Augment or Push? A computational study of Bipartite Matching and Unit Capacity Flow Algorithms

Boris V. Cherkassky
Central Institute for Economics and Mathematics
Krasikova St. 32, 117418, Moscow, Russia
e-mail: cher@cher.msk.su

Andrew V. Goldberg
Paul Martin
NEC Research Institute
4 Independence Way, Princeton, NJ 08540
e-mail: [avg,pmartin]@research.nj.nec.com

João C. Setubal *
Jorge Stolfi †
Institute of Computing, CP6176
University of Campinas, Campinas, SP 13083-970, Brazil
e-mail: [setubal,stolfi]@dcc.unicamp.br

ABSTRACT

We conduct a computational study of unit capacity flow and bipartite matching algorithms. Our goal is to determine which variant of the push-relabel method is most efficient in practice and to compare push-relabel algorithms with augmenting path algorithms. The depth-first search augmenting path algorithm was thought to be a good choice for the bipartite matching problem, but our study shows that it is not robust. For the problems we study, our implementations of the fifo and lowest-level selection push-relabel algorithms have the most robust asymptotic rate of growth and work best overall. Augmenting path algorithms, although not as robust, on some problem classes are faster by a moderate constant factor.

1. Introduction

Maximum unit capacity flow and bipartite matching are two closely related combinatorial problems that have been extensively studied from the point of view of theoretical analysis of algorithms. See *e.g.* [19], [20, 21], [11], [3], [12]. The best bound for the former problem is $O(m^{1.5})$ [11, 20] and for the latter – $O(\sqrt{n}m\log(n^2/m)/\log n)$ [12]. We are aware of three previous computational studies ([8], [5], and [25]) of bipartite matching algorithms and no computational studies of unit capacity flow algorithms. In this paper we study computational efficiency of bipartite matching and unit capacity flow algorithms. Our study includes a refinement of a previous study by the fourth author [26], which itself refined the results in [25] but has not been published in a journal form.

The algorithms we study include augmenting path algorithms, push-relabel algorithms, and an augment-relabel algorithm. The bipartite matching problem is a special case of the unit capacity flow problem, and our codes for the former problem take advantage of its special structure. Otherwise our bipartite matching implementations are very similar to the corresponding unit capacity flow implementations.

*Research was supported in part by CNPq (Brazilian Research Agency) grant number 351056/95-5
†Research was supported in part by CNPq (Brazilian Research Agency) grant number 301016/92-5

The augmenting path algorithms we study find augmenting paths using depth-first search (DFS), breadth-first search (BFS), and a new label-directed depth-first search (LD). Darby-Dowman [8] compared BFS and DFS and found that no implementation performed "consistently better." Chang and McCormick [5] studied different variants of the depth-first search algorithm, implicitly suggesting that this is a good algorithm for the problem. Setubal [25] showed that a push-relabel implementation for bipartite matching was superior to an implementation based on Hopcroft-Karp's algorithm [19].

For the maximum flow problem with arbitrary capacities, several studies have shown that the push-relabel method is superior to other known methods in practice. See *e.g.* [4, 7, 9, 10, 24]. Push-relabel algorithms based on the highest-level selection rule appear to be the fastest. For the unit capacity case, Goldberg and Kennedy [13] show that the lowest-level selection rule, in combination with the global update heuristic, is theoretically efficient. We study the highest-level (HI-PR), lowest-level (LO-PR), and first-in, first-out (FIFO-PR) variants of the push-relabel method. We also study an augment-relabel algorithm (AR) [1] that finds augmenting paths using relabel operations.

Our study includes more algorithms and more problem families than the previous ones and gives new insight into computational performance of the algorithms. The wide variety of problem families was available thanks to some new generators that we created. In terms of code performance we arrived at the following conclusions, which need to be taken in the context of our problem families.

The FIFO-PR and LO-PR codes exhibit the smallest asymptotic growth and are the best or competitive with the best codes for most problem families. One exception is problems specifically designed to be hard for push-relabel algorithms, but such problems are unlikely to come up in practice. Another exception is problems that require a very small number of augmentations, which are very easy for augmenting path algorithms. We also observed that the selection strategies that work best for push-relabel are not the same as in the case of general capacities.

Out of the augmenting path codes, LD is the most robust. There are some problem families where BFS and DFS outperform other codes by a small to moderate constant factor, but on other families these codes lose asymptotically. We also found that DFS had very poor performance on some families compared to BFS. Finally, while the HI-PR and AR codes are competitive on a few problem families, they often perform poorly.

This paper is organized as follows. We define the problems we study in Section 2. Section 3 describes our implementations. Section 4 describes the experimental setup. Section 5 gives experimental results, with interpretation. Section 6 explains the experimental results. Section 7 contains concluding remarks.

2. Definitions

2.1. Unit Capacity Flow

The unit capacity maximum flow problem is defined as follows. Let $G = (V, E)$ be a digraph with special vertices s and t. A *pseudoflow* is a function $f : E \to \{0, 1\}$. For a pseudoflow f and a vertex v, the *excess flow of* v, $e_f(v)$, is defined by $\sum_{(u,v)\in E} f(u, v) - \sum_{(v,w)\in E} f(v, w)$. A *preflow* is a pseudoflow with the property that the excess of every vertex except s is nonnegative. A vertex $v \neq t$ with $e_f(v) > 0$ is called *active*. A *flow* is a pseudoflow f such that, for each vertex $v \in V - \{s, t\}$, $e_f(v) = 0$. A preflow f is a flow if and only if there are no active vertices. The *maximum flow problem* is to find a flow maximizing $e_f(t)$.

Next we define the residual graph. Let $\overline{E}$ be the set of all arcs in E and all their reverses, and let $\overline{G} = (V, \overline{E})$. Given a pseudoflow f, the residual graph of a pseudoflow f is induced in $\overline{G}$ by the arc set $E_f \subseteq \overline{E}$ containing all arcs of E with zero flow and reversals of all arcs of E with unit flow.

2.2. Bipartite Matching

Let $G = (X, Y, E)$ be an undirected bipartite graph, let $n = |X \cup Y|$, and let $m = |E|$. A *matching* in G is a subset of edges $M \subseteq E$ that have no vertex in common. The *cardinality* of the matching is $|M|$. The *bipartite matching problem* is to find a maximum cardinality matching.

The bipartite matching problem can be transformed into a unit capacity flow problem by the following simple transformation (see *e.g.* [23]): add a source s, and arcs from s to each vertex of X; add a sink t and arcs from each vertex of Y to t; and replace each edge (u, v), $u \in X, v \in Y$ with an arc from u to v.

3. Implementations

For the unit capacity flow problem our implementations take as input a graph $G = (V, E)$ in the adjacency list format. For all of our implementations the set of arcs E is sorted by the tail of the arc to increase locality and improve cache performance.

For the bipartite matching problem our implementations take as input a bipartite graph $G = (X, Y, E)$ in adjacency list format, where X and Y are the partitions of the bipartite graph. Then we reduce the problem to a flow problem in the standard way.

The implementations for finding bipartite matchings are refinements of the unit capacity implementations, so we only describe modifications made to take advantage of the additional structure.

3.1. Initialization

We started all implementations for the unit capacity flow problem with the empty flow since there is no obvious way to quickly find a good starting solution.

For the bipartite matching problem, our augmenting path algorithms use the greedy algorithm to find an initial maximal matching. For the push-relabel method we found that using the greedy initial matching never sped up and sometimes slowed down the implementations. All variants of the push-relabel algorithm start with the empty matching.

3.2. Flow Representation

All of our bipartite matching implementations maintain the property that at most one arc out of each vertex of X has flow on it and likewise at most one arc into each vertex of Y carries flow. These arcs with flow define the current matching. Therefore we do not explicitly maintain the flow on each arc but instead for each vertex we maintain a pointer to its current match.

Our flow algorithms maintain residual capacities instead of flows, because the algorithms need the capacities, not the flows, for internal operation.

3.3. Depth-First Search

This implementation uses depth-first search to find augmenting paths. A search is attempted from each of the source's neighbors. It includes the following heuristic (due to Chang and McCormick [5]): If a search fails, all visited vertices in this search are marked as "dead ends" and are not visited in other searches.

For the bipartite matching problem we tested our implementation against that of Chang and McCormick [5], which was written in FORTRAN-77, and concluded that language (or compiler) was not a factor in running time.

3.4. Breadth-First Search

This implementation uses breadth-first search to find augmenting paths. A search is attempted from each of the source's neighbors. Note the following small deviation from strict BFS. If (s, u) and (s, v) exist, as well as (u, v), and if a search from u is attempted first, vertex v will be visited in this search. (In a strict BFS starting from s this would not happen.) Such deviation allows us to use the same heuristic mentioned above for DFS.

3.5. Push-Relabel

We give here a brief description of the push-relabel algorithm (a full description can be found in [17].) The algorithm works by applying the push and relabel operations to active vertices. In our context to be active a vertex must have positive excess flow (as defined before) but it must also have a label below n. A vertex's label is a lower bound on its distance to the sink. A push operation applied to a vertex u consists of moving a single unit of flow to one of its neighbors in the residual graph that has a label with a value one less than its own. A relabel operation occurs when no such neighbor exists and consists of increasing the label of u until such a neighbor exists or until the label reaches n. When there are no active vertices, all that remains is to return excess flow to the source. We do this by finding a succession of paths and cycles along which we can remove flow [16]. (In the case of bipartite matching this is unnecessary.)

We examine three variants of push-relabel based on three selection criteria for the active vertex: highest-level (HI-PR), lowest-level (LO-PR), and first-in, first-out (FIFO-PR). For efficiency HI-PR and LO-PR both require the maintenance of a bucket for each possible distance label, each bucket containing all active vertices at that distance. However for FIFO-PR we can use a simple queue to determine which vertex to scan next.

The more complex data structure used for HI-PR and LO-PR allows us to implement the gap heuristic [6, 9] with little additional effort. The gap heuristic takes advantage of the fact that if there are no vertices with label d then we can delete all vertices with label greater than d since they cannot reach the sink. For FIFO-PR the gap heuristic requires a more expensive data structure which usually outweighs any gain from applying the heuristic, so we do not use it in FIFO-PR.

For all variants of push-relabel we periodically apply the global relabelling heuristic which is known to improve the performance of the push-relabel algorithms for the general maximum flow problem. We apply the global relabelling heuristic after every n relabels. We performed some preliminary experiments which showed that using the global update heuristic greatly improved performance for some input classes while in no case did it significantly degrade performance.

In order to make the algorithms more efficient for the special case of bipartite matching we use the special structure of the problem. We replace the push operation with a *double-push* operation [2, 15] where a unit of excess is pushed first from a vertex in X to a vertex in Y and then is either pushed to the sink or is pushed to a vertex in X. The second portion of the double-push can be performed extremely efficiently since all vertices in Y have only a single arc directed out of them in the residual graph. This is true initially and it is easy to see that by using only double-pushes we maintain this property.

Additionally since each vertex $v \in Y$ has exactly one arc directed out of it, (v, w), the label of v must be the label of w plus one. Therefore we can implicitly represent the labels for all vertices of Y. See [14, 22] for more details.

3.6. Augment-Relabel

The augment-relabel algorithm (see *e.g.* [1]) uses depth-first search in combination with the relabel operation to find augmenting paths. In the context of the bipartite matching problem, Alt, Blum, Melhorn and Paul [3] used a variant of the method in combination with certain word operations to obtain improved bounds for dense graphs. Our implementation of this method, AR, does not use the word operations.

The AR algorithm can be thought of as a cross between the simple search and the push-relabel algorithms, in the following sense: augmenting paths are sought, but distance labels on the vertices are used to determine the directions in which paths are extended. These labels are the same as those in the push-relabel algorithm. If a path cannot advance, a relabel operation is applied, and the current path is shrunk. Periodic global relabeling is also applied after every n relabel operations.

A simple analysis shows that the DFS and BFS algorithms have complexity $O(nm)$. A standard argument shows that the AR algorithm takes $O(nm)$ time. (Note that there are $O(n)$ global updates, each taking $O(m)$ time.)

As observed in [3, 1], one can obtain better theoretical bounds by processing vertices with distance labels bounded by a parameter less then n and using the shortest augmenting path method to complete the computation. However, we were unable to obtain consistent performance gains using this idea, and the implementations we report upon do not switch to the augmenting path algorithm.

For the bipartite matching case, a queue is used to manage the unmatched vertices in X. Every time the global relabeling routine is invoked, it flushes the queue and fills it with the currently unmatched vertices in X with labels below n in increasing order of their labels.

3.7. Label-Directed Depth-First Search

This algorithm uses a relaxation of the distance labels to select an arc to explore next during a depth-first search. The algorithm maintains labels which do not have the distance label properties but are nevertheless estimates on the distances to the sink in the residual graph. In particular, the algorithm periodically sets the labels to exact distances using global updates. After an initial global update, the global updates are performed after every $2n$ relabel operations.

The algorithm builds an augmenting path starting from s. Vertices on the path are marked. Let v be the last vertex on the current path. If $v = t$, the algorithm augments the flow, unmarks all marked vertices, and restarts the search from s. Otherwise, it looks for a vertex w such that (v, w) is a residual arc, w is not on the current path, and $d(w) < d(v)$, and adds w to the path. If no such w exists, the algorithm sets $d(v)$ to be one plus the minimum of $d(w)$ for all unmarked residual neighbors of v and then extends the path as above. If all residual neighbors of v are marked, the algorithm shrinks the path, but does not unmark v.

4. Experimental Setup

The experiments were performed on a 200-MHz Pentium Pro based personal computer running the RedHat 4.0 distribution of Linux with 256 MB of main memory. All implementations of the push-relabel algorithm were written in C++ while the remaining implementations were written in C. The C and C++ code was compiled with the Gnu gcc and g++ compilers respectively with the O4 optimization flag set.

We used three generators to create bipartite matching instances and three generators to create unit capacity flow problems. For most generators several different parameter settings were used creating several families of instances. For all of our bipartite matching instances, $|X| = |Y|$.

4.1. Rand

This generator creates random bipartite graphs by dividing the vertices of each partition, X and Y, into k groups of equal size. For each vertex of the ith group of X the generator chooses y random neighbors from the $(i-1)$th through $(i+1)$th groups of Y (with wraparound), where y is binomially distributed with mean d. The two families we consider are fewg: where there are 32 groups, and manyg: where there are 256 groups; both have average degree 5.

These classes were designed having in mind problems that can be reduced to bipartite matching, such as the maximum vertex-disjoint paths problem. In these problems the resulting graph in the reduction is bipartite, but if the original graph is planar or nearly planar each vertex will only have as neighbors vertices in the surrounding area.

For these classes, the greedy matching paired about 86% of the vertices and maximum matchings paired about 99% of the vertices.

4.2. Hi-Lo Generator

The hi-lo generator of bipartite matching problems separates high and low vertex selection strategies for the push-relabel method. This generator creates a graph with a unique perfect matching and has been motivated by a generator of Kennedy [22].

Let $G = (X \cup Y, E)$ be a graph produced by this generator. This graph is defined by three parameters, ℓ, k, and d. Vertices of X are partitioned into ℓ groups, each containing k vertices. For $1 \leq i \leq k$, $1 \leq j \leq \ell$, we refer to the i-th vertex in group j by x_i^j. Vertices of Y are partitioned similarly, and y_i^j is defined similarly to x_i^j. Each vertex x_i^j is connected to vertices y_p^j for $\max(1, i-d) \leq p \leq i$ and, if $j < \ell$, to vertices y_p^{j+1} for $\max(1, i-d) \leq p \leq i$. There are no other arcs.

The family hi-lo was generating by setting $l = 1$ and $d = 10$. For this class, the greedy matching paired about 96% of the vertices.

4.3. Unibm

This generator was written by the authors, and produces four different classes of bipartite graphs: hexa, rope, grid and zipf. In all our tests the average degree, d was fixed at 6.

In class hexa, the vertices on each side are divided into n/b blocks of size b. One random bipartite hexagon is added between each block i on one side and each of the blocks $i+k$ on the other side, with $|k| \leq K$ for some K. The parameters b and K are chosen by the program in such a way that the average degree is correct (i.e., $3K/b = d$) but few pairs of hexagons have more than one vertex in common. For this class, the greedy matching paired about 83% of the vertices, and the maximum matchings paired about 95% of the vertices.

For the class rope, the vertices on each side are grouped into $t = n/d$ blocks of size d, numbered $X_0 \ldots X_{t-1}$ and $Y_0 \ldots Y_{t-1}$. Block i on one side is connected to block $i+1$ on the other side, for $i = 0, 1, \ldots, t-2$; block X_{t-1} is connected to block Y_{t-1}. Thus, the graph is a "rope" that is folded and twisted over itself, so that it zigzags between the two sides, first up and then down. Consecutive pairs of blocks along the "rope" are connected alternately by perfect matchings ("m-type edges") and random bipartite graphs of average degree $d-1$ ("r-type edges"), beginning and ending with perfect matchings.

The only maximum matching is a perfect one, consisting of all m-type edges. The greedy algorithm tends to use r-type edges, since they are more numerous, and hence to fall short of the maximum; in our tests, it paired about 90% of the vertices.

Each member of class zipf is a random bipartite graph where the edge between u_i and v_j has nominal probability roughly proportional to $1/(ij)$. Thus the graph is denser near the "core" vertices u_0, v_0, and thins out slowly towards the "periphery". In our experiments the greedy matching paired between 62 and 69% of the vertices, while the maximum paired about 5% more vertices.

4.4. Karz Generator

The Karz generator of unit capacity flow problems is inspired by the worst-case problem family developed by Karzanov [20] for a bipartite matching algorithm.

The generator takes parameters k, a, ℓ, F. The vertices of a generated graph are partitioned into several groups: the source s, the sink t, S, X, Y, and P. The source is connected to every vertex in S. Each vertex in S is connected to F randomly chosen vertices in X. Every vertex in X is connected to every vertex in Y. Vertices in P are partitioned into k paths to the sink, with the i-th path containing $(i-1)\ell + 1$ vertices other then the sink. Every vertex in Y is connected to the first vertex on every path.

The family karz was generating by setting $a = k^2, \ell = 10$ and $F = 10$ for several values of the parameter k.

4.5. RMFU Generator

Our RMFU generator is a unit capacity version of the RMF generator due to Goldfarb and Grigoriadis [18]. This generator takes parameters a, b, and c. The graphs that it generates have b "frames" in addition to the source and the sink. Each frame is an a by a grid, with each vertex of the grid connected to the vertices in all four directions, with wrap-around at the edges of the grid. The source is connected to c randomly selected vertices of frame 1. For $1 \leq i < b$, we introduce c

arcs from frame i to frame $i+1$ by selecting c random vertices at each frame and pairing up the selected vertices. Finally, c randomly selected vertices of frame b are connected to the sink. The three variants we consider are: cube (rmfuC), $a = b$, $c = 0.3a^2$; wide (rmfuW), $b = 10$, $c = 0.3a^2$; and long (rmfuL), $a = 10$, $c = 50$.

4.6. Unimf

This program, written by the authors, was used to produce six families of unit capacity flow problems: blow, saus, squa, wave, puff, and dobl. In all cases the graphs had average degree 10.

The basic building block of all these families is an acyclic graph, with a maximum flow consisting of a (s,t)-paths $P_1, P_2, \ldots, P_a$ of length b, which are vertex-disjoint except for s and t. Any arc that is not part of these paths has the tail at an interior vertex of some path P_i, and the head at an interior vertex of some path P_j with $j > i$. These additional arcs are added randomly, with a bias towards arcs that create short s-t paths. Note that the maximum flow, which is unique, must *not* use any of these edges.

The blow family consists of graphs of this form, with $n/4$ paths of length 4. Family saus has 3 paths of length $n/3$, and family squa has $n^{1/2}$ paths of length $n^{1/2}$. Family wave consists of 10 copies of blow connected in series; puff has $n^{1/2}$ copies of blow connected in series; and dobl is derived from a squa by replacing every arc in the unique maximum flow by a path of length 2.

5. Experimental Results

We say that performance of an implementation on a problem family is *good* if it is within a small constant factor (bounded by roughly 3) from the time of the fastest implementation; is *fair* if the performance is within a moderate constant factor (roughly a factor of 8) from the fastest; and *poor* of the implementation is slower by a large constant factor or asymptotically slower. Most bipartite problems in our study are easy, and several algorithms perform well on most of them. Poor performance of an implementation on these problems indicated that the implementation is not robust. Table 1 summarizes results. A technical report version of this paper with tables and plots for each problem family can be accessed via the second author's home page, URL http://www.neci.nj.nec.com/homepages/avg/index.html.

The first-in, first-out and lowest-level implementations: These implementations perform very similarly and we discuss them together. Their performance is good for most problem families. For flows, the only exception is saus, where the performance is fair, and karz, where the performance is poor. (Recall that the karz family is designed to be hard for push-relabel algorithms.) For matchings, the only exception is zipf, where the performance is fair.

The highest-level implementation: This push-relabel implementation does not work nearly as well as the other two. For flows, its performance is poor or fair except for the rmfuW, where the performance is good. For the matching problems, the performance is good except for hi-lo, where the performance is poor.

The augment-relabel implementation: The augment-relabel implementation is consistently worse than the best push-relabel implementations. For flows, it performs poorly most of the time. Its performance is fair on saus and rmfuW. For matchings, the performance is good except for zipf, where it is fair.

The depth-first search implementation: This implementation is worst overall. For flows, it usually performs poorly. Its performance is good only on saus and is fair on rmfuL. For matchings, the performance is poor except for zipf and hi-lo, where it is good.

Table 1: Summary of results. G is good, F is fair, and P is poor.

	FIFO-PR	LO-PR	HI-PR	AR	DFS	BFS	LD
fewg	G	G	G	G	P	P	G
manyg	G	G	G	G	P	P	G
hi-lo	G	G	P	G	G	G	G
grid	G	G	G	G	P	G	G
hexa	G	G	G	G	P	G	G
rope	G	G	F	G	P	G	G
zipf	F	F	F	F	G	G	G
karz	P	P	P	P	G	P	G
rmfuC	G	G	G	P	P	P	G
rmfuL	G	G	F	P	F	P	G
rmfuW	G	G	G	F	P	P	P
blow	G	G	P	P	P	F	P
dobl	G	G	P	F	P	P	F
puff	G	G	P	P	P	P	G
saus	F	F	P	F	G	G	G
squa	G	G	P	P	P	P	P
wave	G	G	P	P	P	P	G

The breadth-first search implementation: Overall, breadth-first search performs significantly better that depth-first search, but it is far less robust than the best push-relabel implementations. Breadth-first search usually performs poorly on the flow problems. The exceptions are fair performance on blow and good performance on saus. For the matching problems, the performance is good except for the fewg and manyg problems, where it is poor.

The label-directed depth-search implementation: The implementation succeeds in making depth-first search more robust. In fact, it is the most robust augmentation-based implementation. For flows, performance of the label-directed search is mixed: fair for dobl, poor for blow, squa, and rmfuW, good for all other families. For matchings, the performance is always good (although in some cases on the borderline with fair).

6. Discussion

Running time of augmenting path algorithms is determined by the number of augmentations and the average fraction of the graph that is examined before an augmenting path is found. In the flow case, the number of augmentations is equal to the maximum flow value. In the matching case, this number is equal to the value of the matching minus the value of the initial greedy matching. Augmenting path algorithms work well when the number of augmentations is very small.

When the number of augmentations is moderate or large, average time to find a path becomes critical. Results of [11] imply that for shortest path augmentation, the total length of augmenting paths is relatively small, especially in the bipartite matching case. If the augmenting paths are always short, breadth-first search tends to examine a small part of the graph for each search and is thus efficient. With a substantial number of long paths, depth-first search is more efficient than breadth-first search. In bipartite matching problems, the augmenting paths tend to be short and breadth-first search outperforms depth-first search. In the flow problems, the paths tend to be longer, and depth-first search performs better relatively to breadth-first search.

Distance labels maintain information from one shortest path computation to another. If the labels are close to actual distances, the relabel-based algorithms work well. For vertices with small labels, the distances are close. While the residual flow is large, the lowest-level and the first-in, first-out algorithms work mostly with low-labeled vertices and thus work well. When the flow is small, the use of global updates assures relatively quick termination. (See [14] for a theoretical analysis of the lowest-level algorithm.) The highest-level and the augment-relabel algorithms do a significant fraction of their work on vertices with high distance labels and do not perform so well.

The label-directed depth-first search algorithm maintains labels to direct the search towards shorter paths. Unless a problem is very easy for depth-first search, extra work of maintaining the labels pays for itself.

7. Concluding Remarks

The first-in, first-out and lowest-level implementations of the push-relabel method are clearly the most robust in our study, especially on the flow problems, where they usually outperform the other codes. For the matching problems, the label-directed search and the breadth-first search implementations are often competitive with the best push-relabel implementations.

We would like to note that all bipartite matching problems we study are easy in the sense that the best algorithms exhibit a rate of growth that is only slightly nonlinear. We leave the problem of generating hard bipartite matching instances open. Since we randomly permute arcs of the input networks, one cannot design problems that are always hard: for example, an initial greedy solution has a nonzero probability of being optimal. One can either design a problem family that is hard on the average or prove that, with the arc list randomization, some algorithm runs in near-linear expected time.

Although pushing is usually better than augmenting, this is not always the case. One clear case when augmenting path algorithms work well is when flow or matching value is very small. In this case breadth-first and depth-first search implementations perform best; the former is more robust. The label-directed search implementation also performs well when the above two implementations do, and is more robust than breadth-first search.

Our study suggests that for an application that is likely to encounter a wide range of network types, the first-in, first-out or the lowest-level push-relabel algorithms should be considered. If the network types are limited, label-directed and breadth-first search algorithms may perform somewhat better.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice-Hall, 1993.

[2] R. K. Ahuja, J. B. Orlin, C. Stein, and R. E. Tarjan. Improved Algorithms for Bipartite Network Flow. *SIAM J. Comput.*, 23:906–933, 1994.

[3] H. Alt, N. Blum, K. Mehlhorn, and M. Paul. Computing a Maximum Cardinality Matching in a Bipartite Graph in time $O(n^{1.5}\sqrt{m/\log n})$. *Information Processing Let.*, 37:237–240, 1991.

[4] R. J. Anderson and J. C. Setubal. Goldberg's Algorithm for the Maximum Flow in Perspective: a Computational Study. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 1–18. AMS, 1993.

[5] S. F. Chang and S. T. McCormick. A Fater Implementation of a Bipartite Cardinality Matching Algorithm. Technical Report 90-MSC-005, Faculty of Commerce and Buisiness Administration, University of British Columbia, 1990.

[6] B. V. Cherkassky. A Fast Algorithm for Computing Maximum Flow in a Network. In A. V. Karzanov, editor, *Collected Papers, Vol. 3: Combinatorial Methods for Flow Problems*, pages 90–96. The Institute for Systems Studies, Moscow, 1979. In Russian. English translation appears in AMS Trans., Vol. 158, pp. 23–30, 1994.

[7] B. V. Cherkassky and A. V. Goldberg. On Implementing Push-Relabel Method for the Maximum Flow Problem. Technical Report STAN-CS-94-1523, Department of Computer Science, Stanford University, 1994.

[8] K. Darby-Dowman. *The Exploration of Sparsity in Large Scale Linear Programming Problems – Data Structures and Restructuring Algorithms.* PhD thesis, Brunel University, 1980.

[9] U. Derigs and W. Meier. Implementing Goldberg's Max-Flow Algorithm — A Computational Investigation. *ZOR — Methods and Models of Operations Research*, 33:383–403, 1989.

[10] U. Derigs and W. Meier. An Evaluation of Algorithmic Refinements and Proper Data-Structures for the Preflow-Push Approach for Maximum Flow. In *ASI Series on Computer and System Sciences*, volume 8, pages 209–223. NATO, 1992.

[11] S. Even and R. E. Tarjan. Network Flow and Testing Graph Connectivity. *SIAM J. Comput.*, 4:507–518, 1975.

[12] T. Feder and R. Motwani. Clique Partitions, Graph Compression and Speeding-up Algorithms. In *Proc. 23rd Annual ACM Symposium on Theory of Computing*, pages 123–133, 1991.

[13] A. V. Goldberg and R. Kennedy. Global Price Updates Help. Technical Report STAN-CS-94-1509, Department of Computer Science, Stanford University, 1994. To appear in SIAM J. on Discrete Math.

[14] A. V. Goldberg and R. Kennedy. An Efficient Cost Scaling Algorithm for the Assignment Problem. *Math. Prog.*, 71:153–178, 1995.

[15] A. V. Goldberg, S. A. Plotkin, and P. M. Vaidya. Sublinear-Time Parallel Algorithms for Matching and Related Problems. *J. Algorithms*, 14:180–213, 1993.

[16] A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. In *Proc. 18th Annual ACM Symposium on Theory of Computing*, pages 136–146, 1986.

[17] A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. *J. Assoc. Comput. Mach.*, 35:921–940, 1988.

[18] D. Goldfarb and M. D. Grigoriadis. A Computational Comparison of the Dinic and Network Simplex Methods for Maximum Flow. *Annals of Oper. Res.*, 13:83–123, 1988.

[19] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ Algorithm for Maximum Matching in Bipartite Graphs. *SIAM J. Comput.*, 2:225–231, 1973.

[20] A. V. Karzanov. O nakhozhdenii maksimal'nogo potoka v setyakh spetsial'nogo vida i nekotorykh prilozheniyakh. In *Matematicheskie Voprosy Upravleniya Proizvodstvom*, volume 5. Moscow State University Press, Moscow, 1973. In Russian; title translation: On Finding Maximum Flows in Network with Special Structure and Some Applications.

[21] A. V. Karzanov. Tochnaya otzenka algoritma nakhojdeniya maksimalnogo potoka, primenennogo k zadache "o predstavitelyakh". In *Problems in Cibernetics*, volume 5, pages 66–70. Nauka, Moscow, 1973. In Russian; title translation: The exact time bound for a maximum flow algorithm applied to the set representatives problem.

[22] R. Kennedy. *Solving Unweighted and Weighted Bipartite Matching Problems.* PhD thesis, Stanford University, 1995.

[23] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Reinhart, and Winston, New York, NY., 1976.

[24] Q. C. Nguyen and V. Venkateswaran. Implementations of Goldberg-Tarjan Maximum Flow Algorithm. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 19–42. AMS, 1993.

[25] J. C. Setubal. New experimental results for bipartite matching. In *Proceedings of* NETFLOW93, pages 211–216. TR–21/93, Dipartimento di Informatica, Universitá di Pisa, 1993.

[26] J. C. Setubal. Sequential and Parallel Experimental Results with Bipartite Matching Algorithms. Technical Report IC-96-09, Institute of Computing, University of Campinas, Brasil, 1996.