the in-degree of a woman in the residual digraph never exceeds 1. So $y \Rightarrow x$ is the only link in the entire residual digraph that arrives at x. And the only way that x could have entered the forest is because y entered the forest first, causing the backward link $y \Rightarrow x$ to be scanned, and, sometime later, x was extracted from the heap with a *delete-min*.

It follows that $\ell(x)$ was determined entirely by the link $y \Rightarrow x$. So we have $\ell(x) = \ell(y) + l_p(y \Rightarrow x) = \ell(y) + 1 - \lceil c_p(x,y)/\varepsilon \rceil$. Since $u + 1 > \lceil u \rceil$ for all real u, we have

$$\frac{c_p(x,y)}{\varepsilon} + 1 > \left\lceil \frac{c_p(x,y)}{\varepsilon} \right\rceil = \ell(y) - \ell(x) + 1,$$

and hence $c_p(x,y) + (\ell(x) - \ell(y))\varepsilon > 0$. Since both x and y belong to the forest, we have $i(x) = \ell(\delta) - \ell(x)$ and $i(y) = \ell(\delta) - \ell(y)$, and thus $\ell(x) - \ell(y) = i(y) - i(x)$. So we find that $c_{p'}(x,y) > 0$ in this case, which is even stronger than the $c_{p'}(x,y) > -\varepsilon$ that we needed. So **I4** is preserved.

8.2.4 R_f remains free of length-0 cycles

To show that **I5** is preserved, we must show that raising the prices at the nodes in the shortest-path forest doesn't cause the residual digraph R_f to acquire any length-0 cycles. But we saw, in Prop 7-5, that increasing the price at a node v by ε lowers the lengths of all links leaving v by 1 and raises the lengths of all links entering v by 1. So price increases have no effect on the overall length of any cycle. Since **I5** assures us that there were no length-0 cycles before our price increases, there won't be any length-0 cycles after them either.

8.3 Finding compatible augmenting paths

The main loop of *Refine* has constructed a shortest-path forest and then raised prices to ensure that there exists some length-0 path in the residual digraph R_f from some surplus to some deficit. If we were mimicking the Hungarian Method, our next step would be to augment along that length-0 augmenting path. But we want $\sqrt{s}$ performance, so, like Gabow-Tarjan, we are going to start mimicking Hopcroft-Karp instead. We define what it means for augmenting paths to be "compatible". We then find a maximal set of compatible length-0 augmenting paths, and we augment along all of the paths in that set, before returning to construct a new forest.

In Hopcroft-Karp, augmenting paths are compatible just when they are vertex-disjoint; but we need a more generous notion of compatibility in *Refine*. Given the graph-theoretic properties of the residual digraph R_f, we can define our notion of compatibility in two different, equivalent ways.

8.3.1 Defining compatibility

We define augmenting paths to be *link-compatible* when they start at distinct surpluses, they end at distinct deficits, and they don't share any links. We define augmenting paths to be *node-compatible* when every woman is visited by at most one of the paths, and the same for every man — that is, the paths are node-disjoint, except for the source and sink.

Prop 8-5. *Augmenting paths are link-compatible if and only if they are node-compatible.*

Proof. It's easy to see that node-compatible augmenting paths are also link-compatible. By node-compatibility, they must start at distinct surpluses and end at distinct deficits. They also can't share any links, since every link has at least one end node that isn't the source or the sink.

Conversely, consider some augmenting paths that are link-compatible, and let x be a woman. If x is a surplus, then, by Corollary 7-2, an augmenting path can visit x only by starting at x, which only one of our link-compatible paths can do. If x is not a surplus, then an augmenting path can visit x only by arriving at x over a link. By Prop 7-1, the in-degree of x in R_f is at most 1, and at most one of our link-compatible paths can travel over any single link. So x is visited by at most one of our paths.

In a similar way, let y be any man. If y is a deficit, then an augmenting path can visit y only by ending at y, which only one of our paths can do. If y is not a deficit, an augmenting path can visit y only if it then leaves y along a link. But there is at most one link leaving y, which at most one of our paths can traverse. So link-compatible paths are also node-compatible. □

Since these two notions of compatibility are equivalent, we can henceforth refer to augmenting paths as simply being *compatible*, without specifying which flavor of compatibility we have in mind.

8.3.2 Finding a maximal set of compatible paths

Let R_f^0 denote that subgraph of the residual digraph formed by links of length zero. Note that we can construct an adjacency-list representation of R_f^0 in $O(m)$ time. The next step in the main loop of *Refine* is to find a maximal set $\boldsymbol{\mathcal{P}}$ of compatible augmenting paths in the subgraph R_f^0. We could search either for paths that are link-compatible or for paths that are node-compatible.

Figure 8.2 shows how to search for node-compatible paths using a depth-first search of the graph R_f^0. We build each path by pushing nodes onto a

```
    for $v$ in $N_G$ do set $v$ to be unmarked;
    set $K$ to empty;
    Surp := S;
A:  while Surp not empty do
        $x$ := first(Surp); Surp := rest(Surp);
        push($x, K$);
    B:  while $K$ not empty do
            $v$ := top($K$);
            if $v \neq \vdash$ and $v \neq \dashv$ then mark($v$) fi;
            if $v$ in $D$ then
                add $K$ to $\boldsymbol{\mathcal{P}}$ as an augmenting path;
                set $K$ to empty;
                goto A;
            fi;
        C:  while $L[v]$ not empty do
                $w$ := first($L[v]$); $L[v]$ := rest($L[v]$);
                if $w$ not marked then push($w, K$); goto B fi;
            od;
            pop($K$);
        od;
    od;
```

Figure 8.2: Finding the augmenting paths

stack K. For each node v, we have a list $L[v]$ of those nodes w for which the link $v \Rightarrow w$ in the residual digraph has length 0. When exploring the node v, we consider the successor nodes w in turn. And Surp is a list of the current surpluses, the nodes where an augmenting path could begin.

Each node has a mark bit, which we use to avoid examining nodes more than once. But we may need to examine the source and sink multiple times, since they can lie on multiple augmenting paths; so we never mark the source or the sink. Instead, each time that we return to the source or the sink, we resume where we left off in considering potential ways to continue an augmenting path from there. Failing to mark some node in a general graph would raise the risk that we would push that node onto the stack K at a time when that node was already on K, thus outputting a path that wasn't simple. But the graph R_f^0 is acyclic by **I5**, so we don't run that risk.

Each iteration of the C loop can be charged to the link $v \Rightarrow w$ being traversed. Of the iterations of the B loop with any particular value for the node v, the first can be charged to v itself, while any following iterations can be charged to the link $v \Rightarrow w$ that caused the stack to grow beyond v, after which it later shrank back to have v on top. Thus, our search for augmenting paths runs in $O(m)$ time.

The paths that we add to $\boldsymbol{\mathcal{P}}$ are clearly augmenting. And every node on any such path, except for the source and the sink, is marked when we add

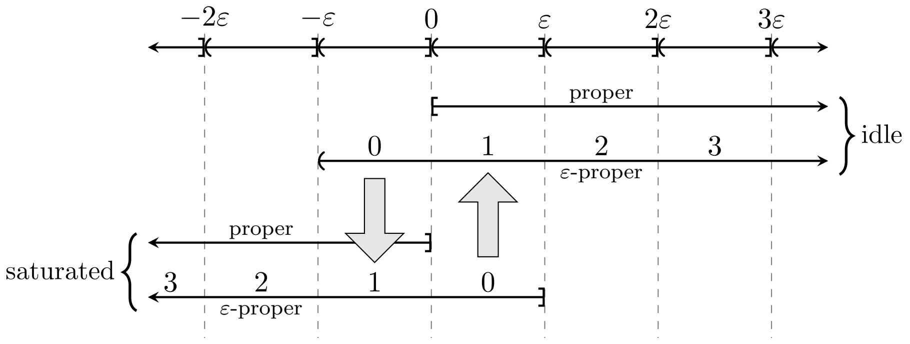

Figure 8.3: Augmentation's effects on the lengths of links in *Refine*

that path to $\mathcal{P}$, so different augmenting paths in $\mathcal{P}$ will be node-compatible, and hence compatible. Finally, when we consider starting at any surplus x, if there is any augmenting path starting at x that is compatible with the paths already in $\mathcal{P}$, we will find it and add it to $\mathcal{P}$. Thus, the set $\mathcal{P}$ that we construct is maximal. (It can easily fail to be maximum; but that's okay.)

If we liked, we could search for paths that are link-compatible, instead of node-compatible. The resulting program would be shorter than the program in Figure 8.2, since it could treat all nodes in the same way: the way in which Figure 8.2 treats the source and sink. We would still need mark bits, however, to mark those deficits reached by augmenting paths already in $\mathcal{P}$. And the program in Figure 8.2 will run faster, because it can cut off failing branches of the search tree sooner.

8.4 Augmenting along those paths

Finally, we augment the pseudoflow f along each of the paths in $\mathcal{P}$. These augmentations reverse the forward-versus-backward orientation of each link along the path and the idle-versus-saturated status of each underlying arc. This restores the flow balance of the surplus at which the path starts and of the deficit at which it ends, while no other flow-balances are affected. So **I1′** is preserved, but with $h = |S| = |D|$ reduced by $|\mathcal{P}|$. Augmenting doesn't change any prices, so **I2** is preserved. As for **I3**, the length-0 forward links along an augmenting path become length-1 backward links, while the length-0 backward links become length-1 forward links, as indicated in Figure 8.3. So all of the underlying arcs are left ε-proper, though no longer ε-tight. The formerly idle bipartite arcs that become saturated during the augmentation, while not left ε-tight, are left ε-snug, by Prop 7-6; so **I4** is also preserved. Finally, for **I5**: Augmentation reverses the directions of some links, and this may well produce cycles in R_f. Indeed, this is the only way that any cycles ever arise in R_f. But every link that changes state during an augmentation ends up being of length 1; so no cycle that exploits any such link can be of length 0.

Chapter 9

Analyzing the performance

To finish analyzing *FlowAssign*, we need three things. First, we must choose a value for the bound Λ, showing that the building of every shortest-path forest finds a path from a surplus to a deficit of length at most Λ. Second, we must show that our prices remain $O(sC)$. Third, to achieve the weight-scaling time bound, we must show that the main loop of *Refine* executes $O(\sqrt{s})$ times. The key to all three of these is the *inflation bound*, which limits the total amount by which prices can increase during a call to *Refine*.

9.1 The inflation bound

The inflation bound applies at any clean point in *Refine*'s main loop, where a *clean point* is just before or just after the execution of one of the four statements of the main loop. By restricting our attention to clean points, we don't have to worry about things being in some inconsistent state, say, because we are in the middle of augmenting along some augmenting path.

The inflation bound involves a quantity Δ, which we now define. In the round of price increases that follows the building of a shortest-path forest, the surpluses at the roots of the trees in that forest have their prices increased by $\ell(\delta)\varepsilon$, where δ is the deficit whose discovery stopped the building of the forest. Note that this is the largest increase that happens, during that round, to the price at any node. Let's refer to that quantity as the *max increase* of that round. At any clean point in the main loop of *Refine*, we define Δ to be the sum of the max increases of all of the rounds of prices increases that have happened so far, during this call to *Refine*.

Prop 9-1. *Consider any clean point during an execution of the main loop of* Refine. *Let Δ denote the sum of the max increases of all of the rounds of price increases so far, during this call to* Refine. *We then have the* inflation bound*:*

$$h\Delta \leq (4q + 4)s\varepsilon. \tag{9-2}$$

This bound holds even just after a round of price increases, during which Δ increased, and before the subsequent batch of augmentations, which will cause h to decrease.

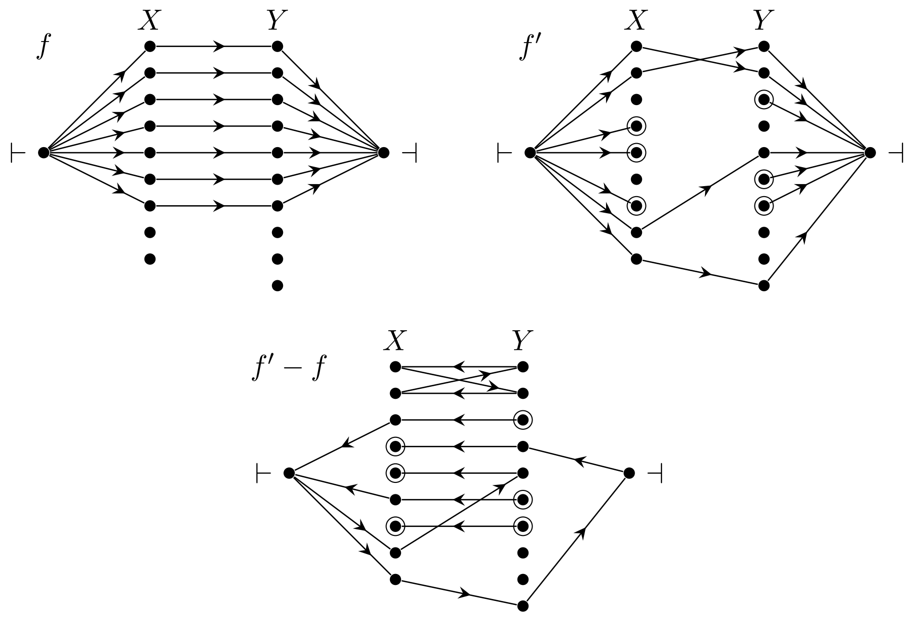

Figure 9.1: An example of the flux $f' - f$

Proof. We prove the inflation bound (9-2) by calculating a particular quantity, $(c_{p'} - c_p)(f' - f)$, in two different ways. But we must first define what we mean by this quantity.

Let f be the flow on the network N_G that was in effect when *Refine* was called. Let p be the prices that were in effect when control entered the main loop of *Refine*. Note that those two times are different. During the initialization code in *Refine*, we convert the input flow into a pseudoflow by zeroing out the flows along all bipartite arcs. We use the symbol f here to denote the input flow, before that zeroing out. We also raise the prices at the various nodes, as described in Figure 7.4, so as to make all arcs ε-proper with respect to the new pseudoflow. We use the symbol p here to refer to the prices after those increases have happened.

Let f' be the pseudoflow at the current clean point in the execution of *Refine*, and let p' be the prices at that same clean point. Our proof will compute the quantity $(c_{p'} - c_p)(f' - f) = c_{p'}(f' - f) - c_p(f' - f)$ in two different ways.

Figure 9.1 shows one example of what might happen — though it is hard for one small diagram to show all of the potential patterns. In Figure 9.1, we have $|X| = 9$, $|Y| = 10$, and $s = 7$. The input flow f of value $|f| = 7$ is laid out very simply in the diagram. We are considering that point in the execution of *Refine* at which there are $h = 3$ remaining surpluses and 3 remaining deficits, which are circled in the diagram of f'. Note that, of the four augmenting paths along which we have augmented so far, at least two of them have visited the source and at least one has visited the sink.

When we subtract f from f', we get the flux $f' - f$ shown on the bottom.

Note that this difference is just a flux, not even a pseudoflow, since it assigns a flow of -1 to various arcs. Since both f and f' have s units of flow leaving the source and s units entering the sink, flow is conserved in the difference flux $f' - f$ at both the source and the sink. Flow is conserved in f at all nodes other than the source and the sink. In f', on the other hand, there are h remaining unit surpluses in X and h remaining unit deficits in Y. Thus, in the difference flux $f' - f$, flow is conserved at all nodes except for those $2h$ nodes, which have the same status in $f' - f$ that they have in f'.

9.1.1 Calculating the value precisely

If a flux conserves flow at some node v, then changing the price at the node v has no effect on the net cost of the flux. Thus, to calculate the difference $(c_{p'} - c_p)(f' - f) = c_{p'}(f' - f) - c_p(f' - f)$, it suffices to consider the changes in price that happen at the $2h$ nodes where $f' - f$ does not conserve flow. Of those nodes, h are women with a unit surplus who have had that surplus since we entered the main loop; and the other h are men with a unit deficit who have had that deficit since we entered the main loop.

During the main loop of *Refine*, the only price changes that happen are the rounds of price increases that follow the construction of a shortest-path forest. Note that all h of the remaining surpluses have been roots of trees in every shortest-path forest that we have constructed so far. So all of them have had their prices increased by the max increase in every round of price increases so far. Thus, for each remaining surplus x, we have $c_{p'}(x) = c_p(x) + \Delta$. On the other hand, a round of price increases doesn't change the price of any current deficit. The shortest-path forest includes only one deficit: the deficit δ whose discovery stopped the growth of the forest. And the repricing formula specifies that the price at δ shouldn't change. Thus, for each remaining deficit y, we have $c_{p'}(y) = c_p(y)$.

So the flux $f'-f$ has h unit surpluses, at each of which the price increases by precisely Δ between p and p', and it has h unit deficits, at each of which the price doesn't change between p and p'. So $(c_{p'} - c_p)(f' - f) = h\Delta$.

9.1.2 Bounding the value

We next derive an upper bound on the quantity $(c_{p'} - c_p)(f' - f)$ by using our bounds on the net costs of individual arcs.

The flow f and the pseudoflow f' each assign a flow of either 0 or 1 to each arc in the network N_G, so we can think of each of them as a set of arcs. Taking their differences as sets, let $f' \setminus f$ denote those arcs that are saturated in f', but idle in f; and define $f \setminus f'$ symmetrically. We then have $f' - f = (f' \setminus f) - (f \setminus f')$, where the arcs saturated in both f and f' have been omitted from both terms on the right-hand side. So we have

$$(c_{p'} - c_p)(f' - f) = c_{p'}(f' \setminus f) - c_{p'}(f \setminus f') - c_p(f' \setminus f) + c_p(f \setminus f'). \quad (9\text{-}3)$$

We now consider each of these four sums in turn.

The first term $c_{p'}(f' \setminus f)$ sums the current net costs of those arcs that are saturated in f', but were idle in f. Since *Refine* maintains all arcs ε-proper by **I3**, any arc that is currently saturated has net cost at most ε. By **I1**$'$, the current pseudoflow f' saturates precisely s left-dummy arcs, $s-h$ bipartite arcs, and s right-dummy arcs. Some of those arcs may have been saturated also in f, in which case they won't contribute to $c_{p'}(f' \setminus f)$. But we certainly have $c_{p'}(f' \setminus f) \leq 3s\varepsilon$.

We boldly tackle the trickiest case next, which is the fourth and final term $c_p(f \setminus f')$ in equation (9-3). Recall that f is the flow that was in effect when *Refine* was called, but p is the prices that were in effect somewhat later, when *Refine* entered its main loop. There are precisely s left-dummy arcs in the flow f, so at most s in $f \setminus f'$. Each of those arcs $\vdash \to x$ was saturated when *Refine* entered its main loop, so we have $c_p(\vdash, x) \leq \varepsilon$ by **I3**. Thus, the left-dummy arcs contribute at most $s\varepsilon$; and a similar argument shows the same for the right-dummy arcs.

The bipartite arcs are the tricky ones. Let $x \to y$ be a bipartite arc that was saturated in f. When *Refine* was called, this arc was saturated and $(q\varepsilon)$-proper. During the initialization of *Refine*, we first changed the flow to make the arc $x \to y$ idle and we then changed prices as described in Figure 7.4. Since x was then a woman with a unit surplus, we left the price at x unchanged. But, since y was a man with a unit deficit, we added $3(q-1)\varepsilon$ to the price at y. It follows that $c_p(x,y) \leq q\varepsilon + 3(q-1)\varepsilon = (4q-3)\varepsilon$. There are at most s arcs of this type, so they contribute $(4q-3)s\varepsilon$ to our sum. Adding in the dummy arcs, we conclude that $c_p(f \setminus f') \leq (4q-1)s\varepsilon$.

What about the second term $-c_{p'}(f \setminus f')$ in (9-3)? This term subtracts off the net costs, in current prices, of those arcs $v \to w$ that were saturated in f, but are idle in f'. Since those arcs currently idle, **I3** tells us that $c_{p'}(v,w) > -\varepsilon$, and hence $-c_{p'}(v,w) < \varepsilon$. Furthermore, if the arc $v \to w$ is either left-dummy or right-dummy, then Prop 6-2 tells us, in fact, that $c_{p'}(v,w) \geq 0$, and hence $-c_{p'}(v,w) \leq 0$. So the dummy arcs $v \to w$ don't contribute anything. There were s bipartite arcs that were saturated in f, of which at most s could be currently idle, so we have $-c_{p'}(f \setminus f') < s\varepsilon$.

The third term $-c_p(f' \setminus f)$ in (9-3) is similar. It subtracts off the net costs, at the time of entry into the main loop, of those arcs $v \to w$ that are currently saturated, but were idle when *Refine* was called. Such an arc $v \to w$ must have been idle also when *Refine* entered its main loop, since converting f into a pseudoflow, while it does idle some saturated arcs, doesn't saturate any idle arcs. Applying **I3** and Prop 6-2 once again, now at the time of entry into the main loop, we conclude that $-c_p(f' \setminus f) < s\varepsilon$.

Adding everything together, we have $h\Delta \leq (4q+4)s\varepsilon$. $\square$

9.2 Determining the bound Λ

Using the inflation bound, we now establish that the bound Λ in Section 8.1 can be taken to be $\Lambda := (4q+4)s/h$.

Corollary 9-4. *The building of any shortest-path forest in the procedure* Refine *is always halted by finding a deficit δ with $\ell(\delta) \leq (4q+4)s/h$.*

Proof. Suppose that we are about to build a shortest-path forest. We have $h \geq 1$, so the matching that is encoded by the flow component $\hat{f}$ of the current pseudoflow f has size $s - h < s$. Since G has matchings of size at least s, the standard theory of matchings tells us that there must exist a path P in the residual digraph R_f from some maiden to some bachelor that alternates between links that are forward bipartite and links that are backward bipartite — no dummy links. The path P may not be an augmenting path in our sense, however, since the maiden μ at which it starts may not be a surplus and the bachelor β at which it ends may not be a deficit. If μ is not a surplus, however, then the left-dummy arc $\vdash \to \mu$ must be idle; so we can tack two more links onto the beginning of P as follows: We choose any surplus we like, say σ; we follow the backward link $\sigma \Rightarrow \vdash$, then the forward link $\vdash \Rightarrow \mu$, and then continue along P. In a similar way, if β is not a deficit, we can tack two more links onto the end of P; we choose any deficit δ and append the links $\beta \Rightarrow \dashv$ and $\dashv \Rightarrow \delta$. The result will be an augmenting path from a surplus to a deficit, so some such path does exist.

If we allocated our Dial array Q large enough, then the building of the forest would halt by discovering such a path. Let δ be the deficit whose discovery would halt the building of the forest. The subsequent round of price increases would raise the price at all remaining surpluses by $\ell(\delta)\varepsilon$. The inflation bound (9-2) would then apply, telling us that $h\Delta \leq (4q+4)s\varepsilon$. But we surely have $\ell(\delta)\varepsilon \leq \Delta$. So we conclude that $\ell(\delta) \leq (4q+4)s/h$. $\square$

9.3 Bounding the prices

The inflation bound also helps us to bound our prices.

Corollary 9-5. *The prices in* FlowAssign *remain $O(qsC)$.*

Proof. Consider the invocation of *Refine* with a particular value for the scaling parameter ε. By how much could any price increase, during this call to *Refine*? During the initialization, we raise the prices by at most $3(q-1)\varepsilon$. Once we enter the main loop, we raise prices only after building each shortest-path forest. From the inflation bound (9-2), we deduce that the total impact of those price increases, throughout this entire call to *Refine*, is at most $(4q+4)s\varepsilon$. Adding these up, we deduce that this call to *Refine* doesn't raise any prices by more than $(4qs + 4s + 3q)\varepsilon = O(qs)\varepsilon$.[*]

On entry to the first call to *Refine*, the prices are zero. And that first call has $\varepsilon = \overline{\varepsilon}/q \leq C$. In subsequent calls, the values of ε decrease in a

[*]Section 10.1 discusses *TightRefine*, a version of *Refine* in which augmenting along an augmenting path involves raising the prices at all of the men along that path by ε. We can allow for those price increases as well by raising the bound to $(4qs + 5s + 3q)\varepsilon$, since only s augmentations happen during *Refine*.

geometric series with ratio $1/q$, whose sum is $O(1)$. Thus, we deduce that all prices are $O(qsC)$. $\square$

Since we store prices and manipulate them as multiples of $\underline{\varepsilon}$ where $1/\underline{\varepsilon} = O(qs)$, we conclude that the integers that we manipulate are $O(q^2s^2C)$.

9.4 Demonstrating square-root performance

Prop 9-6. *The number of iterations of the main loop of* Refine *is* $O(\sqrt{qs})$.

Proof. Note first that every iteration of the main loop of *Refine* reduces h by at least 1. The round of price increases ensures that at least one length-0 augmenting path exists, so we have $|\boldsymbol{\mathcal{P}}| \geq 1$.

Next, we claim that every iteration of the main loop of *Refine*, except perhaps the first, increases Δ by at least ε. Note that Δ could fail to increase in some iteration only if we found a path in R_f from some surplus to some deficit, all of whose links were already of length zero, with no need for any price increases. If such a path A existed in any iteration after the first, however, consider the maximal set $\boldsymbol{\mathcal{P}}$ of compatible augmenting paths that was computed near the end of the preceding iteration. The only changes to the state (f, p) that happen after $\boldsymbol{\mathcal{P}}$ is computed and before A is discovered are the augmentations along the paths in $\boldsymbol{\mathcal{P}}$. But those augmentations affect only the links along those paths, and none of those links can appear in A, since the augmentations leave those links with length 1. So the length-0 augmenting path A must be link-compatible with all of the paths in $\boldsymbol{\mathcal{P}}$, and is hence compatible with them. But the set $\boldsymbol{\mathcal{P}}$ was maximal; so no such path A can exist, and Δ increases in all iterations of the main loop after the first.

Consider the state after $\sqrt{(4q+4)s}$ iterations of the main loop. By this point, since Δ increases in every iteration, we must have $\Delta \geq \sqrt{(4q+4)s}\,\varepsilon$. Applying the inflation bound (9-2), we deduce that $h \leq \sqrt{(4q+4)s}$. Since h decreases in every iteration, we see that the total number of iterations is at most $2\sqrt{(4q+4)s} = 4\sqrt{(q+1)s} = O(\sqrt{qs})$. $\square$

What does this tell us about the overall running time of *Refine*? The building of the shortest-path forest is the most expensive step in the main loop, so each iteration of that main loop takes space and time $O(m + \Lambda)$, where Corollary 9-4 tells us that we can take $\Lambda = O(qs/h)$. Thus, each iteration of the main loop of *Refine* takes space and time $O(m + qs/h)$. We should mention, though, that this bound is only relevant when q is fairly small, say $q < h \log n$. If $q > h \log n$, then we would probably want to abandon the Dial technique and switch over to Fibonacci heaps, which would give us $O(m)$ space and $O(m + s \log n)$ time.

Assuming that we do stick with Dial, however, there will be $O(\sqrt{qs})$ iterations of the main loop of *Refine*, each of which takes space and time $O(m + qs/h)$. The qs/h terms will have the largest sum if h takes on the

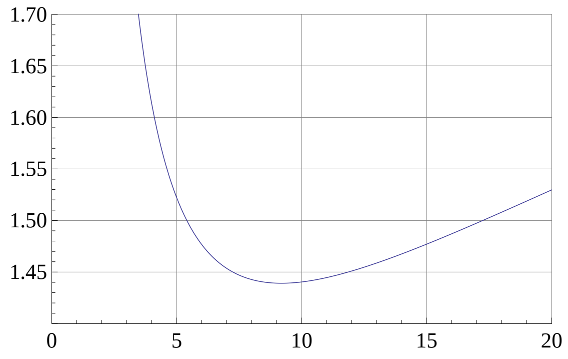

Figure 9.2: The function $\sqrt{q+1}/\ln q$

values 1 through $\sqrt{sz}$ during these iterations, in which case the total time for *Refine* will be

$$O\big((m+qs)+(m+qs/2)+\cdots+(m+qs/\sqrt{qs})\big)=O(m\sqrt{qs}+qs\log(qs)).$$

Recall, from equality (6-3), that the program *FlowAssign* calls the procedure *Refine* $O(\log(sC)/\log q)$ times. Thus, the overall runtime for *FlowAssign* is

$$O\big(m\sqrt{qs}\log(sC)/\log q+qs\log(qs)\log(sC)/\log q\big).$$

Looking just at the first term in this sum, we conclude that we should choose q to be a constant, to avoid being hurt by the $\sqrt{q}$ factor. If we do this, the running time of *FlowAssign* simplifies to $O(m\sqrt{s}\log(sC))$, and the space simplifies to $O(m)$. Note that the $O(m)$ space and the $O(m\sqrt{s})$ time taken by Hopcroft-Karp during the initialization are subsumed by these bounds. So we have finally established the following:

Prop 9-7. *If we take q to be a constant, the algorithm* FlowAssign *solves* ***ImpA*** *in space $O(m)$ and time $O(m\sqrt{s}\log(sC))$.*

In deciding to take $q=O(1)$, we have been manipulating upper bounds on the running time — upper bounds that are likely to be far from tight. So we can't say much about what value for q might lead to the best running time in practice. For whatever it's worth, Figure 9.2 graphs the function $\sqrt{q+1}/\ln q$. The minimum value of about 1.439 happens at about $q \doteq 9.186$, though the function is pretty flat in the neighborhood of that minimum. Since it would probably be convenient to have q be a power of 2, this suggests trying $q=8$ or $q=16$. But experiments on an actual implementation of *FlowAssign* will be needed to find the best value for q in practice.

Chapter 10

Closing remarks

10.1 The variant subroutine *TightRefine*

One way in which *FlowAssign* differs from Gabow-Tarjan involves invariant **I4**, which, you recall, requires that all bipartite, saturated arcs be ε-snug.

Since Gabow-Tarjan works directly on the graph G, rather than on a flow network derived from G, all arcs in Gabow-Tarjan are bipartite. And Gabow-Tarjan keeps all of its saturated arcs, not only ε-snug, but actually ε-tight. Perhaps Gabow and Tarjan did this because they were following the Hungarian Method, which keeps its saturated arcs precisely tight.

Once we move from the graph G to the flow network N_G, it is hopeless to keep all saturated arcs ε-tight. We can and do keep all saturated arcs ε-proper, which puts an upper bound on their net costs. But the net costs of the dummy saturated arcs may get large negative — that seems unavoidable. For the bipartite saturated arcs, however, we can and do impose a lower bound on their net costs. In **I4**, we insist that they be ε-snug. Following Gabow-Tarjan, we could go further and insist that they actually be ε-tight. Let's refer to that stronger invariant as **I4′**.

The main difficulty with maintaining **I4′** crops up when augmenting along an augmenting path. With the executable code of *Refine* as it now stands, the arcs along an augmenting path that change status from idle to saturated end up being ε-snug, but not ε-tight, as shown by the downward arrow in Figure 8.3. The bipartite arcs of this type would blatantly violate **Inv4′**; so something has to change.

Gabow and Tarjan deal with this difficulty by changing the code. In our language, they increase the price of every man along an augmenting path by ε, as part of doing the augmentation. With those price increases, the bipartite arcs that change status from idle to saturated end up being ε-tight, as shown by the downward slanting arrow in Figure 10.1. The good news is that this change to the code succeeds even in our more complicated context of *FlowAssign*, where there are dummy arcs, augmenting paths can visit the source and the sink, and so forth. So we end up with two variants of *FlowAssign*: One uses the version of *Refine* that we have analyzed in this

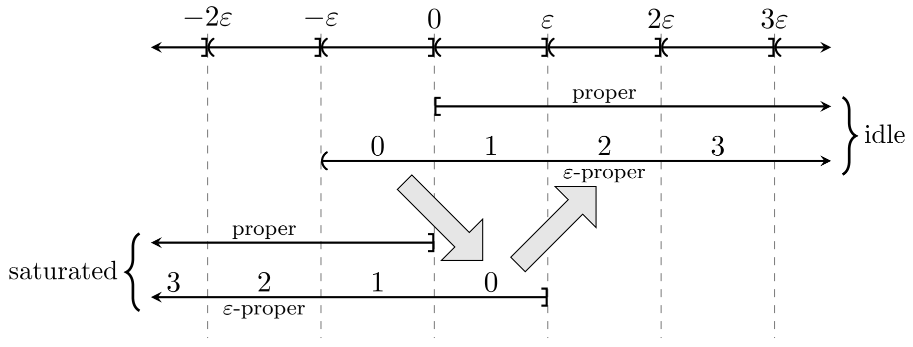

Figure 10.1: Augmentation's effects on the lengths of links in *TightRefine*; contrast these with the effects in *SnugRefine*, shown in Figure 8.3

report, which we now rename *SnugRefine*, while the other uses *TightRefine*, a variant in which augmenting along an augmenting path includes raising the prices of the men on that path by ε.

This change to the code makes *TightRefine* more delicate to analyze than *SnugRefine*. In the main loop of *SnugRefine*, the prices p change only during the price increases after a shortest-path forest is built, while the pseudoflow f changes only during the augmentations. In *TightRefine*, however, the augmentations change p as well as f. More care is then required to verify that the augmentations preserve **I3** and **I5**, and other arguments get more delicate as well. We take some steps, in Appendix C, toward showing that *TightRefine* does the same job as *SnugRefine*, within the same space and time bounds; but we leave the full details of that argument to the reader.

While *TightRefine* is more subtle to analyze than *SnugRefine*, it isn't clear which subroutine would perform better in practice. Experimentation is indicated.

10.2 Quantizing versus nonquantizing

Like the Gabow-Tarjan algorithm, *FlowAssign* achieves its performance by combining weight-scaling with good ideas from the Hungarian Method and from Hopcroft-Karp. From the Hungarian Method, we take the concept of raising prices based on a shortest-path forest. From Hopcroft-Karp, we take the idea of augmenting, not just along one path at a time, but along a maximal set of compatible augmenting paths.

Weight-scaling algorithms come in two flavors. In some of them, such as the ϵ-scaling auction algorithms of Bertsekas [2], each new scaling phase simply reduces the slop allowed in the inequalities that capture complementary slackness. In others, the scaling parameter is used also as a unit of quantization for that scaling phase. That is, all prices during that phase are required to be multiples of the scaling parameter, and all costs and net costs are effectively quantized to be multiples of the scaling parameter as well.

Like Gabow-Tarjan, *FlowAssign* is of this latter, quantizing type. It is because of that quantization that the lengths of the paths in *FlowAssign*'s shortest-path forests are small integers, allowing us to exploit the Dial technique. That quantization is also a key ingredient in allowing us, at the end of *FlowAssign*, to adjust our prices from being ε-proper to being proper simply by rounding. It would be interesting to study whether there is a nonquantizing analog of *FlowAssign*.

Appendix A

Prop 2-8 via LP duality

We here prove Prop 2-8 by applying the standard duality theory of linear programming. In the process, we demonstrate that a primal flow f and dual prices p satisfy complementary slackness just when the pair (f, p) is proper, according to Definition 2-7.

Our primal problem is to minimize the cost of a flow of value s on the flow network N_G. Note that, to remain within the scope of linear programming, we here do not constrain the flow to be integral. We have stated our primal problem as a minimization, while most descriptions of linear programming take the primal problem to be a maximization. So let's instead consider the equivalent problem of maximizing the benefit of a flow f of value s on N_G.

This primal problem has a decision variable $f(v, w)$ for each arc $v \to w$ in the network N_G. The objective function of the primal is to maximize the benefit of the flow f, that benefit being

$$\sum_{v \to w \,\in\, N_G} f(v, w)b(v, w). \tag{A-1}$$

The constraints of the primal problem ensure that the flows $f(v, w)$ along the various arcs fit together to form a flow of value s. So the constraints are:

$$\begin{aligned}
& & f(v,w) &\geq 0 & \text{(A-2)}\\
&h(v,w) & f(v,w) &\leq 1 & \text{(A-3)}\\
&p_d(x) & f(\vdash, x) - \sum_{y\colon x \to y} f(x,y) &= 0 & \text{(A-4)}\\
&p_d(y) & \sum_{x\colon x \to y} f(x,y) - f(y, \dashv) &= 0 & \text{(A-5)}\\
&p_d(\vdash) & 0 - \sum_{x \in X} f(\vdash, x) &= -s & \text{(A-6)}\\
&p_d(\dashv) & \sum_{y \in Y} f(y, \dashv) - 0 &= s & \text{(A-7)}
\end{aligned}$$

Constraints (A-2) and (A-3) ensure that our flows are nonnegative and that they respect the unit capacities of the arcs. Constraints (A-4) through (A-7) ensure that flow is conserved at each node of N_G in turn. Each is here written in the form "flow in − flow out = whatever". For example, (A-4) says that the flow entering a node x in X, which is just $f(\vdash, x)$, minus the total flow leaving x along bipartite arcs should be zero. In (A-6) and (A-7), the right-hand sides reflect the value s of the desired flow.

Constraints (A-6) and (A-7) are actually redundant; either one implies the other, given (A-4) and (A-5). But we leave both (A-6) and (A-7) in the primal problem anyway. If we took one of them out, say removing (A-7), this would effectively force the price at the sink in the dual problem to be zero. The dual problem would still do its job, but we would no longer be free to add any constant to all of the prices, as we'd like to be free to do.

The variables of the dual problem are multipliers on the constraints of the primal; they appear in the left-hand column above. Constraint (A-2) requires that the decision variables of the primal be nonnegative, so it doesn't get a multiplier. There is one instance of (A-3) for each arc in the flow network, and let's write the multiplier associated with the arc $v \to w$ as $h(v, w)$. Because these dual variables multiply inequalities, they will be constrained to be nonnegative. The remaining constraints are equalities, so they get multipliers that are not sign restricted. The multiplier on the flow-conservation equality for some node v turns out to be the dispose price $p_d(v)$. (We get dispose prices because (A-4) through (A-7) are written in the form "flow in − flow out = whatever". If we had instead chosen the equally valid form "flow out − flow in = whatever", we would have ended up with acquire prices as our per-node dual variables.)

We now multiply each of the constraints from (A-3) through (A-7) by the corresponding dual variable and add them all up. Lots of terms drop out of the right-hand side, leaving

$$\sum_{v \to w} h(v, w) - p_d(\vdash)s + p_d(\dashv)s.$$

What about the left-hand side? It will be a linear combination of the flow variables $f(v, w)$. In fact, for any arc $v \to w$, we get $h(v, w)f(v, w)$ from (A-3), we get $-p_d(v)f(v, w)$ from the flow-equality constraint for node v, and we get $+p_d(w)f(v, w)$ from the flow-equality constraint for node w. Note that those latter two claims hold also when $v = \vdash$ or when $w = \dashv$; so these claims hold for dummy arcs, as well as for bipartite arcs. Thus, under our assumption that $h(v, w) \geq 0$ for each arc $v \to w$, the big sum of constraints gives us:

$$\sum_{v \to w} f(v, w)(h(v, w) - p_d(v) + p_d(w)) \leq \sum_{v \to w} h(v, w) - s\big(p_d(\vdash) - p_d(\dashv)\big). \tag{A-8}$$

We want to use (A-8) to get an upper bound on the objective (A-1) of the primal, the objective of the dual then being to minimize that upper bound. So we insist, for every arc $v \to w$ in N_G, that

$$b(v, w) \leq h(v, w) - p_d(v) + p_d(w), \tag{A-9}$$

thus ensuring that the left-hand sum in (A-8) is an upper bound on the objective (A-1) of the primal, so the right-hand sum is also an upper bound. Note that (A-9) can be rewritten as $h(v,w) \geq b(v,w) + p_d(v) - p_d(w) = b_p(v,w)$, using equation (2-4) to compute the net benefit. So the dual variable $h(v,w)$ associated with any arc $v \to w$ must be at least the net benefit of that arc.

So here is the dual problem. It has a decision variable $p_d(v)$ for each node v, not sign-constrained, and a decision variable $h(v,w)$ for each arc $v \to w$, constrained to be nonnegative. The dual problem has one constraint for each arc in N_G, where the constraint for the arc $v \to w$ insists that

$$h(v,w) \geq b(v,w) + p_d(v) - p_d(w) = b_p(v,w). \tag{A-10}$$

The objective of the dual problem is to minimize the quantity

$$\sum_{v \to w} h(v,w) - s(p_d(\vdash) - p_d(\dashv)), \tag{A-11}$$

which, as we have seen, is an upper bound on the value of the primal.

The per-arc dual variables $h(v,w)$ can be removed from the problem easily. The only constraints on them are the sign constraint $h(v,w) \geq 0$ and the arc constraint (A-10). And we want all of the $h(v,w)$ values to be as small as possible, since they are summed in the objective (A-11). So we simply set $h(v,w) := \max(b_p(v,w), 0)$. It follows that the dual problem is always feasible: Whatever values we assign to the per-node variables $p_d(v)$, setting the per-arc variables by the rule $h(v,w) := \max(b_p(v,w), 0)$ guarantees that all constraints will be satisfied.

Let's now suppose that $s \leq \nu(G)$, so that the primal problem is also feasible. Solutions for the primal and dual will both be optimal and will have matching objective values just when complementary slackness holds. There are two parts to complementary slackness:

- If a bounding inequality in the primal has slack, then the corresponding dual multiplier must be zero. So, for any arc $v \to w$ with $f(v,w) < 1$, we must have $h(v,w) = 0$. Recalling that $h(v,w) = \max(b_p(v,w), 0)$, this means that $b_p(v,w) \leq 0$. Rephrasing in the language of Chapter 2, any arc that isn't saturated must have nonpositive net benefit, and hence nonnegative net cost. Taking the contrapositive, any arc with negative net cost must be saturated.

- In the other direction, if a bounding inequality in the dual has slack, then the corresponding primal multiplier must be zero. So suppose that there is slack in the bounding inequality $h(v,w) \geq b_p(v,w)$. Since $h(v,w) = \max(b_p(v,w), 0)$, this is equivalent to supposing that $b_p(v,w) < 0$. Then the primal multiplier $f(v,w)$ must be zero. In the language of Chapter 2, any arc whose net benefit is negative — and whose net cost is thus positive — must be idle.

So complementary slackness is precisely the properness of Definition 2-7.

Appendix B

Slow starts in Hopcroft-Karp

Recall that the algorithm of Hopcroft and Karp [14] builds large matchings in bipartite graphs. This section discusses graphs on which Hopcroft-Karp may, with sufficiently bad luck, experience a slow start.

Let M be a matching in a bipartite graph G. If every augmenting path for M has length at least $2k + 1$, we'll say that M is *k-unaugmentable.* By Corollary 5-3, the size $s := |M|$ of any k-unaugmentable matching M satisfies $s \geq \frac{k}{k+1}\,\nu(G)$. If this lower bound is tight, so that the matching M has as few edges as any k-unaugmentable matching can have, we'll say that the matching M is *k-scrawny.*

Let M_k denote the matching that Hopcroft-Karp has computed after k iterations of its outer loop, and let $s_k := |M_k|$. The matching M_k is always k-unaugmentable. We'll say that a bipartite graph is an *n-stage slow-start graph* for Hopcroft-Karp when, with sufficiently unlucky choices, the matching M_k may be k-scrawny for all $k \leq n$.

The graph G in Figure B.1 is a 3-stage slow-start graph for Hopcroft-Karp. Figure B.1e shows the final matching M_4 that Hopcroft-Karp computes; it is the unique maximum matching in G, of size $s_4 = 12 = \nu(G)$.

- Figure B.1a shows the graph at the start of processing, when the matching $M_0 = \emptyset$ is empty. Any edge is then an augmenting path of the minimum possible length $L = 1$. And the matching M_0 is 0-scrawny, with $s_0 = 0 = \frac{0}{1}(12)$.

- If we are sufficiently unlucky, we may select the six edges that are circled in Figure B.1a to be our maximal set $\mathcal{P}_0$ of vertex-disjoint edges. Note that no other edge is vertex-disjoint from those six. Augmenting along those six edges leads to the 1-scrawny matching M_1 in Figure B.1b, with $s_1 = 6 = \frac{1}{2}(12)$.

- If we are again unlucky, we may select the two augmenting paths of length 3 that are shown circled in Figure B.1b as our set $\mathcal{P}_1$. Note that no other augmenting path of length 3 is vertex-disjoint from those two. Augmenting along those two paths leads to the 2-scrawny matching M_2 in Figure B.1c, with $s_2 = 8 = \frac{2}{3}(12)$.

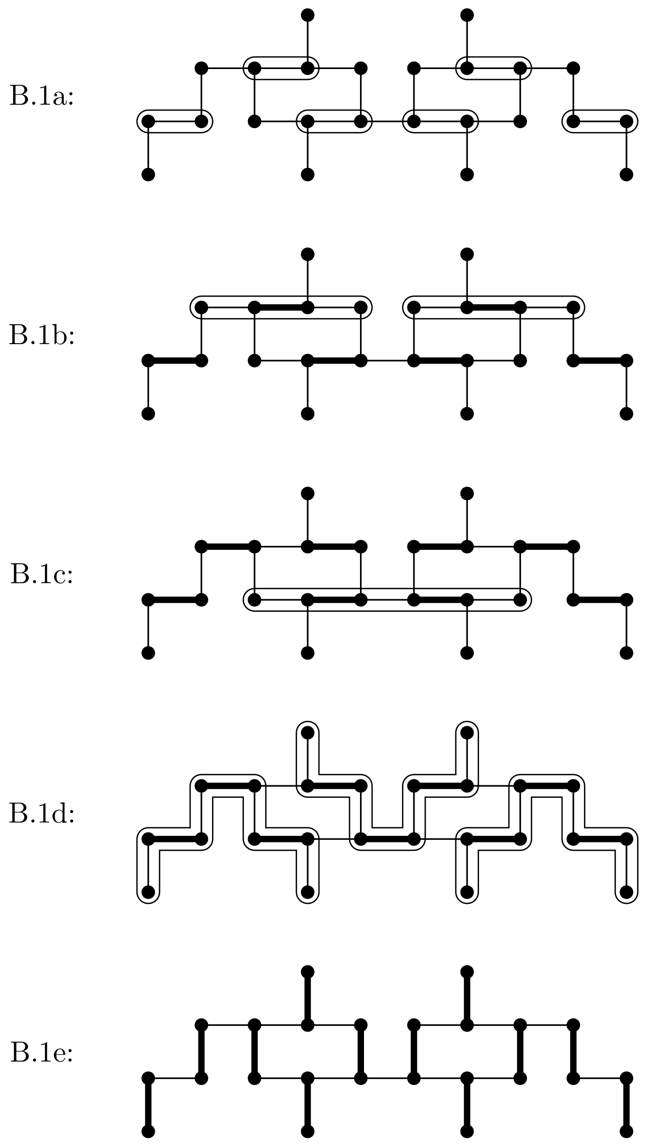

Figure B.1: A 3-stage slow-start graph for Hopcroft-Karp

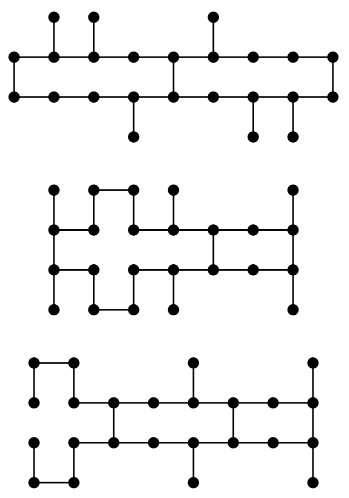

Figure B.2: Other 3-stage slow-start graphs for Hopcroft-Karp

- We may then select the single augmenting path of length 5 that is shown circled in Figure B.1c as $\mathcal{P}_2$. No other augmenting path of length 5 is vertex-disjoint from that path. Augmenting leads to the 3-scrawny matching M_3 in Figure B.1d, with $s_3 = 9 = \frac{3}{4}(12)$.
- At this point, luck stops being an issue. There are precisely three augmenting paths of length 7 in Figure B.1d. Those three paths are vertex-disjoint, so we must choose $\mathcal{P}_3$ to consist of all three of them. Augmenting along them leads to the maximal matching M_4 in Figure B.1e, with $s_4 = 12$. So our matching jumps, in this final iteration, from being scrawny to being maximal.

The graph in Figure B.1 is not unique. Figure B.2 shows three other bipartite graphs, each with 24 vertices and 25 edges, that are also 3-stage slow-start graphs for Hopcroft-Karp. We leave as an open problem whether graphs exist that are n-stage slow-start for $n > 3$. The next graphs to try for would be a 4-stage slow-start graph with 120 vertices and 137 edges and a 5-stage slow-start graph with 120 vertices and 147 edges.

Appendix C

The analysis of *TightRefine*

The bulk of this report deals with *SnugRefine*, the version of the scaling phase of *FlowAssign* in which all bipartite, saturated arcs are kept ε-snug. Section 10.1 discusses the alternative of *TightRefine*, which goes further by keeping its bipartite, saturated arcs ε-tight. In order to do this, *TightRefine* has to raise the prices at all of the men along an augmenting path by ε, as part of augmenting along that path. In this section, we analyze some consequences of those price increases.

C.1 Augmentations preserve the invariants

We first show that an augmentation with those price increases preserves the five invariants of *TightRefine*. **I1′** discusses only the pseudoflow f, not the prices p, and **I2** asserts only that all prices remain multiples of ε; so those invariants are easy. To guide our reasoning about **I3**, **I4′**, and **I5**, consider the example augmenting path of length 13 shown in Figure C.1, connecting the surplus σ to the deficit δ. The left-hand picture shows the path before the augmentation, where the thin lines are ε-tight, idle arcs and the thick lines are ε-tight, saturated arcs. This example path happens to visit both the sink $\dashv$ and the source $\vdash$, in that order. The right-hand picture shows

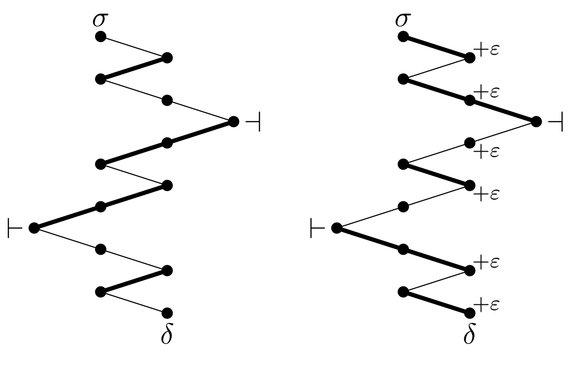

Figure C.1: An example augmenting path in *TightRefine*

Before		After		
state	net cost	state	net cost	ε-tight?
left-dummy idle	$(-\varepsilon\,..\,0]$	left-dummy sat.	$(-\varepsilon\,..\,0]$	no
left-dummy sat.	$(0\,..\,\varepsilon]$	left-dummy idle	$(0\,..\,\varepsilon]$	no
bipartite idle	$(-\varepsilon\,..\,0]$	bipartite sat.	$(0\,..\,\varepsilon]$	yes
bipartite sat.	$(0\,..\,\varepsilon]$	bipartite idle	$(\varepsilon\,..\,2\varepsilon]$	no
right-dummy idle	$(-\varepsilon\,..\,0]$	right-dummy sat.	$(-2\varepsilon\,..\,\varepsilon]$	no
right-dummy sat.	$(0\,..\,\varepsilon]$	right-dummy idle	$(-\varepsilon\,..\,0]$	yes

Table C.1: Arcs on an augmenting path in *TightRefine*

Before		After		
state	net cost	state	net cost	ε-tight?
bipartite idle	$(-\varepsilon\,..\,\infty)$	bipartite idle	$(0\,..\,\infty)$	no
right-dummy sat.	$(-\infty\,..\,\varepsilon]$	right-dummy sat.	$(-\infty\,..\,0]$	no

Table C.2: Arcs that are not on the augmenting path, but that are still affected by the price increases in *TightRefine*

the path after the augmentation. All of the idle arcs have become saturated and vice versa; and the prices at the men on the path have been raised by ε.

Table C.1 summarizes what happens to the arcs on the path as a result of the augmentation (where "sat." is short for "saturated"). All of the arcs on the path start out ε-tight, before the augmentation. So the net cost of an idle arc lies in $(-\varepsilon\,..\,0]$, while that of a saturated arc lies in $(0\,..\,\varepsilon]$. The idle-versus-saturated state of all of the arcs on the path swap during the augmentation. The price increases don't affect the net costs of the left-dummy arcs, but the net costs of the bipartite arcs are raised by ε, while those of the right-dummy arcs are lowered by ε. We can then determine, of the arcs on the path, which end up ε-tight after the augmentation.

The price increases also affect some of the arcs in the residual digraph R_f that aren't on the augmenting path. Table C.2 shows what happens to those arcs. But why is it that Table C.2 has only two rows?

We don't need the two rows for the left-dummy arcs in Table C.2, because a left-dummy arc that isn't on the augmenting path clearly can't be affected by the augmentation.

We claim next that a bipartite saturated arc that is not on the augmenting path can't be affected either. Let $x \to y$ be such an arc. It could be affected only if the node y was on the path, so that the price at y went up by ε. But every man on the augmenting path is the head of at least one bipartite arc on the augmenting path. That arc is saturated either before or after the augmentation. If y was a node on the path, then y would be the head of two distinct bipartite saturated arcs, either before or after the augmentation. This would mean that the man y had a surplus at that time, which would violate $\mathbf{I1'}$. So the node y can't lie on the augmenting path.

We claim also that a right-dummy idle arc that isn't on the augmenting path can't be affected. Let $y \to \dashv$ be such an arc. It is idle both before and after the augmentation, so the node y must not have any incoming flow, both before and after the augmentation. If follows that y cannot lie on the augmenting path.

Table C.2 is thus correct in having only two rows: one for bipartite idle arcs and one for right-dummy saturated arcs. Note that, in both cases, the affected arc won't be ε-tight after the augmentation.

I3 asserts that all arcs remain ε-proper; that is, all idle arcs must have net cost greater than $-\varepsilon$, while all saturated arcs must have net cost at most ε. Now that we have Tables C.1 and C.2, we can check that **I3** is maintained simply by scanning each row in turn.

I4′ asserts that every bipartite, saturated arc must be ε-tight. We can verify that invariant also by scanning the two tables; indeed, only one row in Table C.1 is actually relevant.

I5 asserts that there are no length-0 cycles in the residual digraph R_f. We can assume that there are no length-0 cycles before the augmentation. So any length-0 cycle that comes into existence because of the augmentation must use some link whose status changes during the augmentation. Looking through Tables C.1 and C.2, we see that there are only two types of arcs whose status changes so as to make them newly ε-tight. In both cases, the arc involved lies on the augmenting path. It either starts out bipartite idle, say as $x \Rightarrow y$, and ends up bipartite saturated, as $y \Rightarrow x$; or it starts out right-dummy saturated, say as $\dashv \Rightarrow y$ ends up right-dummy idle, as $y \Rightarrow \dashv$. Thus, all of the new links that appear in the graph R_f^0 have, as their tail, some man y on the augmenting path.

Suppose that there is some cycle in the new graph R_f^0 that traverses one of these links, either $y \Rightarrow x$ or $y \Rightarrow \dashv$. Consider the link that precedes that link along the cycle. It must have the form $v \Rightarrow y$, for some node v. We've just seen that the only arcs along the augmenting path that end up ε-tight, after the augmentation, end up underlying links that have a man as their tail. Such a link can't also have a man as its head, so our assumed link $v \Rightarrow y$ can't lie on the augmenting path. On the other hand, the arc underlying any link $v \Rightarrow y$ that doesn't lie on the augmenting path is certainly affected by the augmentation, since y lies on that path. So the arc underlying $v \Rightarrow y$ must be listed in Table C.2. In both rows of Table C.2, however, the arc ends up not being ε-tight after the augmentation. So no link of the form $v \Rightarrow y$ can exist in the new graph R_f^0, which shows that the augmentation leaves the graph R_f^0 acyclic and **I5** is preserved.

C.2 An augmentation's effect on other paths

Raising prices as part of augmenting along an augmenting path, as we do in *TightRefine*, complicates the issue of how this augmentation affects other actual or potential augmenting paths. The following lemma is useful.

Lemma C-1. *Let B be a length-0 augmenting path in some residual digraph that arises during* TightRefine. *If A is a length-0 augmenting path in that* *same digraph that is compatible with B, then A will remain a length-*0 *augmenting path after we augment along B. In the reverse direction, if A is a length-*0 *augmenting path that exists just after we augment along B, then A was also a length-*0 *augmenting path before we augmented along B, and A and B were then compatible.*

The forward direction of Lemma C-1 ensures that, as we iterate through the paths in the maximal set $\boldsymbol{\mathcal{P}}$, augmenting along each in turn, the earlier augmentations don't destroy the augmenting-path properties of the latter paths. The backward direction of Lemma C-1 is needed when establishing the square-root bound on the running time of *TightRefine*. In Section 9.4, we argued that, in every iteration of the main loop after the first, the round of price increases that follows the construction of the shortest-path forest will have to raise some prices by some nonzero amount. If no price increases were necessary, then the path A from some surplus to some deficit that the shortest-path forest revealed would have been of length 0 already, back at the end of the previous iteration. And we could use the backward direction of Lemma C-1 to contradict the maximality of that iteration's set $\boldsymbol{\mathcal{P}}$.

The *SnugRefine* version of Lemma C-1 is so obvious that we didn't call it out as a lemma. In *SnugRefine*, augmenting along an augmenting path B eliminates the surplus at which B started and the deficit at which B ended. The augmentation also replaces each of the length-0 links along B in the residual digraph with a length-1 link in the reverse direction. But no other surpluses, deficits, or links are affected, and no prices at all are changed. In *TightRefine*, on the other hand, things are more complicated.

Proof. In the forward direction, assume that the path A is compatible with B. By link-compatibility, A starts at a different surplus from B and ends at a different deficit from B. Also, none of the links along A appear along B, so augmenting along B doesn't reverse their direction. By node-compatibility, the path A doesn't visit any of the men that lie along B; so the price increases that are part of augmenting along B in *TightRefine* don't change the lengths of any of the links along A.

For the reverse direction, suppose that A is a length-0 augmenting path that exists just after we augment along B. We argue by induction along A that any woman or man that A visits can't belong to B. Keep in mind, though, that A and B might both visit the source and the sink.

As the base case of the induction, the path A starts at a surplus. Since augmenting along B left every woman along B either married or a nonsurplus maiden, we deduce that the start node of A doesn't lie on B. We now consider, in turn, the six types of links that A might traverse.

- If A backs up along a saturated, left-dummy arc $\vdash \to x$, thus following the backward link $x \Rightarrow \vdash$, it arrives at the source; so there is nothing to prove.

- If A moves forward along an idle, left-dummy arc $\vdash \to x$, thus following the forward link $\vdash \Rightarrow x$, the woman x at whom it arrives must be a nonsurplus maiden. There may be women along B who are now nonsurplus maidens; but Table C.1 shows that the left-dummy arcs that lead to such women, which are now idle, were not left ε-tight by the augmentation. So x can't belong to B.

- The path A might move forward along an idle bipartite arc $x \to y$, thus following the link $x \Rightarrow y$ and arriving at some man y. If y belonged to B, however, we would have raised the price of y by ε as part of the augmentation. Tables C.1 and C.2 tell us that bipartite arcs that are affected by the augmentation and left idle are never left ε-tight. So the man y can't belong to B.

- If A moves backward along a saturated bipartite arc $x \to y$, thus following the link $y \Rightarrow x$, the arc $x \to y$ must belong to the current matching. But every woman who belongs to B and is now married is married to a man who also belongs to B. Since our inductive hypothesis tells us that the man y does not belong to B, the woman x can't belong to B either.

- If A moves forward along an idle, right-dummy arc $y \to \dashv$, following the link $y \Rightarrow \dashv$, it arrives at the sink, so there is nothing to prove.

- Finally, suppose that A backs up along a saturated right-dummy arc $y \to \dashv$, following the link $\dashv \Rightarrow y$. If the man y belonged to B, we would have raised the price at y by ε. But Tables C.1 and C.2 tell us that right-dummy arcs that are affected by the augmentation and left saturated are never left ε-tight. So y can't belong to B.

From this induction, we conclude that the paths A and B don't share any nodes except for perhaps the source and the sink. Thus, the changes to the flow and to the prices that we made while augmenting along B didn't affect A, and the paths A and B must have been compatible, length-0 augmenting paths before we augmented along B. $\square$

Bibliography

[1] Ravindra K. Ahuga, James B. Orlin, Clifford Stein, and Robert E. Tarjan. Improved algorithms for bipartite network flow. *SIAM Journal on Computing* **23** #5 (1994), pp. 906–933.

[2] Dimitri P. Bertsekas. Auction algorithms for network flow problems: A tutorial introduction. *Computational Optimization and Applications* **1** (1992) pp. 7–66.

[3] Dimitri P. Bertsekas and David A. Castañon. A forward/reverse auction algorithm for asymmetric assignment problems. *Computational Optimization and Applications* **1** (1992) pp. 277–297.

[4] Rainer Burkard, Mauro Dell'Amico, and Silvano Martello. *Assignment Problems*, Society for Industrial and Applied Mathematics (SIAM), 2009.

[5] Pages 42–45.

[6] Boris V. Cherkassky, Andrew V. Goldberg, and Craig Silverstein. Buckets, heaps, lists, and monotone priority queues. *ACM-SIAM Symposium on Discrete Algorithms* (1997) pp. 83–92.

[7] Robert B. Dial. Algorithm 360: Shortest path forest with topological ordering. *Communications of the ACM* **12** (1969) pp. 632–633.

[8] Ran Duan and Hsin-Hao Su. A scaling algorithm for maximum weight matching in bipartite graphs. *ACM-SIAM Symposium on Discrete Algorithms* (Jaunary 2012) pp. 1413–1424.

[9] Tomás Feder and Rajeev Motwani. Clique partitions, graph compression and speeding-up algorithms. *Journal of Computer and System Sciences* **51** #2 (1995) pp. 261–272.

[10] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM* **34** #3 (1987) pp. 596–615.

[11] Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing* **18** #5 (1989) pp. 1013–1036.

[12] Andrew V. Goldberg and Robert Kennedy. An efficient cost scaling algorithm for the assignment problem. *Mathematical Programming* **71** (1995) pp. 153–177.

[13] Andrew V. Goldberg and Robert Kennedy. Global price updates help. *SIAM Journal on Discrete Mathematics* **10** #4 (1997) pp. 551–572.

[14] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing* **2** #4 (1973) pp. 225–231.

[15] Ming-Yang Kao, Tak-Wah Lam, Wing-Kin Sung, and Hing-Fung Ting. A decomposition theorem for maximum weight bipartite matchings. *SIAM Journal on Computing* **31** #1 (2001) pp. 18–26.

[16] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics* **2** (1955) pp. 83–97; republished with historical remarks in *Naval Research Logistics* **52** (2005) pp. 6–21.

[17] James B. Orlin and Ravindra K. Ahuja. New scaling algorithms for the assignment and minimum mean cycle problems. *Mathematical Programming* **54** (1992) pp. 41–56.

[18] Lyle Ramshaw and Robert E. Tarjan. A weight-scaling algorithm for min-cost imperfect matchings in bipartite graphs. HP Labs Technical Report HPL-2012-72 (June 2012).

[19] Mikkel Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *Journal of Computer and System Sciences* **69** (2004) pp. 330–353.

[20] Marcos C. Vargas M. *Some algorithms for the assignment problem*, Master thesis, Mathematics Department, Cinvestav-IPN (August 2011). Available by a request to `e_socram@math.cinvestav.edu.mx`.

Index of symbols

$\vdash$	the source node in the flow network N_G.
$\dashv$	the sink node in the flow network N_G.
Λ	in *Refine*, in building a shortest-path forest, a bound on the length of path that will be needed to reach from some surplus to some deficit.
β	some bachelor, that is, a man who is not currently matched.
δ	in *Refine*, some man who is a unit deficit of the current pseudoflow.
ε	in weight-scaling, the slop in the approximate notion of properness.
μ	some maiden, that is, a women who is not currently matched.
$\nu(G)$	the maximum size of a matching in the bipartite graph $G = (X, Y; E)$.
σ	in *Refine*, some woman who is a unit surplus of the current pseudoflow.
C	a real number that satisfies $C \geq \bar{C}$ and $C > 1$.
$\bar{C}$	the maximum magnitude of an edge weight: $\bar{C} := \max_{(x,y)\in G} \lvert c(x,y) \rvert$.
D	in *Refine*, the set of remaining deficits.
E	the set of edges in the bipartite graph $G = (X, Y; E)$.
G	a bipartite graph $G = (X, Y; E)$.
M	a matching in the graph G.
N_G	the flow network built from the graph G.
$\boldsymbol{\mathcal{P}}$	a maximal set of compatible augmenting paths.
Q	an array of pointers to doubly-linked lists for a heap as in Dial [7].
R_f	in *Refine*, the residual digraph of the pseudoflow f.
R_M	in the Hungarian Method, the residual digraph of the matching M.
S	in *Refine*, the set of remaining surpluses.

X	the part of $G = (X, Y; E)$ whose vertices we call women.
Y	the part of $G = (X, Y; E)$ whose vertices we call men.
a	as a subscript, indicates acquire prices.
$b(x, y)$	the benefit of the edge (x, y) in G.
$b(v, w)$	the benefit of the arc $v \to w$ in the flow network N_G.
$c(x, y)$	the cost of the edge (x, y) in G.
$c(v, w)$	the cost of the arc $v \to w$ in the flow network N_G.
$c_p(v, w)$	the net cost of the arc $v \to w$ in N_G, adjusted for the prices p.
d	as a subscript, indicates dispose prices.
e	in *FlowAssign*, the power to which q is raised when setting ε.
f	a flux, pseudoflow, or flow on the network N_G.
$\hat{f}$	the flow component of the pseudoflow f during *Refine*.
h	in *Refine*, the number of surpluses and deficits remaining.
$l_p(v \Rightarrow w)$	the length of the link $v \Rightarrow w$ in the residual digraph R_f.
$\ell(v)$	in building a shortest-path forest, the length of the shortest path yet found to v.
m	the number of edges $m := \lvert E \rvert$ in the bipartite graph $G = (X, Y; E)$.
n	the number of vertices $n := \max(\lvert X \rvert, \lvert Y \rvert)$ in the larger of the two parts of the bipartite graph $G = (X, Y; E)$.
$p_a(v)$	the acquire price at the node v in N_G.
$p_d(v)$	the dispose price at the node v in N_G, where $p_d(v) = -p_a(v)$.
q	in *Refine*, the integer factor by which each scaling phase tightens the approximation to properness.
r	the number of vertices $r := \min(\lvert X \rvert, \lvert Y \rvert)$ in the smaller of the two parts of the bipartite graph $G = (X, Y; E)$.
s	the size of the matching computed by an assignment algorithm.
t	the target size in **ImpA**; the size of the resulting matching is then $s := \min(t, \nu(G))$.
v, w	nodes in the flow network N_G.
x	some woman, that is, an element of X.
y	some man, that is, an element of Y.

Index