An Implementation Study of Flow Algorithms in

Unit Capacity, Undirected Networks

by

Dennis S. Ruhl

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

September 3, 1999

Author_____
Department of Electrical Engineering and Computer Science
September 3, 1999

Certified by _____
David Karger
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

An Implementation Study of Flow Algorithms in
Unit Capacity, Undirected Networks
by
Dennis S. Ruhl

Submitted to the
Department of Electrical Engineering and Computer Science

September 3, 1999

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Within the extensive literature of maximum flow, there have recently been several exciting new algorithms developed for unit capacity, undirected networks. In this paper we implement five of these algorithms, from Even and Tarjan [1] and Karzanov [2], Karger and Levine [3], Goldberg and Rao [4], and Karger [5] with an eye towards comparing their efficiency in practice and suggesting possible practical and/or theoretical improvements. We also evaluate the performance of directed graph algorithms implemented in Cherkassky, Goldberg, Martin, Setubal, and Stolfi [6] on undirected graphs, using a simple doubled-edge directed graph representation for undirected graphs.

Thesis Supervisor: David Karger
Title: Associate Professor of Electrical Engineering and Computer Science

Contents

Chapter 1

Introduction

A *maximum flow* in an undirected graph is a routing of flow along the graph's edges that satisfies *skew symmetry, capacity,* and *flow conservation* constraints, and results in the largest possible amount of flow being transported from the *source* to the *sink*. The *skew symmetry* constraint requires that the sum of the flows in both directions along an edge be equal to zero. The *capacity* constraint dictates that the flow along an edge cannot exceed that edge's capacity. The *flow conservation* constraint prescribes that the flow into a node must equal the flow out of that node for all nodes except the *source*, where the outward flow can exceed the inward flow, and the *sink*, where the inward flow can exceed the outward flow. These concepts can be understood more readily from a diagram; see Figure 1.1 below.

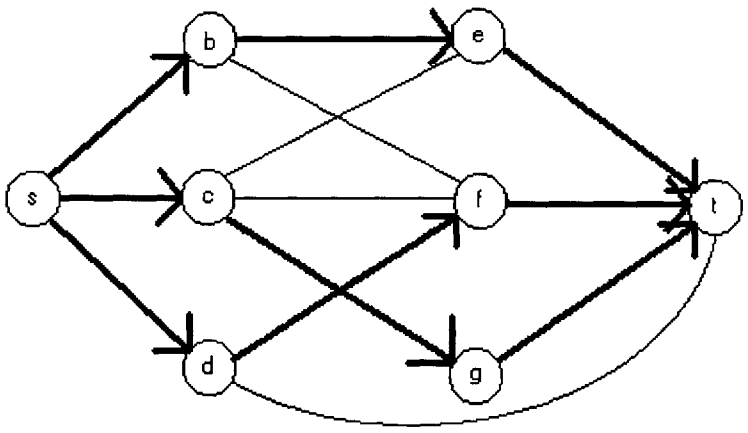


Figure 1.1: A maximum flow with value 3. Node *s* is the source, node *t* is the sink.

In this thesis, we examine a special case of this problem, namely maximum flow in *unit-capacity*, undirected graphs. A *unit-capacity* graph is a graph in which the capacity of each edge in the graph is one. Several different algorithms for this problem have been proposed, with theoretical bounds that are often rather close to each other. We implement the algorithms and examine which of them is best in practice.

1.1 Previous Work

In this section we examine the research literature that influences this thesis. We first examine the theoretical results and then review some experimental papers.

1.1.1 Theoretical Work

The history of study of algorithms for maximum flow in networks began largely in 1956 with the seminal paper "Maximal Flow Through a Network" by Ford and Fulkerson [7], in which a deterministic $O(mv)$ (where v is the value of the maximum flow) time algorithm was introduced for determining the maximum flow in a directed graph with capacities.

This paper started a flood of research into maximum flow algorithms. As the research literature expanded, study began of special cases of the general maximum flow problem. Among these cases is maximum flow on an undirected, unit capacity network. Such a network has no direction on the edges (note that undirected networks can be simulated, with only constant overhead, by directed networks) and a capacity of one on each edge. Within this special case, we will consider two subcases: networks without parallel edges and networks with parallel edges. In a network that contains no parallel edges, the value of a maximum flow on the network can exceed neither m nor n. In this section, all time bounds and diagrams of time bounds are for networks with no parallel edges. We will refer to a unit capacity graph with no parallel edges as a *simple graph.*

For unit capacity networks (directed or undirected), papers by Karzanov [2] and Even and Tarjan [1] showed that a method called *blocking flows* (invented by Dinic [8]) solves the maximum flow problem in $O(m \min\{n^{2/3}, m^{1/2}, v\})$ time.

In 1994, Karger [5] published an important paper making a randomized entry into this area by contributing a Las Vegas (probabilistic running time) algorithm that ran in $\tilde{\mathbf{O}}(mv/c^{1/2}\})$ time, where all cuts have value at least c. Karger's next paper, published in 1997 [9], included a Las Vegas algorithm that ran in $\tilde{\mathbf{O}}(m^{2/3}n^{1/3}v)$ time as well as a Las Vegas algorithm that ran in $\tilde{\mathbf{O}}(m^{5/6}n^{1/3}v^{2/3})$ time. These time bounds are equivalent when $v = m^{1/2}$, with the former algorithm dominating for lesser v and the latter algorithm dominating for greater v. Unfortunately, of these three algorithms, the latter two (as well as several other Las Vegas algorithms that followed in 1998 [3, 10]) were very complicated and represented only a small performance improvement on the earlier, simpler algorithms. Consequently, only the first algorithm is simple enough to be implementable within the scope of this thesis.

Later in 1997, Goldberg and Rao [4] published a paper that utilized the *graph sparsification* algorithm of Nagamochi and Ibaraki [11]. This algorithm allows, in linear time, extraction of the "important" edges in a graph, thus quickly producing a smaller graph that has similar properties. Using it as a black box, Goldberg and Rao were able to produce a simple, deterministic $O(m^{1/2}n^{3/2})$ time algorithm. This algorithm is an improvement on the blocking flow technique for relatively large values of m and v (to be precise, $m \geq n^{5/3}$ **and** $n \leq m^{1/3} v^{2/3}$). For all other values, Goldberg and Rao's algorithm runs in equivalent time to the blocking flow technique of Dinic [1, 2]. These time bounds are summarized in Figure 1.2 below, in which a point (a, b) represents a network where $v = O(n^a)$ and $m = O(n^b)$ and the color of a point represents the algorithm that will (theoretically) run best on networks with those parameters. The maximum values on the two axes are due to constraints on the possible range of values for these parameters in undirected simple graphs. Also, graphs with fewer edges than nodes are excluded from the figure, as the unconnected nodes can be discarded quickly.

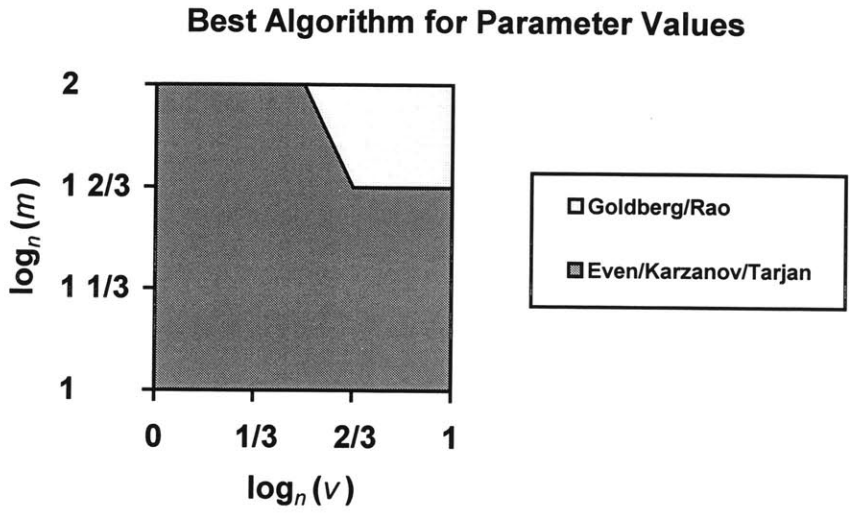


Figure 1.2 Plot showing how different values for m and v influence which algorithm is best in theory.

Finally, in 1998, Karger and Levine [3], using *graph sparsification* techniques like Goldberg and Rao [4] did, published a pair of deterministic algorithms that run in $O(m + nv^{3/2})$ time and $O(nm^{2/3}v^{1/6})$ time. These algorithms are still simple enough to be considered practical. Furthermore, their theoretical running times are at least as good as Goldberg-Rao for all values of m and v and also at least as good as Even-Karzanov-Tarjan for most values of m and v (to be precise, for $m \geq nv^{1/2}$). This information is summarized in Figure 1.3 below. It should be read in the same manner as Figure 1.2. Here "Karger-Levine 1" is the $O(m + nv^{3/2})$ time algorithm and "Karger-Levine 2" is the $O(nm^{2/3}v^{1/6})$ time algorithm.

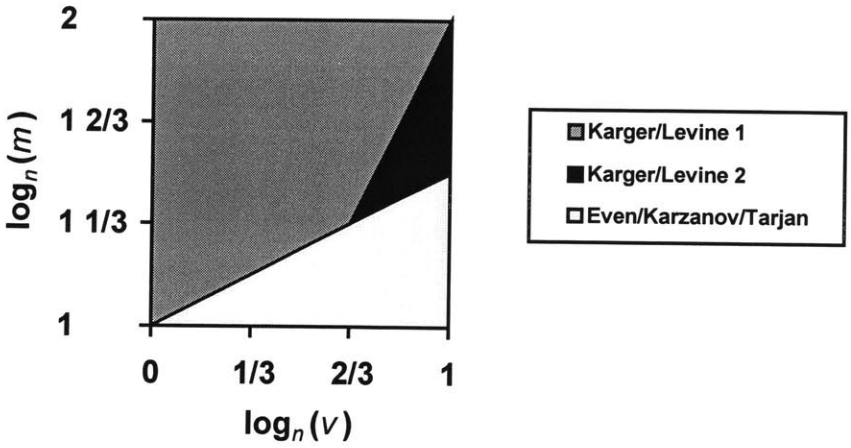


Figure 1.3 Best algorithms, in theory, for certain parameter values. The Goldberg-Rao algorithm is always slower.

The current theoretical knowledge of algorithms for the maximum flow problem on undirected simple networks reveals that the problem lacks a "natural" time bound and has been attacked with a wide variety of algorithms that are theoretically best for different parameter values. These factors strongly suggest that theoretical improvements are possible.

1.1.2 Experimental Work

Besides the theoretical work, there is an extensive body of literature of implementation studies. Two are particularly relevant to this thesis because they involve implementations of similar algorithms.

The first is an implementation study of unit capacity flow and bipartite matching algorithms by Cherkassky, Goldberg, Martin, Setubal, and Stolfi [6]. This study looks at the directed case, with parallel arcs, so it is substantially different than this thesis, but it does provide some guidance. In particular, it proposes problem families and performance measuring guidelines that can be usefully extended to the undirected case. Also, it provides several excellent augmenting paths codes that can be used with minimal adaptation in some of the algorithms studied in this paper. The augmenting paths codes are especially useful, as it makes it

possible to easily test several well-written, heuristically optimized augmenting paths codes in each algorithm.

The second relevant implementation study is by Anderson and Setubal [13] from the First DIMACS Implementation Challenge. Although it is dated, it does provide implementations of Dinic's algorithm and blocking flows on which ours are based.

1.2 Our Contribution

The first contribution of this thesis is to test five of the major algorithms discussed in the background section (from Even/Karzanov/Tarjan, Goldberg/Rao, the two latest algorithms from Karger/Levine, and the first randomized algorithm by Karger) and obtain experimental data on their performance on various networks. This experimental data includes graphs depicting how run times vary in practice with values for m and v. Results also include analysis showing how the algorithms' performance scales with input size, and which algorithms are best for different families of graphs with certain properties. These families were chosen to provide a mixture of graphs that seem relatively "natural" and graphs that are designed specifically to attempt to incur the worst-case behavior of certain algorithms.

The second contribution of this thesis is to determine whether algorithms specifically designed to handle undirected graphs perform better than directed graph algorithms processing the undirected graphs as edge-doubled directed graphs. To this end, the algorithms and code from Cherkassky, Goldberg, Martin, Setubal, and Stolfi [6] were tested alongside the algorithms implemented for this thesis.

A third main contribution of this thesis is to form a preliminary assessment of whether algorithms that use Nagamochi and Ibaraki and/or randomization to sparsify graphs are practical.

Chapter 2

Background

In this chapter we discuss the theory behind maximum flow algorithms. We will begin by reviewing a few algorithms that many of the algorithms tested in this paper use as subroutines, namely algorithms for augmenting paths, blocking flows and graph contraction. Then we will outline the theory behind each of the major algorithms tested in this paper. First, however, we will introduce some terminology for the maximum flow problem.

Let $G = (V, E)$ be an undirected graph with vertex set V edge set and edge set E. We use n to denote the cardinality of V and m for the cardinality of E. An undirected edge between vertex v and vertex w in this graph will be denoted by $\{v, w\}$; a similar directed edge will be referred to as (v, w), where v is the beginning, or *tail*, of the edge and w is the *head*. The set of all edges that lead outward from a vertex v is denoted by $\vec{E}(v)$; the set of all edges that lead inward to a vertex v is denoted by $\overleftarrow{E}(v)$. $E(v)$ is the union of those two sets (equivalently, the set of all undirected edges with v as an endpoint). The flow along an edge e will be referred to as $f(e)$. The *source* of the graph is conventionally referred to as s; the *sink* as t. The value v of a maximum flow in a graph is defined by

$$f(G) \equiv \sum_{\forall e \in \vec{E}(s)} f(e) = \sum_{\forall e \in \overleftarrow{E}(t)} f(e)$$

The second equality follows from the flow conservation constraint, which states that flow must be conserved at all vertices besides s and t. Note that while there may be more than one possible maximum flow in a graph, all maximum flows have the same value, and thus our algorithms must only find one maximum flow (which we refer to as "the maximum flow.")

The *residual graph*, G_f, is defined by $G_f = (V, E_f)$ where E_f contains a pair of directed edges in opposite directions each with capacity one for each undirected edge in E that carries zero flow as well as, for each undirected edge that is carrying flow, a pair of directed (since we can send one unit to reverse the unit being sent and another to consume the edge in the other direction) edges each with capacity one in the opposite direction of the flow. Thus, although G is an undirected graph, its residual graph G_f is directed. The *residual flow* is the flow in the residual graph. An *augmenting path* is a path from s to t along edges in the residual graph. A *blocking flow* is a set of augmenting paths in a residual graph such that every path from s to t contains a used edge. A blocking flow is allowed, however, to create new paths from the source to the sink because of residual edges that are added to the residual graph when the flow in the graph is increased by the blocking flow.

2.1 Basic Subroutines

Many of the new algorithms examined in this paper achieve their speedy time bounds not by fundamentally changing the algorithms that are used to route flow, but rather by reducing the number of edges that are active in the graph at the time they run and then finding and adding flow until no more can be added. As a result, both our two comparatively basic algorithms (Dinic's and the directed algorithms) and the four algorithms that take advantage of the properties of undirected graphs share two basic classes of flow-finding algorithms, namely augmenting paths and blocking flows. Furthermore, of the four more complex algorithms, three of them rely on the same graph sparsification technique. We will begin our discussion of the theory behind these algorithms by examining the basic subroutines they use: augmenting paths, blocking flows, and graph sparsification, so we do not need to cover them more than once in the sections on the main algorithms.

2.1.1 Augmenting Paths

Algorithms using augmenting paths have a long history within the maximum flow problem, dating all the way back to the original maximum flow algorithm (the Ford-Fulkerson method). Although the basic concept of an augmenting path as a source-to-sink path in the graph along which it is possible to send flow remains unchanged, many different ways of finding augmenting paths in a residual graph have been invented. In this paper, we used four different methods: breadth-first search, depth first-search, label-directed search, and augment-relabel. Although all of these methods have the same theoretical $O(m)$ time per path found, their actual running times can vary greatly on different graphs. Although these different approaches could be considered to be merely be different heuristics for augmenting paths, we discuss them here as they are very different and each one incorporates a significant number of heuristics unique to itself.

Breadth-first search (BFS) performs a breadth-first search in the residual graph to the sink until it finds a path and returns this path as the augmenting path. There are two simple heuristics used in the implementation of this algorithm that will be discussed in the implementation section.

Depth-first search (DFS) performs a depth-first search in the residual graph to the sink until it finds a path and returns this path as the augmenting path. The implementation used in this thesis involves three straightforward heuristics that will be explained in the implementation section.

Label-directed search (LDS) [6] is an enhanced version of depth-first search, although one with the same theoretical bounds. Briefly, the algorithm employs vertex labels that are estimates of the distances to the sink in the residual graph (similar to distance labels in push-relabel algorithms). We denote the vertex label of a vertex v by $L(v)$. The algorithm begins building an augmenting path from the source, marking vertices on the path as it proceeds. Assume that the algorithm has built a path ending at a vertex v. To proceed, it first checks if $v = t$; if so it unmarks all nodes on the path and returns the path as an augmenting path. If not, it looks for a vertex w such that (v, w) is an edge in the residual graph, w is not marked, and $L(w) < L(v)$. If such a w exists it is added to the path and the algorithm continues. If not, the algorithm

sets $L(v)$ to one greater than the minimum of the vertex labels of all unmarked neighbors of v (if there are none it does not change the label) and proceeds to unmarked neighbors with a minimal vertex label. If v has no unmarked neighbors, the algorithm removes v from the path and continues the search from the previous vertex on the path. The algorithm does not unmark v in this case. The implementation used in this thesis employs one fairly complex heuristic that will be explained in the implementation section.

Augment-relabel (AR) [14] is an attempt to combine the techniques used by DFS and by push-relabel algorithms (which will be discussed later). Theoretical improvements including using word operations [15, 16] are not used in this implementation. Other theoretical improvements [14, 15] involving the use of shortest augmenting paths at the end of the maximum flow computation were tried [6] and not found useful; as a result, they are not employed here. Augment-relabel is essentially the same as the LDS algorithm with two exceptions. First, vertices are not marked and unmarked. This does not allow cycles, however, as the distance label of the last node reached to make the cycle would be higher then the current distance label, so the path would not be extended to complete the cycle. Second, when the algorithm is forced to change the vertex label of a vertex, instead of continuing from one of the neighbors of the relabeled vertex, it discards the vertex from the path and continues the search from the previous vertex on the path. This algorithm also uses a fairly complex heuristic that will be explained in the implementation section.

2.1.2 Blocking Flows

Algorithms involving blocking flows have almost as extensive a part in the history of maximum flow algorithms as those involving augmenting paths. Blocking flows were invented as a result of the observation that, in the same $O(m)$ time it takes to find an augmenting path, many augmenting paths could be found. Blocking flows are required to increase the length of the shortest augmenting path in the graph; consequently, the concept of a *layered graph* is necessary. A layered graph, $L(G_f)$, is defined as the graph that contains all the edges in all the shortest paths from the source to the sink in the residual graph G_f. The blocking flow techniques find blocking

flows in the layered graph of the residual graph. We chose to use Dinic's algorithm [8] as explained by Even and Tarjan [1] to compute blocking flows. It has a theoretical running time of $O(m)$ for a blocking flow.

Dinic's algorithm proceeds in two phases. In the first phase, it performs a breadth-first search from the source, labeling each node as it is scanned with its distance from the source. This scan terminates when the sink has been labeled and the node to be scanned has a greater distance from the source than the sink. In the second phase, the algorithm performs a series of depth first searches from the source to the sink, using only edges that were scanned during the breadth-first search. At each vertex, the depth first search attempts to proceed to a vertex with a higher distance label until it reaches the sink. After a search reaches the sink, all the edges on the current path are removed from the set of edges being searched, and the search continues with the next edge that leaves the source. If at any point we fail to reach the sink and need to retreat, we remove the edges that we retreat along from the graph, so we have an $O(m)$ time bound. The actual implementation differs from this explanation in several ways that will be explored in the implementation section.

2.1.3 Graph Sparsification

The quick theoretical time bounds that many of the algorithms implemented achieve are primarily due to their use of graph sparsification techniques. These techniques remove edges from the graph, without changing the value of the flow, and thus allow us to speed up our algorithms when the input graphs are sufficiently dense. The procedure FOREST [11] is the sparsification technique of choice of all the algorithms that use sparsification but one (which effectively uses randomization to perform sparsification). FOREST takes as input an undirected graph and labels each edge in the graph with a number. Each number then identifies a certain set of edges. Each set of edges is a maximal spanning forest in the graph formed by removing the edges in lower-numbered spanning forests from the original graph. Impressively, this algorithm runs in time linear in the number of edges. The correctness arguments for this algorithm are rather involved, and will not be described here (see Nagamochi and Ibaraki [11] or Levine [17]).

When the algorithm is finished, each edge e is identified as belonging to the $r(e)^{th}$ maximal spanning forest. Below is pseudocode for FOREST adapted from Nagamochi and Ibaraki [11].

```
                    FOREST(V, E)
1       FOR all v ε V
2           scanned(v) = FALSE
3           r(v) = 0
4       FOR all e ε E
5           scanned(e) = FALSE
6           r(e) = 0
7       WHILE there exists a vertex v with scanned(v) = FALSE
8           V' = {v such that scanned(v) = FALSE}
9           x = v such that v ∈ V' and r(v) = max{r(n), n ∈ V'}
10          E'(x) = {e ε E(v) such that scanned(e) = FALSE}
11          FOR all e = {x, y} ε E'(x)
12              r(e) = r(y) + 1
13              r(y) = r(y) + 1
14              scanned(e) = TRUE
15          scanned(x) = TRUE
```

To further help explain the FOREST algorithm, Figures 2.1 and 2.2 below show a graph before and after FOREST. This example is also found in Nagamochi and Ibaraki [11].

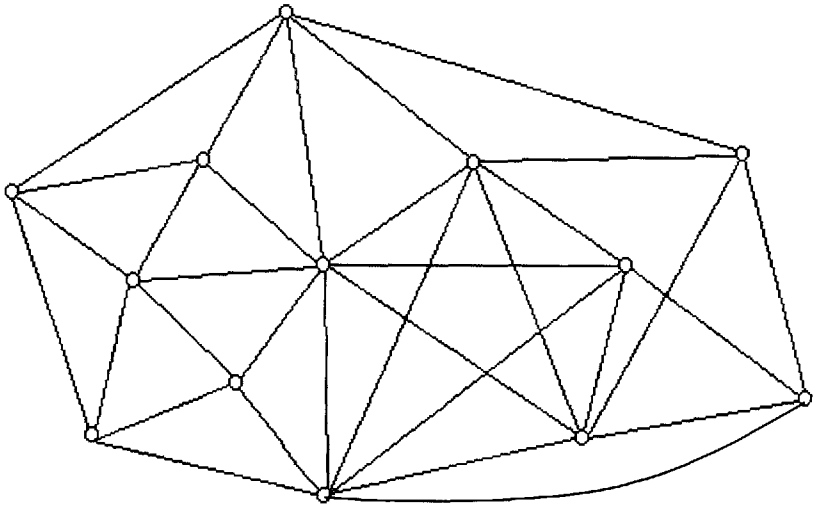


Figure 2.1 A graph G

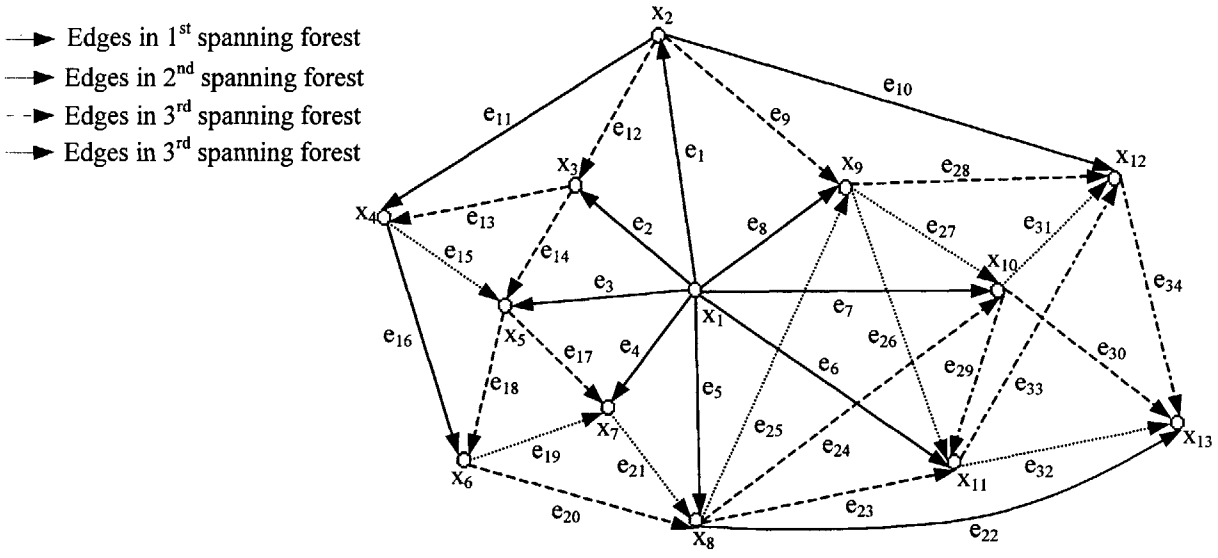


Figure 2.2 G after FOREST. Edge and vertex numbers indicate the order they were processed in.

2.2 Algorithms

In this section we examine the theory behind the actual algorithms we used to find the maximum flows. We look first at a series of directed approaches including push-relabel and Dinic's blocking flow algorithm, then examine three approaches using Nagamochi and Ibaraki sparsification with augmenting paths and/or blocking flows, and finally examine a randomized algorithm. Since we have already examined the augmenting paths routines, we do not discuss them here, except to note that all of the augmenting paths algorithms run an augmenting paths routine over and over until no more flow can be found, thus achieving $O(mv)$ time bounds.

2.2.1 Push-Relabel

To understand push-relabel algorithms, we first need to define a few new terms. A *preflow* is much like a flow, except it follows a looser version of the flow conservation constraint. Instead of requiring that the net flow at all vertices besides the source and sink be 0, in a preflow the net flow at each vertex besides the source and sink must be non-negative. Thus, any flow is also a preflow. The *excess flow* of a vertex, denoted by $e(v)$, is simply the amount by which a given node violates the flow conservation constraint. More formally,

$$e(v) \equiv \sum_{\forall e=(u,v)\in E} f(e) - \sum_{\forall e\in E(v)} f(e)$$

A vertex v is said to be *overflowing* if $e(v) > 0$. A distance function is a set of integer labels for vertices where the sink is labeled 0, and if an edge (u, v) exists in the residual graph, the label of u is less than or equal to the label of v plus one. We refer to the label of a vertex v as $d(v)$. An *active* vertex is an overflowing vertex with a low enough distance label (exactly what is low enough depends on the details of the implementation; here it is with a distance $\leq n$).

The algorithm consists of two basic operations, push and relabel, which are used to convert a preflow into a maximum flow. The push operation can be applied to an edge (u, v) in the residual graph if u is overflowing and $d(u) = d(v) + 1$. Under these conditions, it pushes one

unit of flow along the edge (u, v), which saturates the edge, thus removing it from the residual graph, reduces the excess of u by one, and increases the excess of v by 1. The relabel operation can be applied to a vertex u when u is overflowing, and for all edges (u, v) in the residual graph, $d(u) < d(v) + 1$. The relabel operation increases $d(u)$ to be one greater than the minimum of the $d(v)$ values for all edges (u, v) in the residual graph.

The push-relabel algorithm, then, simply executes push and relabel operations until none are possible or the vertices all have labels that are high enough to conclude that all excess is either at the source or sink and the preflow, which will then be a flow, is maximal. The level the vertices need to reach to be considered "high enough" depends on the particulars of the implementation and will be discussed in the implementation section; for now it is enough to say that for any reasonable implementation it will be $O(n)$. The running time of this algorithm can be bounded fairly simply by $O(mn)$, but with certain heuristics an $O(m \cdot \min\{m^{1/2}, n^{2/3}\})$ bound can be proven. The details of the implementation and heuristics used affect the details of these proofs; therefore we will not discuss them now but will outline them in the implementation section. For a more detailed explanation of this algorithm, as well as detailed analysis of the correctness and running time arguments, and an attempt to provide the intuition behind the algorithm, consult Ahuja, Magnanti, and Orlin [14] or Cormen, Leiserson and Rivest [18].

2.2.2 Dinic's Algorithm

Dinic's algorithm is a simple extension of Dinic's method of finding blocking flows, namely finding blocking flows over and over until the flow is maximal. Fortunately, however, better time bounds can be proven then the naïve $O(mv)$ (based on v blocking flows each of which take $O(m)$ time). In fact, for simple graphs, Even and Tarjan [1] and Karzanov [2] proved that Dinic's algorithm runs in $O(m \cdot \min\{n^{2/3}, m^{1/2}\})$ time.

Theorem 2.2.2.1 *[1] In a directed graph with no flow on any edge, the maximum distance from the source to the sink is less than m/v.*

Proof. Let V_i be the set of vertices at a distance i from the source and let l be the length of the shortest augment path. Then, for all $0 \leq i < l$, the number of edges between V_i and V_{i+1} is at least v, or else we would have a cut with value smaller than v so we could not have a flow value of v. But $l \cdot v \leq m$, so $l \leq m/v$. ■

Theorem 2.2.2.2 *[1] In an undirected, unit-capacity graph with no parallel edges Dinic's algorithm runs in $O(m^{3/2})$ time.*

Proof. If $v \leq m^{1/2}$, the $O(m^{3/2})$ time follows immediately. Otherwise, consider the algorithm immediately before the first blocking flow that raises the flow value of the graph above $v - m^{1/2}$. The residual graph before this blocking flow is a graph with no more than two directed edges between any pair of vertices and no flow on any edge, so the maximum distance from the source to the sink in the residual graph is bounded by $m/\sqrt{m} = m^{1/2}$ by theorem 2.2.2.2. So, it will take $O(m^{1/2})$ blocking flows to reach the point where the flow value is just below $v - m^{1/2}$ because each blocking flow increases the length of the shortest augmenting path by one. It will also take $O(m^{1/2})$ blocking flows to complete the computation from this point since each blocking flow raises the value of the flow by at least one. So the whole algorithm will take $O(m^{1/2})$ blocking flows, and thus $O(m^{3/2})$ time. ■

The condition in theorem 2.2.2.3 that the graph has no more than two edges between any pair of nodes allows the theorem to apply to the residual graph of a graph with no parallel edges.

Theorem 2.2.2.3 *[1] In a directed graph with no more than two edges between any pair of vertices and no flow on any edge, the maximum distance from the source to the sink is less than $3n/\sqrt{v}$.*

Proof. Let V_i and l be the same as in the proof of theorem 2.2.2.1. Then, for all $0 \le i < l$, $2 \cdot |V_i| \cdot |V_{i+1}| \ge v$ (since we have at most two edges between any pair of vertices) and thus either $\sqrt{2} \cdot |V_i| \ge \sqrt{v}$ or $\sqrt{2} \cdot |V_{i+1}| \ge \sqrt{v}$. Since $\sum_{i=0}^{l} |V_i| \le |V|$, we have $\frac{\sqrt{v}}{\sqrt{2}} \cdot \left\lfloor \frac{l+1}{2} \right\rfloor \le n$ and thus $l \le 3n/\sqrt{v}$. ■

Theorem 2.2.2.4 *[1] In a undirected, unit-capacity graph with no parallel edges Dinic's algorithm runs in $O(mn^{2/3})$ time.*

Proof. Same as theorem 2.2.2.2, except with $n^{2/3}$ substituted for $m^{1/2}$. ■

Theorem 2.2.2.5 *[1] In an undirected, unit-capacity graph with no parallel edges Dinic's algorithm runs in $O(m \cdot \min\{n^{2/3}, m^{1/2}\})$ time.*

Proof. Immediate from theorems 2.2.2.2 and 2.2.2.4. ■

The only heuristics used in the implementation of this algorithm are used to find individual blocking flows and will be discussed in the blocking flow section of the implementation chapter.

2.2.3 Goldberg-Rao Sparse Blocking Flows

The Goldberg-Rao sparse blocking flow algorithm centers around the Nagamochi and Ibaraki sparsification procedure and blocking flows, as does the Karger-Levine sparse blocking flow algorithm, but does not invoke augmenting paths at all and takes a distinctly different approach to sparsification. The Goldberg-Rao algorithm does not remove an edge from the graph unless it will never be needed. Although this "lazier" approach to edge removal results in slightly worse

asymptotic performance, it would seem that its relative simplicity should entitle it to smaller constant factors in practice (this will be explored later).

To explain the Goldberg-Rao algorithm, we introduce the following terminology. We define E^0 to be the set of edges of $G = (V, E)$ that have no flow on them, and E^1 to be the set of edges with flow on them (in either the positive or negative direction). E^0 consists entirely of undirected edges; E^1 consists entirely of directed edges. We also define $G^0 = (V, E^0)$ and $G^1 = (V, E^1)$. Pseudocode for the Goldberg-Rao algorithm appears below.

```
                    GOLD(G)
1   DO
2       D_f = Shortest Path from s to t in G_f (computed by
              breadth-first search)
3       FOREST(G^0)
4       E' = Edges of First (n/D_f)^2 Maximal Spanning Forests of
             G^0 (computed by FOREST)
5       G = (V, E' ∪ E^1)
6       BLOCKINGFLOW(G)
7   WHILE BLOCKINGFLOW found additional flow
```

Theorem 2.2.3.1 *[4] At any point during the execution of the algorithm, there are at most* $4n^{3/2}$ *edges that carry flow.*

Theorem 2.2.3.2 *[1, 3] In a simple graph with a flow f, the maximum residual flow is at most* $2 \cdot (n/D_f)^2$, *where* D_f *is the length of the shortest s-t path in* G_f.

Theorem 2.2.3.2 *[4] On an undirected, simple graph* GOLD *runs in* $O(\min\{m, n^{3/2}\}m^{1/2})$.

Proof. Lines 2-6 take $O(m)$ time per iteration of the loop, so we need only bound the number of times the loop runs. If $m = O(n^{3/2})$, then since GOLD spends the same amount of time (asymptotically) per execution of BLOCKINGFLOW as Dinic's algorithm, the $O(m^{3/2})$ time bound we proved for Dinic's algorithm holds, since GOLD always preserves enough edges to keep the

flow. Since $m = O(n^{3/2})$, $m^{3/2} = O(n^{3/2}m^{1/2})$ (also note that since GOLD and Dinic's algorithm spend the same asymptotic amount of time per execution of BLOCKINGFLOW, GOLD dominates Dinic's algorithm). Otherwise, $m = \Omega(n^{3/2})$. We divide the iterations into three groups: those at the beginning where $n(2n/D_f)^2 > m$, those at the end where $n(2n/D_f)^2 \le 4n^{3/2}$, and those in the middle where neither of those conditions hold. During the iterations where $n(2n/D_f)^2 > m$, $D_f < 2n^{3/2}/m^{1/2}$. Since D_f increases by at least one each time BLOCKINGFLOW runs, these iterations take $O(n^{3/2}m^{1/2})$ time. During the final iterations when $n(2n/D_f)^2 \le 4n^{3/2}$, the remaining flow (which is bounded by $(2n/D_f)^2$ by 2.2.3.2) is less than $4n^{1/2}$ so the sparsification reduces the number of edges to $M = O(n^{3/2})$. The $O(M^{3/2})$ time bound we proved for Dinic's algorithm holds in this case (as shown above). Since $M = O(n^{3/2})$ and $m = \Omega(n^{3/2})$, $M^{3/2} = O(n^{9/4}) = O(m^{1/2}n^{3/2})$, since. Finally, we examine the remaining iterations. Let i be D_f during the ith iteration. Then the number of edges during that iteration m_i is bounded by $n(2n/i)^2 + 4n^{3/2}$, since the remaining flow will be $(2n/i)^2$, so that many maximal spanning trees each with at most n edges will be kept by the sparsify operation. By theorem 2.2.3.1, the $4n^{3/2}$ term bounds the number of edges that are kept that already have flow. Since each iteration will take m_i time and this phase starts when $D_f = 2n^{3/2}/m^{1/2}$, we get

$$\sum_{i=2n^{3/2}/m^{1/2}}^{\infty} n\left(\frac{2n}{i}\right)^2 + 4n^{3/2} \le \sum_{i=2n^{3/2}/m^{1/2}}^{\infty} 2n\left(\frac{2n}{i}\right)^2 = 8n^3 \sum_{i=2n^{3/2}/m^{1/2}}^{\infty} \left(\frac{1}{i}\right)^2$$

$$\le \frac{8n^3}{\left(2n^{3/2}/m^{1/2}\right)^2} \sum_{i=2n^{3/2}/m^{1/2}}^{\infty} \frac{1}{\left\lfloor m^{1/2}i/2n^{3/2} \right\rfloor^2}$$

$$\le \frac{8n^3}{2n^{3/2}/m^{1/2}} \sum_{i=1}^{\infty} \frac{1}{i^2} \le \frac{4n^{3/2}m^{1/2}\pi^2}{6} = O(m^{1/2}n^{3/2}) \qquad \blacksquare$$

Heuristics used in this algorithm, many of which are suggested by Goldberg and Rao [4], will be discussed in the implementation section.

2.2.4 Karger-Levine Sparse Augmenting Paths

The Karger-Levine sparse augmenting paths algorithm (SA3) [3] uses the graph sparsification technique of Nagamochi and Ibaraki [11] to reduce the number of edges present in the graphs on which it searches for augmenting paths by examining ever-growing subsets of the edges at any given time while holding extra edges in reserve. It also uses a doubling-estimation trick to estimate the final value of the flow (which the execution of the algorithm depends on). The core of the algorithm is the routine SPARSEAGUMENT2, for which source code appears below. FOREST was explained in section 2.1.3; DECYCLE is explained below.

```
                    SPARSEAUGMENT2(G)
1   k = ⌈m/n⌉
2   oldflow = newflow = 0
3   DO
4       oldflow = newflow
5       DECYCLE(G)
6       FOREST(G)
7       E_k = Edges of First k Maximal Spanning Forests of G^0
              (computed by FOREST)
8       G_k = (V, E_k ∪ E^1)
9       AUGPATH(G_k)
10      FOR all e ε G_k
11          e' = Edge in G Corresponding to e in G_k
12          f(e') = f(e)
13      newflow = f(G)
14  WHILE newflow – oldflow ≥ k
```

SPARSEAUGMENT2 repeatedly obtains a new graph that consists of G^1 combined with the edges from the first k maximal spanning forests of G^0 and runs AUGPATH (which can be any of the augmenting paths algorithms from section 2.2.1 or Dinic's algorithm from section 2.2.2). DECYCLE is an algorithm that takes a flow in a graph and removes any cycles while preserving the same flow value [3, 19].

We now sketch the correctness proof and running time analysis of SPARSEAUGMENT2.

Theorem 2.2.4.1 *[3] Let E_k be the edges of the first k maximal spanning forests of G^0, and let $G_k = (V, E_k \cup E^1)$. Then G_k contains exactly $i < k$ augmenting paths if and only if G_f contains exactly i augmenting paths, and G_k contains at least k augmenting paths if and only G_f contains at least k augmenting paths.*

Then, since SPARSEAUGMENT2 terminates when AUGPATH increases the flow value by less than k (indicating it found less than k augmenting paths), by Theorem 2.2.4.1 there must be no augmenting paths remaining in G_f, so the flow is maximal.

Theorem 2.2.4.2 *[3, 20, 21] An acyclic flow f in a simple graph uses at most $3n\sqrt{v}$* edges.

Theorem 2.2.4.3 *[3, 19] In a unit-capacity graph, it is possible to take a flow f and find an acyclic flow of the same value in time linear in the number of flow-carrying edges in f.*

Theorem 2.2.4.4 *The running time of SPARSEAUGMENT2(G) on a simple graph is $O(m + r(n\sqrt{v} + \sqrt{mn}))$, where r is the number of augmenting paths that need to be found.*

Proof. Lines 1-3 clearly take $O(1)$ time. Lines 5-9 take $O(m)$ time per iteration of the loop by theorem 2.2.4.3. Lines 11-14 also take $O(m)$ time per iteration of the WHILE loop. The cost of line 10 during the ith iteration of the loop is $O(m_i r_i)$, where m_i is the number of edges in G_k during the ith iteration of the loop and r_i is the number of augmenting paths found in the ith iteration of the loop. By theorem 2.2.4.2 and the fact that the first k maximal spanning forests contain at most nk edges, $m_i \leq nk + n\sqrt{v}$. Since each augmenting path increases the flow by one, and the algorithm terminates when a loop does not increase the flow by at least k, each loop must find at least k augmenting paths, and thus there are at most $\lceil r/k \rceil$ iterations of the loop. Summing, we get:

$$O(1)+\sum_{i=1}^{\lceil r/k\rceil}(m+m_i r_i)\le O(1)+m\cdot\lceil r/k\rceil+\sum_{i=1}^{\lceil r/k\rceil}r_i\left(nk+n\sqrt{v}\right)\le O(1)+m(r/k+1)+r\left(nk+n\sqrt{v}\right)$$
$$=O(1)+m+r\sqrt{mn}+r\left(\sqrt{mn}+n\sqrt{v}\right)=O\left(m+r\left(\sqrt{mn}+n\sqrt{v}\right)\right) \quad \blacksquare$$

There is a subtle difference between this version of SPARSEAUGMENT2 and the original one [3]. In the original, the AUGPATH routine is allowed to find at most k augmenting paths each time it is called. The difference changes the analysis a bit, but does not affect the asymptotic run time (although it seems that the version described here should be better in practice, as it does more work on graphs with slightly fewer edges).

This running time is not quite ideal; when $m \ge nv$ the $\sqrt{mn}$ term dominates the $n\sqrt{v}$ term. If, however, we could guess v at the beginning we could begin the algorithm with a sparsification down to nv edges and achieve an $O\left(m+rn\sqrt{v}\right)$ running time. Fortunately, we can in fact guess v, either by using a graph compression technique to get a 2-approximation of v in $\tilde{O}(m + nv)$ time [22] or by a doubling trick. Although both achieve the same theoretical bound, the doubling trick is much simpler to implement; thus we use it in this study and analyze it here. SPARSEAUGMENT3, which contains the doubling trick, calls SPARSEAUGMENT2 as its primary subroutine. Below is pseudocode for SPARSEAUGMENT3.

SPARSEAUGMENT3(G)

1 $w = f(G)$
2 $G_{initial} = G$
3 FOREST($G_{initial}$)
4 $oldflow = newflow = 0$
5 DO
6 $oldflow = newflow$
8 E_w = Edges of First w Maximal Spanning Forests of $G^0_{initial}$ (computed by FOREST)
9 $E_{w'} = E_w$
10 WHILE $|E_{w'}| \leq 2 \cdot |E_w|$
11 $w' = w' + \lfloor (|E_{w'}| - 2 \cdot |E_w|)/n \rfloor$
12 $E_{w'}$ = Edges of First w' Maximal Spanning Forests of $G^0_{initial}$ (computed by FOREST)
13 $E' = E_{w'} \cup E^1_{initial} \cup E^1$
14 $G' = (V, E')$
15 SPARSEAUGMENT2(G')
16 FOR all $e \in G'$
17 e' = Edge in G Corresponding to e in G'
18 $f(e') = f(e)$
19 $newflow = f(G)$
20 WHILE $newflow - oldflow \geq w$

It is important to notice that $G_{initial}$ does not change during the algorithm, so FOREST need only be called on $G_{initial}$ once as long as the results are stored. Line 11 results from the observation that each spanning forest has at most n edges.

Theorem 2.2.4.5 *The running time of SPARSEAUGMENT3 on a simple graph is* $O(m + rn\sqrt{v})$, *where r is the number of augmenting paths that need to be found.*

Proof. Lines 1-4 run in $O(m)$ time. Lines 6-8 run in $O(1)$ time per iteration of the loop (notice that until line 12 we do not actually need to create E_w, just know, for each value of w, how many edges it has, which we can determine once and then store in $O(m)$ time). Lines 10 and 11 run in $O(1)$ time per iteration of the WHILE loop. Since $w \leq n$ (there are at most n non-empty maximal

spanning forests), this loop can execute at most $O(n)$ times per iteration of the larger loop. Lines 12 and 13 are very tricky; it would seem that it would require $O(m)$ time to look through all the edges and choose the ones we need. We can, however, choose the correct edges in $O(m_i + n)$ (where m_i is the number of edges in E' in the ith iteration of the loop) time if we proceed cleverly; the tricks needed will be discussed in the implementation section. Line 15 runs in $O(m_i + r_i(n\sqrt{v} + \sqrt{m_i n}))$ time by theorem 2.2.4.4 (where r_i is the number of augmenting paths found in the ith iteration of the loop). Summing, we get

$$O(m)+\sum_{i=1}^{\lfloor \log(v)+1 \rfloor}\left[O(m_i+n)+O\left(m_i+r_i\left(n\sqrt{v}+\sqrt{m_i n}\right)\right)\right]=O(m)+\sum_{i=1}^{\lfloor \log(v)+1 \rfloor}O\left(m_i+r_i\left(n\sqrt{v}+\sqrt{m_i n}\right)\right).$$

Since we double w each time we restart the DO-WHILE loop and the algorithm terminates when $w > v$, w reaches a maximum value of $2v + 1$ (the extra 1 comes from the non-strict inequality in line 10). Thus, since a maximal spanning forest can have at most n edges, $m_i \leq 2nv + n$. So

$$\begin{aligned}O(m)+\sum_{i=1}^{\lfloor \log(v)+1 \rfloor}O\left(m_i+r_i\left(n\sqrt{v}+\sqrt{m_i n}\right)\right) &\leq O(m)+\sum_{i=1}^{\lfloor \log(v)+1 \rfloor}O\left(m_i+r_i\left(n\sqrt{v}+\sqrt{2n^2v+nv}\right)\right)\\ &=O(m)+\sum_{i=1}^{\lfloor \log(v)+1 \rfloor}O(m_i)+\sum_{i=1}^{\lfloor \log(v)+1 \rfloor}O\left(r_i n\sqrt{v}\right)\\ &=O(m+rn\sqrt{v})\end{aligned}$$

since the sum of the r_i's is r and the m_i's double each time until they reach m. ■

There are many heuristics and implementation choices in the implementation of this algorithm, some of which alter the theoretical running time of the actual implementation (though only by log factors). These choices will be discussed in depth in the implementation section.

2.2.5 Karger-Levine Sparse Blocking Flows

The Karger-Levine sparse blocking flows algorithm is an extension of the Karger-Levine sparse augmenting paths algorithm that uses blocking flows to perform the first augmentations and thus achieves a slightly better running time on dense graphs with small flow values. Like the

augmenting paths algorithm, its execution also relies on the value of the flow, so it must also use a doubling trick to estimate the value of the flow. Pseudocode for BLOCKTHENAUGMENT appears below.

```
                    BLOCKTHENAUGMENT(G)
1   k = 1
2   D_f = Shortest Path from s to t in G_f (computed by
          breadth-first search)
3   DO
4       WHILE D_f < kn/m^{1/3}
5           BLOCKINGFLOW(G)
6           D_f = Shortest Path from s to t in G_f (computed by
                  breadth-first search)
7       SPARSEAUGMENT3(G) stopping when f(G) ≥ k^6
8       k = k · 2
9   WHILE SPARSEAUGMENT3 stopped before it had found all paths
```

Notice that some changes to SPARSEAUGMENT3 are required to ensure that it terminates when $f(G) \geq k^6$.

Theorem 2.2.5.1 *On an undirected simple graph* BLOCKTHENAUGMENT *runs in* $O\left(nm^{2/3}v^{1/6}\right)$ *time.*

Proof. Lines 1 and 2 take $O(m)$ time. Lines 5 and 6 take $O(m)$ time for each iteration of the inner WHILE loop. Since D_f increases by at least 1 each time the BLOCKINGFLOW subroutine is run, lines 4-6 take at most $O\left(k_i nm^{2/3}\right)$ time for the ith iteration of the outer WHILE loop. By theorem 2.2.3.2, in the ith iteration the call to SPARSEAUGMENT3 will take

$O\left(m + 2n\left(\frac{n}{kn/m^{1/3}}\right)^2 \sqrt{k^6}\right) = O\left(m + nm^{2/3}k\right)$. Notice that since we do not allow v to exceed k^6 during SPARSEAUGMENT3, we can replace v with k^6 in the running time of SPARSEAUGMENT3. Since we double k each time, and we can be assured of running SPARSEAUGMENT3 to completion when $k^6 > v$, k^6 can reach at most $2v$. Using this fact and summing, we get

$$O(m)+\sum_{i=0}^{\lfloor 1+\log_2 v^{1/6}\rfloor} O\left(m\cdot\frac{n}{m^{1/3}}\cdot 2^i\right)+O\left(m+nm^{2/3}2^i\right)\le O(m)+O(nm^{2/3})\sum_{i=0}^{1+\log_2 v^{1/6}}2^i+\sum_{i=0}^{1+\log_2 v^{1/6}}O(m)$$

$$\begin{aligned}
&= O(m)+O(nm^{2/3})\frac{1-2^{\log_2 v^{1/6}+1}}{1-2}+O\left(m\log v^{1/6}\right)\\
&= O\left(m\log v^{1/6}\right)+O\left(nm^{2/3}\right)\left(2v^{1/6}-1\right)\\
&= O\left(m\log v^{1/6}+nm^{2/3}v^{1/6}\right)\\
&= O\left(nm^{2/3}v^{1/6}\right) \qquad \blacksquare
\end{aligned}$$

Most of the heuristics and implementation choices that affect this algorithm are actually within SPARSEAUGMENT3 and will be discussed within the implementation section on that algorithm; the remainder will be discussed in the implementation section on this algorithm.

2.2.6 Karger Randomized Augmenting Paths

The Karger randomized augmenting path algorithm [5] is unique among the new algorithms we tried in several ways. First, it is the only algorithm that is non-deterministic; as a consequence, for this algorithm we must examine the "Las Vegas" (high-probability) running time, whereas for all the other algorithms consider the "definite" worst-case bound. Furthermore, as with many randomized algorithms, the description of the algorithm itself is very simple, but the analysis of the running time is extremely involved. Finally, unlike most of the algorithms, which use the Nagamochi and Ibaraki algorithm to sparsify, this algorithm relies on sampling properties of undirected graphs to achieve sparsification. Here we explain how the algorithm runs and argue why we should expect it to run fast.

The algorithm randomly divides the edges of the graph into two subgraphs, and recursively finds the flow in each of those graphs. It then pastes the graphs back together, and runs augmenting paths or blocking flows on the resulting graph to find the remaining flow. To explain the running time of this algorithm, we introduce a theorem.

Theorem 2.2.6.1 *[5] If each cut of G has value at least c and edges are sampled with probability p, then with high probability all cuts in the sampled graph are within* $1 \pm \sqrt{8 \ln n / pc}$ *of their expected values.*

When we assign the edges with probability ½ to each of the two subgraphs, we expect that the minimum *s-t* cut is $\frac{v}{2}\left(1 - O\left(\sqrt{\log n/c}\right)\right)$, which will give us a flow of at least $v\left(1 - O\left(\sqrt{\log n/c}\right)\right)$ when we paste the graphs back together. There remain $O\left(v\sqrt{\log n/c}\right)$ augmenting paths to be found. The time to find these augmenting paths is the dominant factor in the recursion, which makes the final running time of $O\left(mv\sqrt{\log n/c}\right)$ unsurprising.

While clearly not a proof, this discussion is sufficient to supply the intuition behind the algorithm. Most of the implementation decisions and heuristics involved in this algorithm lurk within the decision of which augmenting paths algorithm to use; the other implementation choices and heuristics will be discussed in the implementation section. As a final detail, notice that if we choose the Karger-Levine sparse augmenting paths algorithm as our augmenting path subroutine (instead of one of the other four $O(m)$ time options), we achieve a running time of $\tilde{O}\left(m + nv\sqrt{v/c}\right)$.

Chapter 3

Implementation

In theory, theory and practice are the same; in practice, they differ. Unfortunately, while the theory behind a certain algorithm may dictate the broad boundaries within which its running times will fall, the precise implementation can often make a tremendous difference. In this chapter, we discuss the heuristics and implementations that we tried as well as indicate which ones we used in our final tests.

In creating these implementations, we tried to pay equal attention to each of the algorithms, with the exception of the directed approaches, which were already extensively refined. While we did take some care to produce efficient code, we did not spend excessive time attempting to tweak every last microsecond from the code; this tweaking can presumably be done to any code, and does not help us determine which algorithms are best.

Finally, while we recognize that most heuristics occasionally hurt the running time of the algorithm, we attempted to ensure, particularly with algorithms other than the directed approaches, that the time spent actually running the heuristics did not have a dominating influence on the run time.

We begin by discussing the graph data structures; continue on to discuss the implementations of several algorithms that underlie many of the other algorithms; and finally detail each algorithm's heuristics and implementation. With the exception of the first section, this chapter roughly parallels the previous one.

3.1 Graph Data Structures

In implementation studies, the choice of data structure can often have a large effect on the results. For this reason, we attempted to keep our data structures as consistent as possible across the implementations of different algorithms, and as a result we only have one basic graph data structure.

The implementations of all the algorithms utilize a graph data structure that consists of two arrays. The first is an array of all the edges in the graph sorted by the vertex from which they leave. The second array is of all the vertices; each vertex has a pointer to the first and (depending on the algorithm) possibly the last edge that leave from it. In the data structure, both the edges and vertices are structures, so additional arrays are necessary only to store information about the edges and vertices that is of very temporary importance (i.e. arrays that are only visible within one function).

To aid efficiency, we represent an undirected edge using only two directed edges; each represents one direction of the undirected edge and the reverse edge of its partner. This representation effectively cuts the number of edges by a factor of two over the natural four-edge representation (using one edge for each direction of the directed edge and one edge for each reverse edge).

3.2 Basic Algorithms

As mentioned earlier, many of the algorithms in this study share common subroutines. Thus, ensuring that these subroutines use intelligent heuristics and are well implemented is crucial, as each mistake or oversight could impact multiple algorithms. Furthermore, improving these common building blocks allows us to improve many different algorithms at once, contributing to programmer efficiency. Finally, it is important to note that sharing mostly the same implementation between different algorithms represents a decision in that the subroutines are

used in sufficiently similar ways so as to suggest that optimizing each occurrence individually is probably somewhat useless.

Although we treated the algorithms mostly the same, and although there is some support for this decision (theoretical work has generally treated the subroutines interchangeably), optimizing the basic algorithms differently for different uses is an interesting possibility that we did not completely explore. However, there seems to be no reason to think that the different algorithms pass graphs with different characteristic structures into the basic algorithms.

3.2.1 Augmenting Paths

The implementations of all four augmenting path algorithms were taken directly from Cherkassky, Goldberg, Martin, Setubal, and Stolfi [6], to which the reader can refer for more information. Although we examined the codes, it did not seem that we could make substantial heuristical improvements to any algorithm in all situations, so we did not alter the codes substantially. However, the heuristics involved in these algorithms play a large part in the final conclusions of our study, so we describe them in detail here.

As mentioned in the theory section, breadth-first search utilizes two major heuristics. The first is simple; instead of doing a breadth-first search from the source, we do a separate breadth-first search from each of the source's neighbors. Consequently, we do not necessarily find the shortest path to the sink, so in that way it differs from standard breadth-first search. It also, however, means that often we expand much less of the graph before we find a path, and that we do not use as much memory as we do not need to hold as large a search tree. This heuristic was found to be helpful [6]; seeing no reason why it should be different for undirected graphs, we did not re-test that assessment. We also use the *clean-up heuristic*, which says that if a search from a vertex fails, we mark that vertex so it is not searched from in the future. This heuristic, due to Chang and McCormick [23], is permissible because a search from a vertex only fails if there is no path from that vertex to the sink, and once there is no path, one will not be created later in the algorithm.

Depth-first search uses two major heuristics, which are similar to breadth-first search. First, depth-first search incorporates the clean-up heuristic, which functions identically in depth-first and breadth-first search. Second, the depth-first search includes a *look-ahead heuristic*, which entails checking all of a node's neighbors to see if any of them is the sink the first time the node is reached. As in the clean-up heuristic, the look-ahead heuristic allows a path to be found after a smaller fraction of the graph is searched.

The label-directed search and augment-relabel algorithms employ only a single major heuristic, global relabelling [6, 24] (which they actually share with the push-relabel algorithms and, in some sense, with the blocking flow algorithms). Since both algorithms have the same heuristic, we discuss them together. The global-relabelling heuristic uses a breadth-first search from the sink to label vertices with their distances from the sink. This search takes linear time, and thus to preserve the theoretical time bounds, we can only apply it after $\Theta(m)$ work has been done since the last global relabelling. However, it was found to be preferable, in practice, to apply a global relabelling after only $\Theta(n)$ work [6]. We continue that strategy here with some reservations as it does render the theoretical bounds invalid.

3.2.2 Blocking Flows

Our implementation of blocking flows began with the implementation for Dinic's algorithm from Anderson and Setubal [13]. On close examination this implementation differs from the implementation suggested by Even and Tarjan [1] in several interesting ways.

First, Even and Tarjan describe the algorithm as keeping the distance labels from breadth-first search to help guide the depth-first search; the implementation by Anderson and Setubal does not. On further inspection, this choice seems intelligent; in a layered graph, all edges from nodes with a certain distance label go to nodes with a distance label one greater, so the labels are superfluous.

Second, the Anderson-Setubal implementation does not terminate the breadth-first scan from the source when it completes the layer of nodes where the sink is first reached. As a result, the implementation has not only a longer breadth-first search, but also spends more time in

depth-first searches. It cannot, however, affect what paths are found (the nodes in layers further from the source than the sink will not have arcs to the sink in the layered graph). Since it was not clear immediately whether this heuristic was helpful, we were careful to test it extensively. These tests revealed that this heuristic is helpful, but often not as much as hoped. Many graph families that we looked at have the characteristic that most of their edges are between vertices that are closer to the sink than to the source; obviously, for these graphs the heuristic is not very helpful (though not harmful either). In graph families where many edges are between vertices farther from the source than the sink is, this heuristic can be very helpful.

Finally, there is the choice of whether to actually create separate layered graphs (which involves making copies of graphs repeatedly) or simply set flags on each of the edges to indicate whether they are in the layered graph. Fortunately, the implementation of the cleanup heuristic suggested by Even and Tarjan (removing edges from the layered graph after they are checked) and used by Anderson and Setubal guarantees that the flags of the edges not in the layered graphs are checked only once during the search for paths along which to route flow, so copying would not be worthwhile.

The final implementation of blocking flows incorporates the early search termination heuristic but follows Anderson and Setubal in not retaining the distance labels of the vertices and setting flags on edges instead of copying when representing the layered graphs.

3.2.3 Graph Sparsification

The implementation of graph sparsification given by Nagamochi and Ibaraki does not admit much in the way of heuristic improvement; it is already a very tight procedure from a point of view of constant factors. There is, however, an implementation choice to be made as far as data structures are concerned. The $O(m)$ time implementation requires a data structure that allows constant time extraction of the vertex with the largest index as well as constant time vertex index changes. The authors [11] suggest a bucket-based data structure to meet these theoretical demands, but Levine [17] indicates that performance testing suggested that a 4-ary heap works better, at least on capacitated graphs. We follow Levine and actually utilize their

implementation; however, it would be interesting to try a bucket-based implementation. Note, however, that this choice bloats the theoretical running time of graph sparsification from $O(m)$ to $O(m + n\log n)$, although if $m = \Omega(n^a)$ for $a > 1$, these are equivalent. This change affects the theoretical running times of algorithms based on sparsification; these effects will be noted in the sections discussing this algorithm.

3.3 Augmenting Paths and Push-Relabel

As mentioned earlier, the implementations in this area were all taken from Cherkassky, Goldberg, Martin, Setubal, and Stolfi [6], which contains more details. The augmenting paths implementation are fairly uninteresting; they just run the augmenting paths routines over and over until the flow is maximal. As such, there are not any particularly interesting heuristics to be applied to them. The push-relabel algorithms, however, allow for many implementation choices.

The first, and most important, heuristic decision to be made in implementing push-relabel is the decision of how to select which active vertex to process. There are a number of strategies for this task; Cherkassky, Goldberg, Martin, Setubal, and Stolfi [6] consider three and provide the code for four, all of which we examined. The four strategies are highest-level; lowest-level; first-in, first-out; and last-in, first-out. The highest-level and lowest-level strategies, as their names suggest, choose the vertex with the highest or lowest distance label, respectively. First-in, first-out and last-in, first-out are based on the respective types of queues; they simply enqueue nodes as they are processed and pull them out in order.

Another major heuristic decision is whether to use of global relabelling, as discussed in the section on augmenting paths implementations. Cherkassky, Goldberg, Martin, Setubal, and Stolfi [6] includes this strategy, using it after $O(n)$ work is done. As before, this decision is not theoretically justifiable, but has been found to yield better results. In the context of push-relabel, it is important to note that global relabelling will also determine if there is any possibility of more flow reaching the sink (if no nodes with excess have distance labels lower than n, then no more excess will reach the sink). This observation is important in this implementation due to the use of the two phase approach, discussed later.

Yet another major heuristic for push-relabel (it is interesting to speculate whether the computational performance of this algorithm is inherent or due to the preponderance of heuristics developed for it) is the *gap heuristic*. This heuristic, which is similar to the cleanup heuristic, [25, 26] relies on the observation that if there are no vertices with label x, we can ignore all vertices with labels greater than x. This observation stems from the definition of the distance function. Cherkassky, Goldberg, Martin, Setubal, and Stolfi [6] implements it for the highest-level and lowest-level routines, but not for the first-in, first-out and last-in, first-out routines, since the gap heuristic is costly without bucket-based data structures, which are needed for highest-level and lowest-level anyway, but can be dispensed with in first-in, first-out and last-in, first-out.

The last heuristic used in this implementation is the two phase approach. In this approach, when there are no active vertices the push-relabel algorithm (phase one) ends, and another algorithm is used to send flow back to the source. This algorithm works by searching from all nodes that have excess along edges with flow and removing cycles, and then removing flow from edges going into vertices with excesses until the vertices no longer have excess. Due to the use of this heuristic, we need only raise the vertex labels to $n + 1$ before vertices become inactive (since they can be a distance of at most n from the sink, which is labelled zero). Also, we set the initial excess of the source to be the minimum of the number of edges leaving the source and the number of edges entering the sink, which potentially reduces the amount of flow we will need to push back. Finally, we set the excess of the sink to be $-\infty$ to insure it never becomes active.

With the details out of the way, we can now analyze these algorithms theoretically. These analyses assume that global relabelling runs only after $O(m)$ work has been performed since the last global relabelling; although we do not follow this practice (as mentioned), theoretical analysis of performing global relabelling after only $O(n)$ work does not seem to exist (and is a potentially interesting open problem).

First we show that push-relabel, independent of the choice of the vertex-selection heuristic, runs in $O(nm)$ time.

Theorem 3.3.1 *Push-relabel executes in $O(nm)$ time on a unit capacity graph.*

Proof. We analyze the amount of time taken by the relabel operations and by the push operations. Since each relabel must examine the distance labels of all of the target vertex's neighbors, it takes $O(neighbors)$ time. Considering a set of relabel operations consisting of one relabel on each vertex in the graph, we see that there will be $O(m)$ neighbors and thus the set of relabel operations will take $O(m)$ time. Since each relabel increases the value of the distance label by at least one, and the distance labels need to increase to at most $O(n)$, this relabelling takes $O(nm)$ time. Considering pushes, if we keep track of the last edge that each node pushed flow along, resetting when the node is relabeled, we can perform all the pushes required from a node between relabels in $O(neighbors)$ time, so all the pushes can be done in $O(nm)$ time as well. Next, we consider the work involved in running global relabelling. Since we only run it after $O(m)$ work has been done, and it takes $O(m)$ time, we can charge its cost to the rest of the algorithm. Finally, we consider the work done in the second phase of the two-phase heuristic. We can remove all the cycles from the flow using depth-first search along flow-carrying edges in $O(m)$ time and then remove flow from inbound flow-carrying edges to overflowing nodes in $O(m)$ time, so this phase runs in $O(m)$ time, thereby proving that push-relabel runs in $O(nm)$ time. ■

Next, we analyze the run time of push-relabel specifically with the lowest-level vertex selection and global relabelling heuristics [24].

To implement lowest-level selection, we use a collection of buckets $b_0, b_1, \ldots, b_{2n}$, where each bucket b_i contains the active nodes with distance label i. The algorithm also keeps track of u, the index of the bucket from which we last selected a unit of flow excess to process. Then, we know that we can keep processing that unit of excess through a sequence of pushes and relabels until one of the relabels brings a node above u. As mentioned, we also use global relabelling; for this analysis we assume it runs once at the beginning of the algorithm and then once after any unit of flow reaches the source or sink or after $O(m)$ time has been used by pushes and relabels. After each global relabelling we need to rebuild the buckets, which takes $O(n)$ time. Since u

decreases only when we perform a global relabelling, the algorithm spends at most $O(n)$ time examining empty buckets between global relabels.

Theorem 3.3.2 *The algorithm performs $O(km)$ work during the period before u increases above k, where k is a chosen parameter with $2 \leq k \leq n$.*

Proof. In this period, each node can be relabeled at most $k + 1$ times. By the arguments used in the proof of 3.3.1, the pushes and relabels take $O(km)$ time. Since there are at most $O(k)$ global relabellings in the period, and each takes at most $O(m)$ time, the time spent in global relabelling is $O(km)$. Finally, we spend at most $O(kn) = O(km)$ time examining empty buckets. Combining the terms, we get the $O(km)$ bound. ■

Theorem 3.3.3 *The residual flow in a unit capacity simple graph is at most* $\min\{m/D_f, 2(n/D_f)^2\}$.

Proof. The $2(n/D_f)^2$ term is from theorem 2.2.3.2; the m/D_f follows immediately from the definition of D_f. ■

Theorem 3.3.4 *The algorithm performs at most $O(m)$ work between successive pushes of excess flow to the sink.*

Proof. At any point in the execution of the algorithm, there will be at most $O(m)$ time until the next global relabelling. That global relabelling will take $O(m)$ time. After that global relabelling, either the algorithm will stop performing push-relabel and move to phase two, or there will be a path to the source. Pushing a unit of excess along this path will require no relabels, and at most $O(m)$ time to perform the pushes. So there is at most $O(m)$ time from any point in the execution of the algorithm until the next push of excess flow to the sink, which proves the theorem. ■

Theorem 3.3.5 *Two phase push-relabel with the lowest-level selection and global relabelling heuristics runs in* $O(m \cdot \min\{m^{1/2}, n^{2/3}\})$ *time.*

Proof. By theorem 3.3.2, the work performed until u increases above k is $O(km)$. After the u goes above k, by theorem 3.3.3 the residual flow is at most $\min\{m/k, 2(n/k)^2\}$. By theorem 3.3.4, the work done after u goes above k is $m \cdot \min\{m/k, 2(n/k)^2\}$. We know from the proof of theorem 3.3.1 that the second phase only takes $O(m)$ time. So the algorithm takes $O(km + m \cdot \min\{m/k, 2(n/k)^2\})$. Choosing k to balance the terms and thus minimize the running time, we get $O(m \cdot \min\{m^{1/2}, n^{2/3}\})$. ■

3.4 Dinic's Algorithm

The implementation of Dinic's algorithm is very straightforward; it just repeatedly runs the blocking flow computation. There are no significant heuristics used (besides those in the blocking flow computation itself), and the simplicity of the algorithm suggests that there are not any to consider.

3.5 Goldberg-Rao Sparse Augmenting Paths

The Goldberg-Rao sparse augmenting paths algorithm, because it is simpler than several of our other algorithms, requires only a couple of implementation choices. First, we chose to use the implementation of FOREST discussed in section 3.2.3. This changes the theoretical running time of the Goldberg-Rao algorithm from $O(m^{1/2} \cdot \min\{m, n^{3/2}\})$ to $O((m^{1/2} + n\log n) \cdot \min\{m, n^{3/2}\})$. Second, we must choose when to run FOREST. The original algorithm specifies that it should be run before every blocking flow computation, but Goldberg and Rao [4] suggest some decision criteria for whether it needs to be run.

They point out that at the beginning of the algorithm, when $n(2n/D_f)^2 \geq m$, the first $(2n/D_f)^2$ maximal spanning trees will definitely contain all the edges in the graph, so running FOREST is useless. Restricting when FOREST runs using this criterion results in a respectable speed increase. Their other observation is that at the end of the algorithm, when $4n^{3/2} \geq |E^1| \geq n(2n/D_f)^2$, $M = O(n^{3/2})$, and thus the sparsification may not achieve an asymptotic improvement in the number of edges in the graph. Experimentation on the final test suite of graph families revealed that this heuristic was rarely activated; in fact, in our test suite, none of the graphs achieved the requisite number of edges with flow on them. We chose to include the heuristic, however, as it did not slow down the implementation.

3.6 Karger-Levine Sparse Augmenting Paths

The Karger-Levine sparse augmenting paths algorithm presents many implementation choices, several of which can effect the theoretical running time of the algorithm. We need to decide the following: how to implement for FOREST, how often to run DECYCLE, how to implement AUGPATH, and how to implement lines 11 and 12 of SPARSEAUGMENT3. For convenience, the pseudocode for the algorithm is reproduced below.

```
                    SPARSEAUGMENT2(G)
1   k = ⌈m/n⌉
2   oldflow = newflow = 0
3   WHILE newflow – oldflow ≥ k
4       oldflow = newflow
5       DECYCLE(G)
6       FOREST(G)
7       E_k = Edges of First k Maximal Spanning Forests of G^0
                (computed by FOREST)
8       G_k = (V, E_k ∪ E')
9       AUGPATH(G_k)
10      FOR all e ε G_k
11          e' = Edge in G Corresponding to e in G_k
12          f(e') = f(e)
13      newflow = f(G)
```

```
SPARSEAUGMENT3(G)
1  w = f(G)
2  G_initial = G
3  FOREST(G_initial)
4  oldflow = newflow = 0
5  DO
6      oldflow = newflow
7      E_w = Edges of First w Maximal Spanning Forests of
               G^0_initial (computed by FOREST)
8      E_w' = E_w
9      WHILE |E_w'| ≤ 2·|E_w|
10         w' = w' + ⌊(|E_w'| − 2·|E_w|)/n⌋
11         E_w' = Edges of First w' Maximal Spanning Forests
                  of G^0_initial (computed by FOREST)
12     E' = E_w' ∪ E^1_initial ∪ E^1
13     G' = (V, E')
14     SPARSEAUGMENT2(G')
15     FOR all e ε G'
16         e' = Edge in G Corresponding to e in G'
17         f(e') = f(e)
18     newflow = f(G)
19 WHILE newflow − oldflow ≥ w
```

For FOREST, we follow the implementation outlined in section 3.2.2, using a 4-ary heap that changes the running time of FOREST from $O(m)$ to $O(m + n\log n)$. This implementation changes the running time of SPARSEAUGMENT2 from $O\left(m + r\left(\sqrt{mn} + n\sqrt{v}\right)\right)$ to $O\left(m + r\left(\sqrt{mn} + n\sqrt{v} + n\log n\right)\right)$. The running time of SPARSEAUGMENT3 is also changed, from $O\left(m + rn\sqrt{v}\right)$ to $O\left(m + r\left(n\sqrt{v} + n\log n\right)\right)$. The analysis for both of these changes follows the original analysis, simply replacing the old running time for FOREST with the new one.

Karger and Levine [3] recommend changing the frequency of the running of the DECYCLE algorithm as a practical heuristic. This recommendation does make sense; DECYCLE is only useful when there are cycles in the flow and is not called for theoretically when the number of edges used in the flow is less than $3n\sqrt{v}$. However, profiling shows that the total cost of DECYCLE is always less than 5% of the running time of the algorithm. Due to the profiling

results, we did not experiment with altering when DECYCLE runs; such experimentation might be desired if a perfectly tweaked implementation is a necessity.

The running times of SPARSEAUGMENT2 and SPARSEAUGMENT3, both theoretically and practically, depend heavily on the choice of algorithm for AUGPATH. For this reason, we focussed most of our energy on implementing several choices. We implemented the algorithm using the label-directed search, breadth-first search, depth-first search, and augment-relabel versions of augmenting paths as well as using blocking flows. All of these choices produce the same theoretical time bounds, but the actual running times vary significantly.

Finally, there is the choice of implementation for lines 11 and 12 of SPARSEAUGMENT3. We experimented with two implementations. The first created a new graph each time line 12 was executed by scanning through all the edges of the original graph and copying the edges in the correct maximal spanning forests to the new graph. Unfortunately, this implementation, while having the advantage of being simple, adds an $O(m\log v)$ term to the running time of SPARSEAUGMENT3 and, more importantly, consumes extra memory. The other implementation that we tried involved sorting the edges leaving from each node in the edge array by their maximal spanning forest number. Then, each time we need to create a graph with more maximal spanning forests, we adjust the pointers in each node that signal where that node's edges end in the edge array. This implementation can be performed in $O(m)$ time for the executions of lines 11 and 12 within the entire algorithm, and does not require any extra memory. It does, however, add complexity to the code, and the sorting increases the actual running time. More importantly, it changes the edge order, which could have serious implications for the running times of the AUGPATH routines.

Fortunately, experimentation showed that having the edges in a sorted order never made more than a 10% difference in the running times, and the difference never approached 10% in the label-directed search and augment-relabel algorithms. These algorithms showed less response to the change because, although they still scan through the edges in the same order, they have other criteria for choosing which edges to search along.

Using the second implementation for lines 11 and 12 also requires a choice of a sorting algorithm. Although the theoretical bounds require a linear-time sort, we chose to use quicksort,

as it is relatively simple and has been proven to be efficient in practice [18]. We could have experimented with other sorting algorithms as well, but profiling revealed that the sorting never took more than 10% of the running time. Thus, this experimentation would probably meet with minimal results, so it was not performed.

Overall, our experimentation between the first and second options for making G' revealed that the second option was better, usually by 20% to 25%. Furthermore, it used significantly less memory. One potential negative, however, was the fact that it used an extra four bytes of memory for the vertex representations (due to the addition of a last-arc pointer). Surprisingly, this change can effect running times by as much as 5%, especially for blocking flows. These effects are not a result of copying data structures, but rather a result of worse caching, which surprisingly turns out to be a significant factor in the running time of these algorithms. By caching, we are referring to the on-board processor cache (128K level 2 for code and data and two separate 16K level one caches, one for code and one for data); the test machine had enough RAM that swapping to disk was not necessary.

Our final implementation of the whole SPARSEAUGMENT3 algorithm uses the FOREST implementation from section 3.2.3; does not attempt to intelligently decide when to run DECYCLE; has separate versions for the label-directed search, breadth-first search, depth-first search, and augment-relabel variants of augmenting paths as well as a version for blocking flows; and chooses the second option for implementing lines 11 and 12 of SPARSEAUGMENT3.

Final tuning on the implementation of the algorithm involves the choice of the constant on line 9 of SPARSEAUGMENT3. The theoretical running time of the algorithm does not depend on this constant; there is little guidance as to what value it should have, and the performance of the algorithm is quite sensitive to its exact value. Experimentation on our final suite of graphs led us to choose a value of 2.5, which seemed to work well for all of the subroutines and for most of the graph families. This value seems to work well for a variety of graphs, but any implementation would need to tune this constant to fit the specific types of graphs the algorithm is expected to perform on.

3.7 Karger-Levine Sparse Blocking Flows

Most of the choices in the implementation of the Karger-Levine sparse blocking flows algorithm actually lie within the implementation of SPARSEAUGMENT3. We follow the same implementation we outlined in the previous section, although with the caveat that the option chosen for the implementation of lines 11 and 12 of SPARSEAUGMENT3 is only preferable if the algorithm spends a significant portion of its running time within SPARSEAUGMENT3. If it does not, the significant caching performance hit of blocking flows (as much as 5%) due to the extra pointer required for this choice of implementation outweighs the benefits.

The only choice that need be made within the algorithm proper is the choice of constant on line 8 (the pseudocode is reproduced below for convenience). Similar to the constant in SPARSEAUGMENT3, the theoretical running time of the algorithm does not depend on this constant and little guidance in choosing it is provided, but the actual running time of the algorithm is very sensitive to it. Experimentation on our final suite of graphs led us to choose a value of 1.25 for implementations using augment-relabel or breadth-first search as the flow-finding subroutine and a value of 2 for all other implementations. As with the previous section, these choices would need to be tuned if the algorithm was used on a different set of graphs.

```
                    BLOCKTHENAUGMENT(G)
1   k = 1
2   D_f = Shortest Path from s to t in G_f (computed by
          breadth-first search)
3   DO
4       WHILE D_f < kn/m^(1/3)
5           BLOCKINGFLOW(G)
6           D_f = Shortest Path from s to t in G_f (computed by
                  breadth-first search)
7       SPARSEAUGMENT3(G) stopping when f(G) ≥ k^6
8       k = k · 2
9   WHILE SPARSEAUGMENT3 stopped before it had found all paths
```

3.8 Karger Randomized Augmenting Paths

The description of the Karger randomized augmenting paths algorithm is fairly simple; as a consequence, few heuristic implementation choices are necessary for it. We will examine three: the choice of maximum flow algorithm to find the flow in the graph that results when two subgraphs are pasted back together, the implementation of the data structures necessary to perform the recursion, and the choice of maximum recursion depth.

The choice of which maximum flow algorithm to use is not an obvious one. The proof of the $O\left(mv\sqrt{\log n/c}\right)$ time bound for the algorithm requires only that the flow-finding routine be able to find all the flow within the new graph, and be guaranteed to find it in $O(mv)$ time. This leaves many options open; we created separate implementations for the label-directed search, breadth-first search, depth-first search, and augment-relabel versions of augmenting paths as well as blocking flows.

The details of how the data structures are implemented to allow recursion is also crucial to the actual running time of the algorithm. Copying is an option, but would require large amounts of time and make the memory usage of the algorithm unacceptable. We chose to use the same trick we did in the implementation of Karger-Levine sparse augmenting paths. The edges in each node's section of the edge array are sorted according to their random seed. When the algorithm recurses, it knows what level of the recursion tree it is on, and what the calling path leading to it was. With this information, and the knowledge that on each call we split the range of random seeds in two, each instance of the algorithm can deduce what range of random seeds it is responsible for. It then sets each node's first and last edge pointers so they enclose only the edges with the random seeds within the algorithm's range. As the calls to the algorithm return, the algorithm restores the pointers to their original values using backups of the pointers that were made before they were changed. This allows us to perform the data structure work necessary for the recursion in about $m/2$ time for each level of the recursion tree and not consume copious amounts of memory. This implementation also requires a choice of sort; we used quicksort, which is not theoretically justifiable, but has been shown to perform well in practice. Profiling

showed that the sort is never a dominant factor in the running time of the algorithm, so experimentation on whether other sorts would be preferable was not done.

Finally, we need to choose a recursion depth. Choosing this depth is tricky; it does not effect the theoretical run time of the algorithm and the research on this algorithm does not suggest one. It is clear that the bound cannot be absolute; it must be a function of the number of nodes. Experimentation with the final test suite of problems revealed a few interesting properties of the ideal recursion depth. First, it varies greatly depending on the subroutine used to find augmenting paths. Second, it depends greatly on the structures of the graphs. In general, the harder a particular graph instance is for an augmenting paths algorithm, the more is to be gained from deeper recursion. We set the recursion depth in our final implementations by specifying the maximum size in edges of a subproblem that the algorithm will choose to not recurse on. For each subroutine, this size was a constant factor multiplied by the number of vertices in the graph instance. The constant factors we decided upon were: 2 when augment-relabel or label-directed search were used as the subroutine, 1.5 when blocking flows was used as the subroutine, and .5 when breadth-first search or depth first search was used as the subroutine. These choices should be re-examined if the class of problems the algorithm is being applied to changes.

Chapter 4

Experiments

In this chapter, we discuss the experiments carried out using the implementations in the previous chapter. We will begin by discussing our experimental method and then discuss our results, grouped both by problem family and by algorithm. Full data is presented in Appendix A.

4.1 Experiment Design

The results we show are governed by our choice of input problems to the algorithms. It is unfortunate that there is no objective way to justify which problems are used. We were unable to locate any problems from actual applications of these algorithms, so we attempted to choose our problem families to meet certain goals, which we outline in the next section. We then explain our problem families and give details on our experiments.

4.1.1 Goals

Since it is not clear which problem families should be used, we followed the guidelines below:

1) Problem families used in previous studies should be used (with modification if necessary) to allow comparisons or analogies with previous research.

2) Problem families that expose weaknesses in the algorithms or cause dramatically different performance in different algorithms are interesting.
3) Problem families that highlight certain aspects of graph structure are instructive.
4) Problem families that seem "natural" should be studied.

Rule 1 allows us to root our paper in the previous literature; this has obvious appeal and is considered good practice in experimental papers [27]. Rule 2 is somewhat less obvious, but is defensible on several grounds. It lets us show that the analysis for a certain algorithm is tight and it can reveal the particular aspects of the algorithm that prevent the analysis from being improved. This rule also is common in experimental papers [17]. The third rule allows us to anticipate, in part, future questions. By choosing graph families that are strongly characterized by certain aspects of their structure, we provide guidelines for guessing the performance of these algorithms on graph families we have not tried (but which can be characterized). This rule has also been advocated in experimental papers [28]. Rule 4 guides us to study problems that are similar in nature to problems which would come up in applications. It also attempts to ensure that the experimentation provides a weighted average-case analysis for the algorithm, with higher weights placed on graphs that are more likely to be encountered in practice.

We also followed a guideline when implementing the algorithms. Although efficient implementations are important in experimental papers [27, 28], it is neither necessary nor desirable to attempt to tweak every last bit of performance out of each code. It is more useful to attempt to ensure that all the codes have a similar level of performance tweaking, as then the comparisons are meaningful. Accordingly, we followed this principle.

4.1.2 Problem Families

We choose several different problem families to meet different aspects of out goals.

Class Name	Brief Description
KARZ	Family Designed to be Difficult for Push-Relabel
KARZ_DENSE	Version of Karz with More Edges
RANDOM	Random
RANDOM_DENSE	Random with More Edges
SHADED_DENSITY	Number of Edges Declines With Distance from Source
CUBE_GRID	Full Grid; Grid Width, Length, and Number of Grids are Same
LONG_GRID	Full Grid; Number of Grids Greater than Grid Length and Width
WIDE_GRID	Full Grid; Grid Length and Width Greater than Number of Grids

4.1.2.1 Karz

Karz is borrowed from the implementation study by B. Cherkassky, A. Goldberg, P. Martin, J. Setubal, and J. Stolfi [6], with the simple modification that all of the edges are made undirected. The family takes parameters k, a, l, F, d, and S. The graph generated contains vertices divided into several groups: the source s, the sink t, S, X, Y, and P. The source is connected to every vertex in S, which contains k vertices. X and Y each contain a vertices; each vertex in S connected to F randomly chosen vertices in X and every vertex in X is connected to d randomly chosen vertices in Y. Finally, vertices in P are partitioned into k paths to the sink, where the ith path has $(i\text{-}1)*l + 1$ edges. Each vertex in Y is connected to the first node of every path. S is used as a random seed for the generator. Thus, a graph produced by the Karz generator has $2 + 2k + 2a + \frac{lk(k-1)}{2}$ vertices and $2k + Fk + ad + ak + \frac{lk(k-1)}{2}$ edges. This is much easier to understand visually; see Figure 4.1.

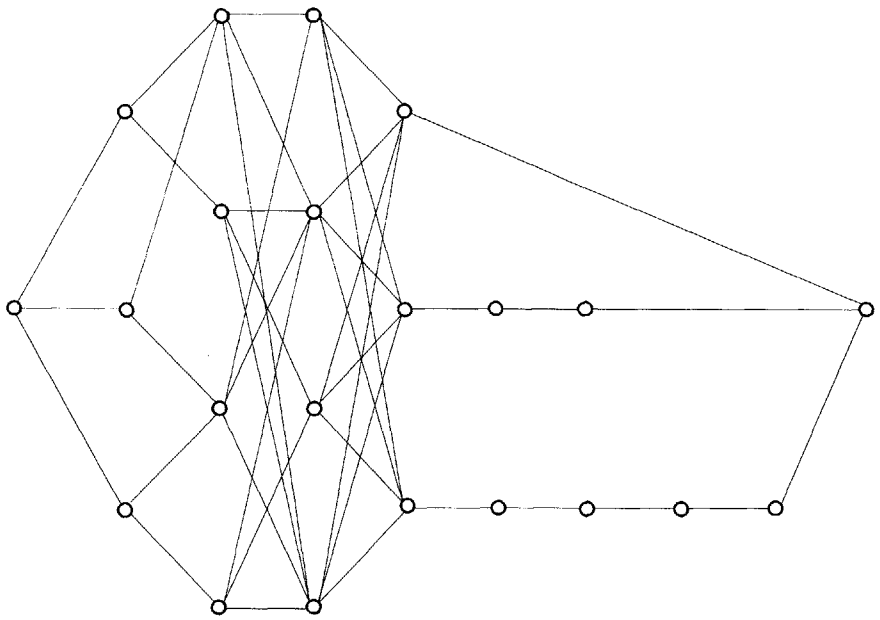


Figure 4.1 A graph generated by Karz with $k = 3$, $a = 4$, $l = 2$, $F = 2$, and $d = 3$.

We used the Karz generator to produce two families of graphs, KARZ and KARZ_DENSE. We give the values of the parameters used below.

Instance	***k***	***a***	***l***	***F***	***d***	***M***	***n***	***V***
KARZ_64K	32	1024	10	512	10	64416	7074	32
KARZ_167K	45	2025	10	1012	10	166905	14042	45
KARZ_454K	64	4096	10	2048	10	454464	28482	64
KARZ_1215K	90	8100	10	4050	10	1214730	56432	90
KARZ_DENSE_24K	32	128	10	96	96	24480	5282	32
KARZ_DENSE_79K	45	256	10	192	192	79302	10504	45
KARZ_DENSE_274K	64	512	10	384	384	274240	21314	64
KARZ_DENSE_988K	90	1024	10	768	768	987942	42280	90

4.1.2.2 Random

The Random graph generator takes two parameters, n and desired m. Using these, it computes the probability that an edge exists in the graph as $\frac{2m}{n(n-1)}$. It then chooses whether to create each edge in the graph with this probability. Self-edges are not permitted. Since edge-creation is random, the actual number of edges in the graph will not be quite the same as the desired m that is input. This means that different instantiations of the same graph instance will not have quite the same number of edges; the difference, however, should be negligible. We used the Random generator to produce two families of graphs, RANDOM and RANDOM_DENSE. We give the values of the parameters used below.

Instance	***desired m***	***N***
RANDOM_64K	64416	7074
RANDOM_167K	166905	14042
RANDOM_454K	454464	28482
RANDOM_1214K	1214730	56432
RANDOM_DENSE_24K	24480	5282
RANDOM_DENSE_79K	79302	10504
RANDOM_DENSE_274K	274240	21314
RANDOM_DENSE_988K	987942	42280

4.1.2.3 Shaded Density

The Shaded Density graph generator takes two parameters, *npl* and *l*. It then produces a graph with *l* layers and *npl* nodes per layer in addition to a source and sink. The source is connected to all the nodes in the first layer, and the sink is connected to all the nodes in the last layer. There

are no intra-layer connections in the graph, only inter-layer connections. The inter-layer connections are random. Each edge between the first two layers exists with probability 1; each edge between the last two layers exists with probability $1/npl$. The probability that an edge exists decreases linearly between these two probabilities as the layer in consideration is farther from the sink. In particular, for the edges between the *ith* and i+1th layers, each edge exists with probability $\left[\left\{\left(1-\frac{i-1}{l-2}\right)*\left(npl^2-npl\right)\right\}+npl\right]\Big/npl^2$ for $1 \le i \le l-1$. The graph produced for parameters *npl* and *l* will have $2+l*npl$ nodes and will be expected to have $2npl+\frac{\left(npl+npl^2\right)(l-1)}{2}$ edges.

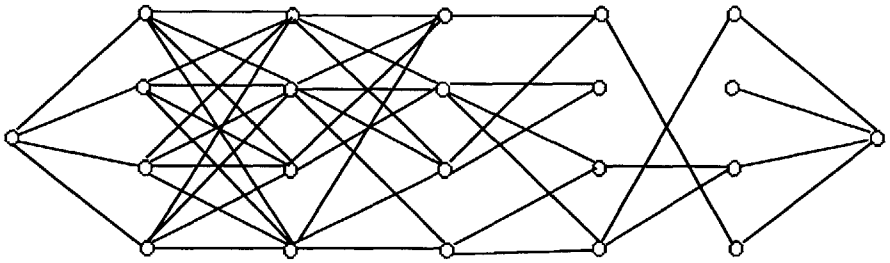


Figure 4.2 A graph produced by Shaded Density with $npl = 4$ and $l = 5$.

We used the Shaded Density graph generator to produce the SHADED_DENSITY problem family. The parameter values we used are listed below.

Instance	*Npl*	*l*	*Expected m*	*n*
SHADED_DENSITY_26K	32	50	25936	1602
SHADED_DENSITY_81K	57	50	81111	2852
SHADED_DENSITY_278K	106	50	278091	5302
SHADED_DENSITY_995K	201	50	995151	10052

4.1.2.4 Grid

Karz is borrowed from the implementation study by B. Cherkassky, A. Goldberg, P. Martin, J. Setubal, and J. Stolfi [6], with the simple modification that all of the edges are made undirected. The family takes parameters a, b, and c. It then generates a graph with b "frames," which are a by a grids with wrap-around at the edges of the grid. The source is connected to c random vertices in the first frame, and the sink is connected to c random vertices in the last frame. Between each pair of successive frames there are c random connections. A graph produced by this generator has $2 + a^2b$ nodes and $2a^2b + c(b + 1)$ edges. See figure 4.2 below for clarification.

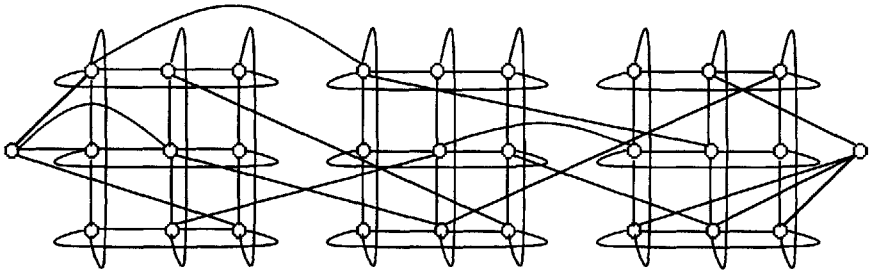


Figure 4.3 A graph created by Grid with $a = 3$, $b = 3$, and $c = 4$.

We used the Grid generator to produce three families of graphs, CUBE_GRID, LONG_GRID, and WIDE_GRID. We give the values of the parameters used below.

Instance	*A*	*b*	*c*	*m*	*n*
CUBE_GRID_62K	30	30	270	62370	27002
CUBE_GRID_171K	42	42	529	170923	74090
CUBE_GRID_450K	58	58	1009	449755	195114
CUBE_GRID_1180K	80	80	1920	1179520	512002
LONG_GRID_65K	10	258	50	64550	25802
LONG_GRID_167K	10	667	50	166800	66702
LONG_GRID_455K	10	1818	50	454550	181802
LONG_GRID_1215K	10	4859	50	1214800	485902
WIDE_GRID_65K	53	10	843	65453	28092
WIDE_GRID_168K	85	10	2168	168348	72252
WIDE_GRID_457K	140	10	5880	456680	196002
WIDE_GRID_1211K	228	10	15595	1211225	519842

4.1.3 Codes

As discussed in the implementation chapter, we experimented with many different approaches, far too many to report full results for. We will confine ourselves to reporting results for the codes listed below.

Code	Description
PR_FIFO	Push-relabel with first-in, first-out vertex selection heuristic
PR_HI	Push-relabel with highest level vertex selection heuristic
PR_LIFO	Push-relabel with last-in, first-out vertex selection heuristic
PR_LO	Push-relabel with lowest level vertex selection heuristic
AP_AR	Augment-relabel version of augmenting paths
AP_BS	Breadth-first search version of augmenting paths
AP_DS	Depth-first search version of augmenting paths
AP_LD	Label-directed search version of augmenting paths
DINIC	Dinic's Algorithm
GR	Goldberg-Rao sparse blocking flows algorithm
SAP_AR	Karger-Levine sparse augmenting paths algorithm with augment-relabel
SAP_BF	Karger-Levine sparse augmenting paths algorithm with blocking flows
SAP_BS	Karger-Levine sparse augmenting paths algorithm with breadth-first search
SAP_DS	Karger-Levine sparse augmenting paths algorithm with depth-first search
SAP_LD	Karger-Levine sparse augmenting paths algorithm with label-directed search
SBF_AR	Karger-Levine sparse blocking flows algorithm with augment-relabel
SBF_BF	Karger-Levine sparse blocking flows algorithm with blocking flows
SBF_BS	Karger-Levine sparse blocking flows algorithm with breadth-first search
SBF_DS	Karger-Levine sparse blocking flows algorithm with depth-first search
SBF_LD	Karger-Levine sparse blocking flows algorithm with label-directed search
KR_AR	Karger randomized augmenting paths algorithm with augment-relabel
KR_BF	Karger randomized augmenting paths algorithm with blocking flows
KR_BS	Karger randomized augmenting paths algorithm with breadth-first search
KR_DS	Karger randomized augmenting paths algorithm with depth-first search
KR_LD	Karger randomized augmenting paths algorithm with label-directed search

4.1.4 Setup

Our experiments were conducted on a machine with a 400MHz Pentium II processor, 384M of RAM, an integrated 512K Level 2 cache, and two (one for code, the other for data) 16K Level 1 caches. All of our codes are written in C++ and compiled with the GNU C++ compiler (g++) using the O4 optimization option. All codes were checked to ensure that the answers each yielded for a given flow problem were the same.

In problem families that are created using randomization, for each setting of parameters of the problem family we created five problems, each with a different random seed. We used the same set of five random seeds for all problems created by a generator to attempt to ensure that the only variable changing was size, not the "luckiness" of the particular graph constructed. We do not report results for instances that took over 1800 seconds to run.

For all algorithms we report the average running time of the code and the average number of arc fetches for each setting of parameter values within a problem family. We also provide standard deviations as a percent of the average for each of these quantities. We also provide some information specific to certain algorithms. For Dinic's algorithm, we report the number of blocking flows needed to reach the solution. For the Goldberg-Rao sparse blocking flows algorithm, we report the number of blocking flows needed to reach the solution, and also report the split of average running time and average arc fetches between blocking flows and sparsification.

For the Karger-Levine sparse augmenting paths algorithm, we report the split of average running time and average arc fetches between the augmenting paths code and the sparsification code. We also report the number of time SPARSEAUGMENT2 is called and the number of times AUGPATH is called (remember that each call to AUGPATH finds all the augmenting paths in the graph it is called on).

For the Karger-Levine sparse blocking flows algorithm, we report the same information as the Karger-Levine sparse augmenting paths algorithm, except the split of the averages is between sparsification, augmenting paths **and** blocking flows. We also report the number of blocking flows needed.

Finally, for the Karger randomized augmenting paths algorithm, we report the split of average running time and average arc fetches between the augmenting paths routine and the code which prepares the data structures for the recursion.

4.2 Results

In this section we discuss our results. We give more details by first discussing the results on each problem family and then discussing the results from each algorithm.

Most of the data in this section is in the form of plots; for exact numbers refer to the appendix. The plots have log(*running time*) on the vertical axis, and log(m) on the horizontal axis. This allows us to determine the asymptotic running time of the algorithm (slope of the line in the plot) as well as the constant factor (y-intercept). We also use linear regression to find best-fit lines and report the slope and y-intercept. We use the abbreviation K-L to stand for Karger-Levine.

Since we cannot plot 0.00 on a log scale, we plotted it as 0.01. Our timer is not precise for small values.

4.2.1 Results by Problem Family

Here we examine the results for each of the four problem generators we used, Karz, Random, Shaded Density, and Grid. For each problem family, we dedicate a graph to all the push-relabel approaches, another graph to all the augmenting paths approaches, a third graph to all of the variants of Karger-Levine sparse augmenting paths, a fourth graph to all of the variants of Karger-Levine sparse blocking flows, and a fifth graph to all of the variants of Karger randomized augmenting paths. We then use a sixth graph to graph the best algorithm from each of the five previous graphs along with Dinic's algorithm and Goldberg-Rao sparse blocking flows. We also provide the slope and intercept from regressions of the runtime and the number of edges.

4.2.1.1 Karz

The Karz generator was specifically designed to be bad for push-relabel algorithms; hence, it comes as little surprise that these algorithms perform fairly badly on the graphs produced by this generator. As described earlier, we used it to generate two families of graphs, KARZ and KARZ_DENSE, which were identical in structure except for the number of edges between the first two layers. We used these families both to examine a bad case for the push-relabel algorithm and to see whether varying edge density would make the sparsifying algorithms gain in effectiveness as compared to their peers. We first present discussion, graphs, and tables and for KARZ, then provide the same for KARZ_DENSE, and finally discuss our observations formed from the two families.

The push-relabel algorithms all have difficulty with the KARZ family, although the highest-label vertex selection heuristic performs noticeably better than the rest because its selection heuristic happens to function quite well on this graph. The augmenting paths codes display similar asymptotic growth rates but significantly different constant factors. In particular, we see that the augment-relabel approach does not work nearly as well as the other three approaches. This is likely due to its similarities to the push-relabel algorithm. It is interesting to note that the label-directed search algorithm, which is also similar to the push-relabel algorithms but obeys the labeiling less strictly then the augment-relabel algorithm, performs noticeably better than the augment-relabel algorithm. Since label-directed search does not follow the labels as strictly, it may often perform respectively problem instances that are bad for push-relabel (which usually involve "tricking" the algorithm so the labels steer the algorithm in the wrong direction).

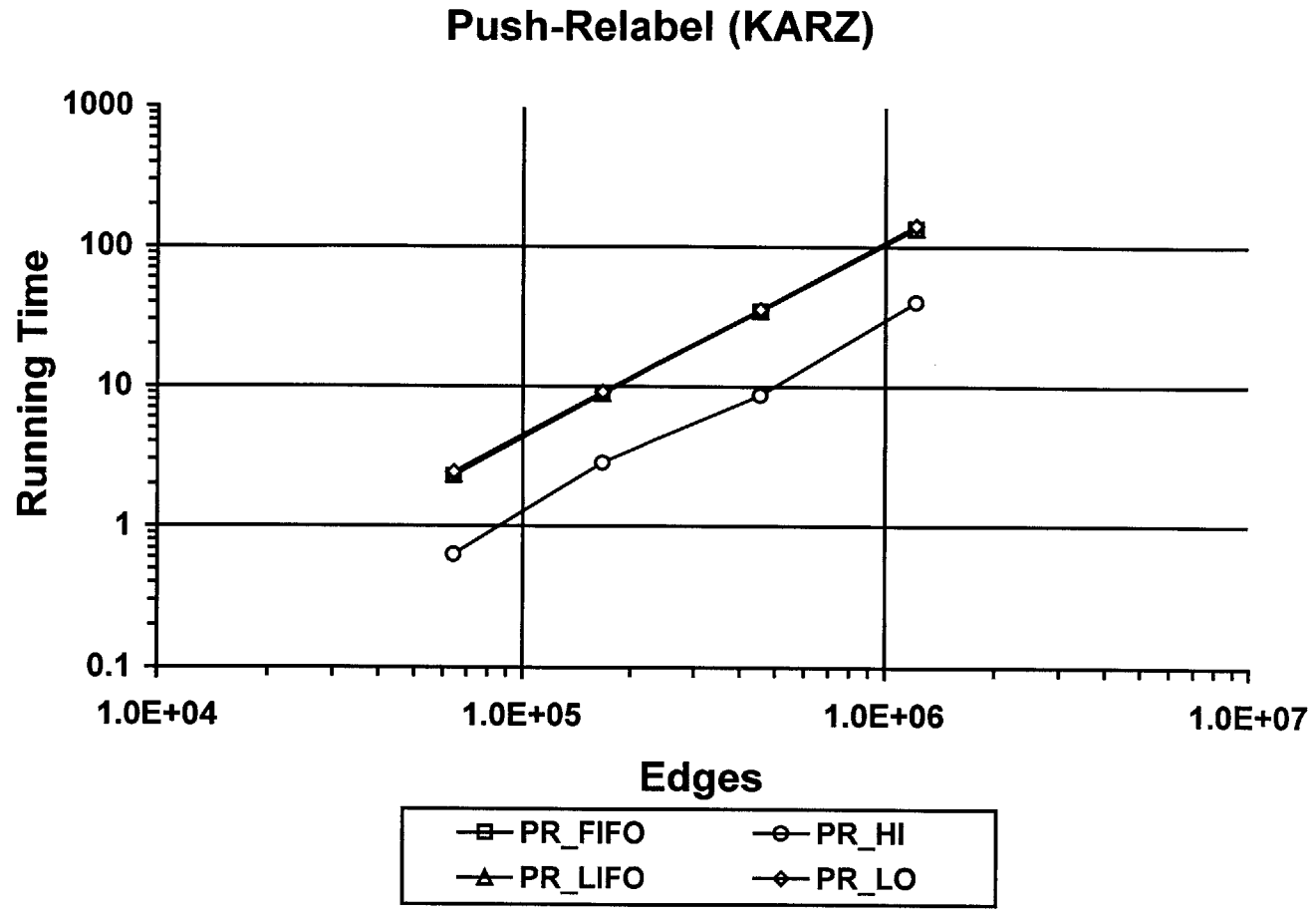
The results from the Karger-Levine sparse augmenting paths algorithms indicate that sparsification may be practically useful. Although the performance of the sparse versions is often slower than that of their respective plain vanilla augmenting paths algorithms, the growth rate for all of the sparsifying versions are better than their non-sparsifying counterparts. Furthermore, we see that, with the exception of the depth-first search version (which, unfortunately, is the best

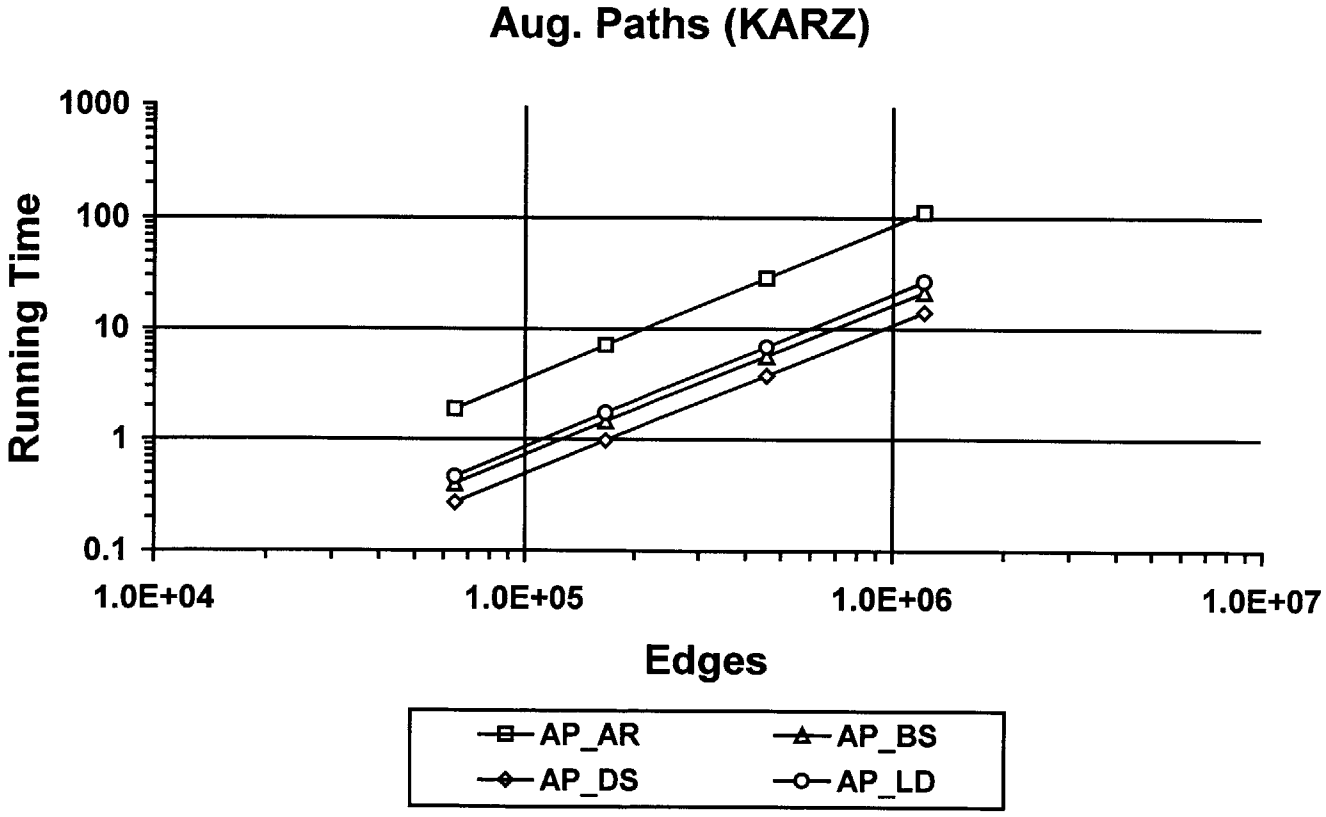
version), the sparse augmenting paths implementations outperform their respective plain augmenting paths counterparts versions within the range of test data we tried.

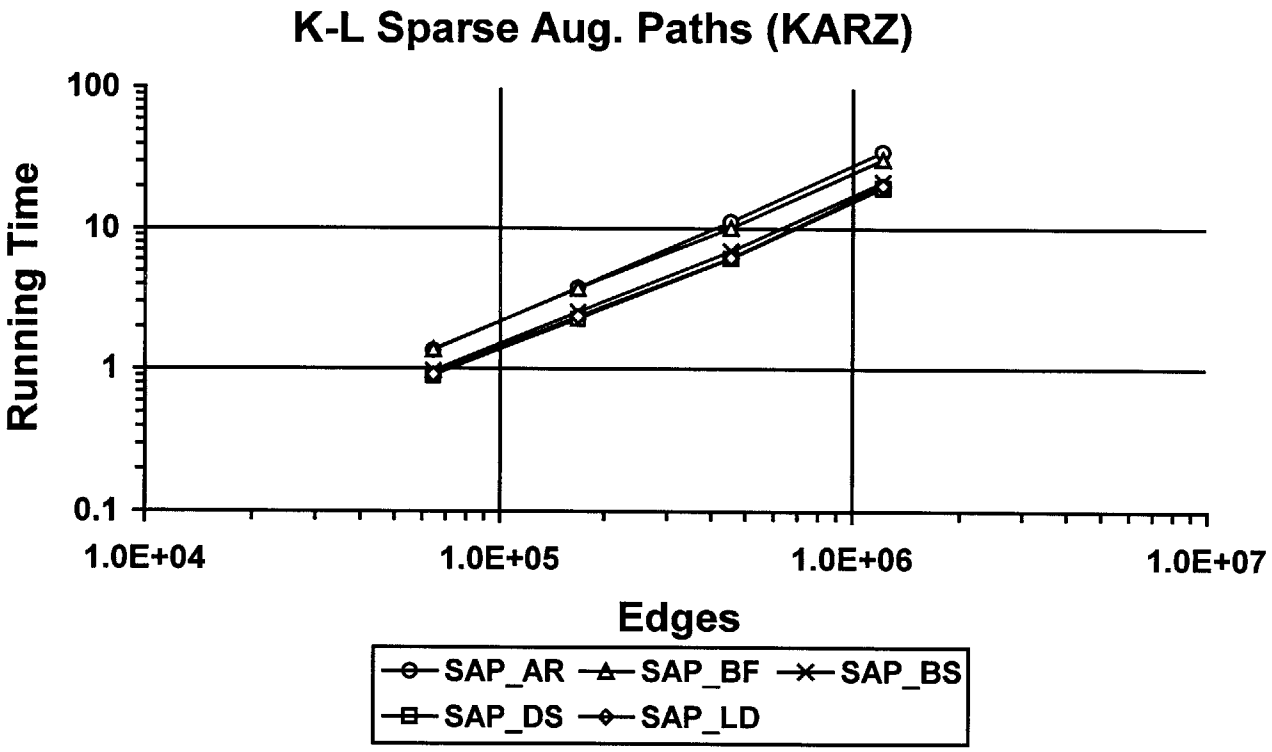
The results for the Karger-Levine sparse blocking flows algorithms seem confusing at first. However, a review of the pseudocode for the algorithm shows that whenever $v < n/m^{1/3}$, the augmenting paths codes will not be invoked, and thus all the variants of the algorithm will be exactly the same (except for small differences in the data structures which are necessary to use the different augmenting paths approaches). Moreover, it comes as no surprise that the Karger-Levine sparse blocking flow algorithms perform about the same as Dinic's algorithm.

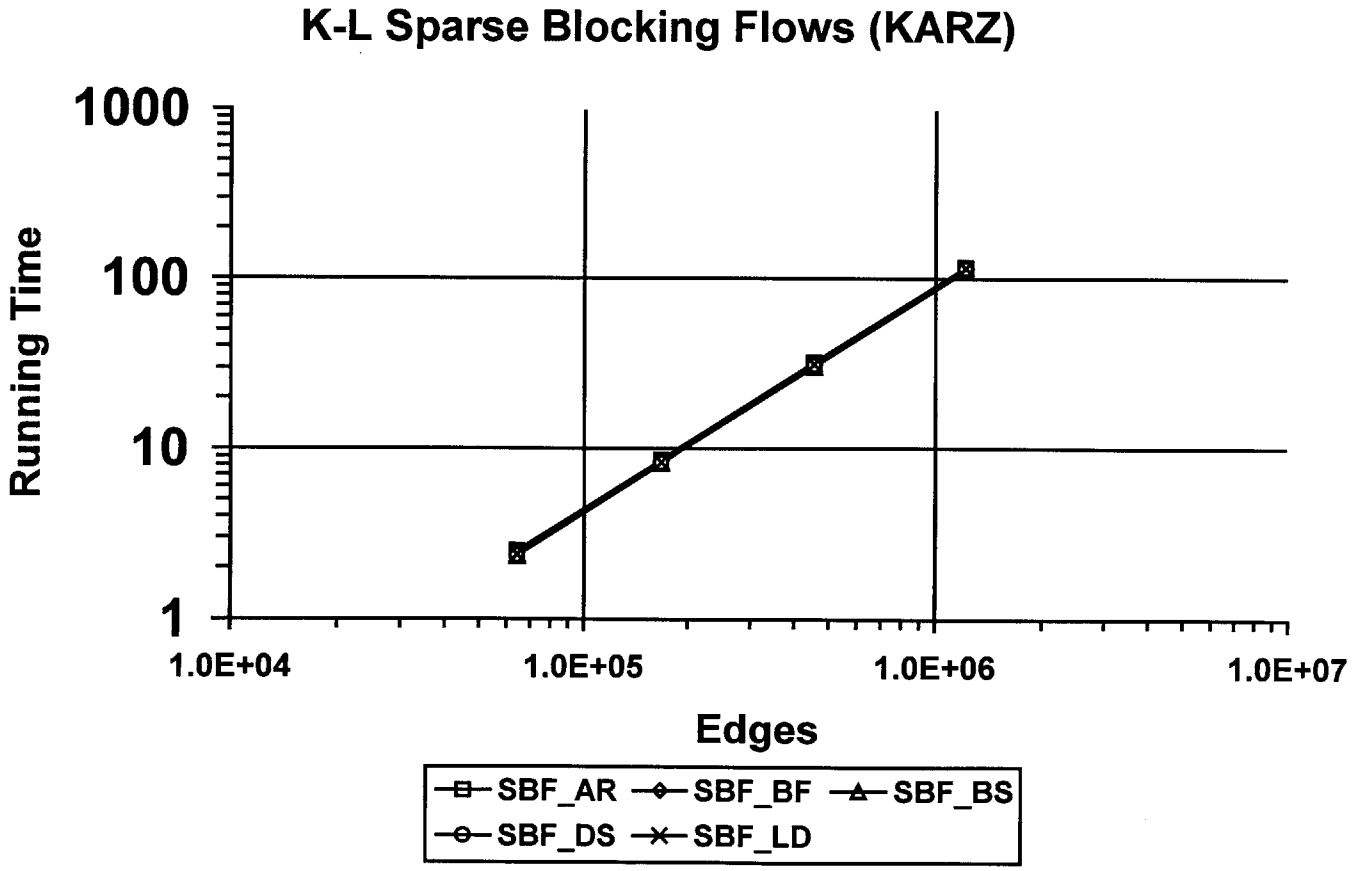
The Karger randomized algorithms do not perform as well as the Karger-Levine sparse augmenting paths algorithms on the KARZ family because the long, narrow paths in the graph make it unlikely that a path will be present in the graph during the lower levels of the recursion. Nonetheless, the randomized algorithms have noticeably better growth rates than their plain vanilla augmenting paths counterparts. Also worthy of note is the large gap between the augment-relabel and blocking flow approaches and the other three approaches. Although the reason behind this sizeable gap is not immediately clear, we speculate it is because the performance of the augment-relabel and blocking flow approaches are better when the graph has many flow paths (although it is surprising that this would be true for augment-relabel and not label-directed search).

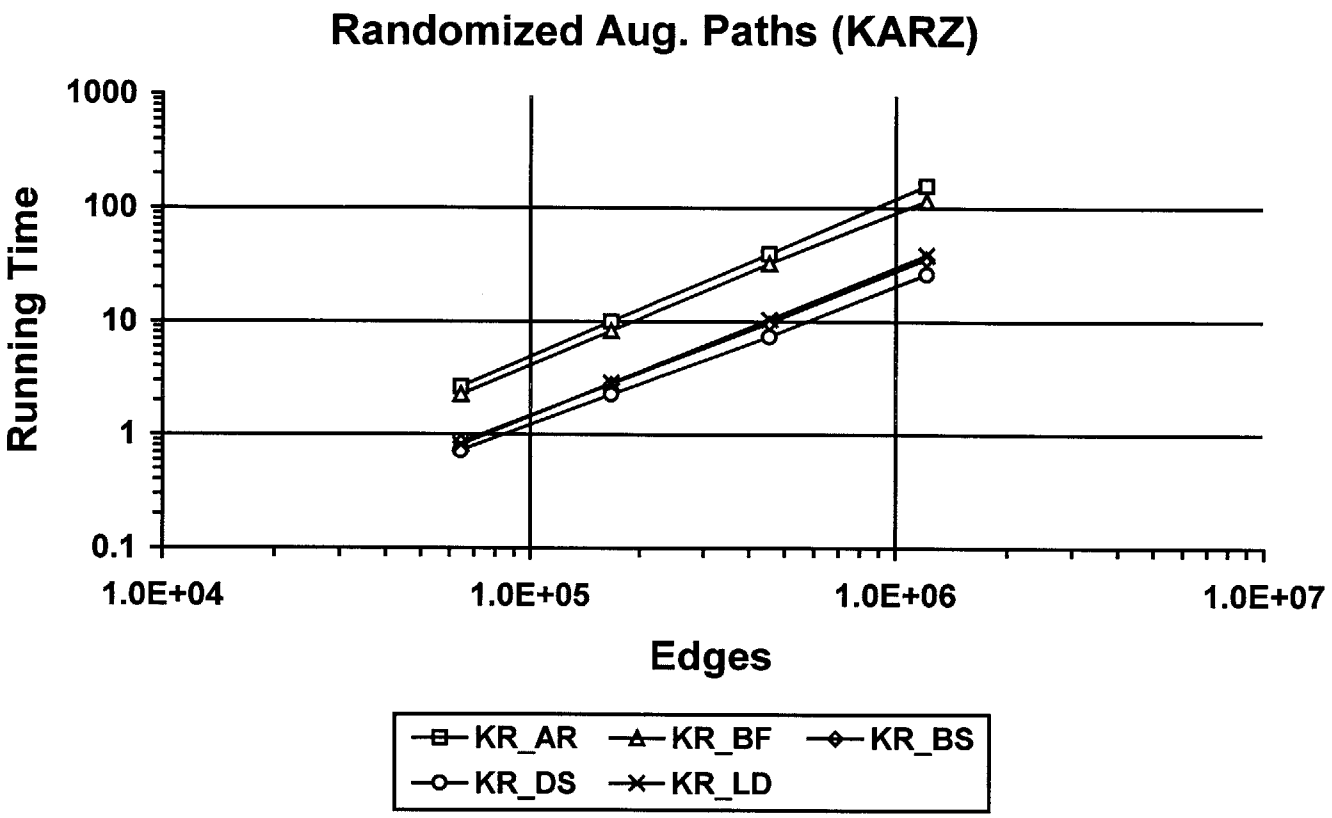
Looking at the overall picture, we see that the Dinic, Goldberg-Rao, and Karger-Levine sparse blocking flow approaches display almost identical poor performances. Also, although the best asymptotic growth rates belong to the sparsified augmenting paths approaches (with the Karger-Levine version being much better than the Karger randomized version), the constant factors on these algorithms are so large that the push-relabel and plain augmenting paths approaches have competitive running times.

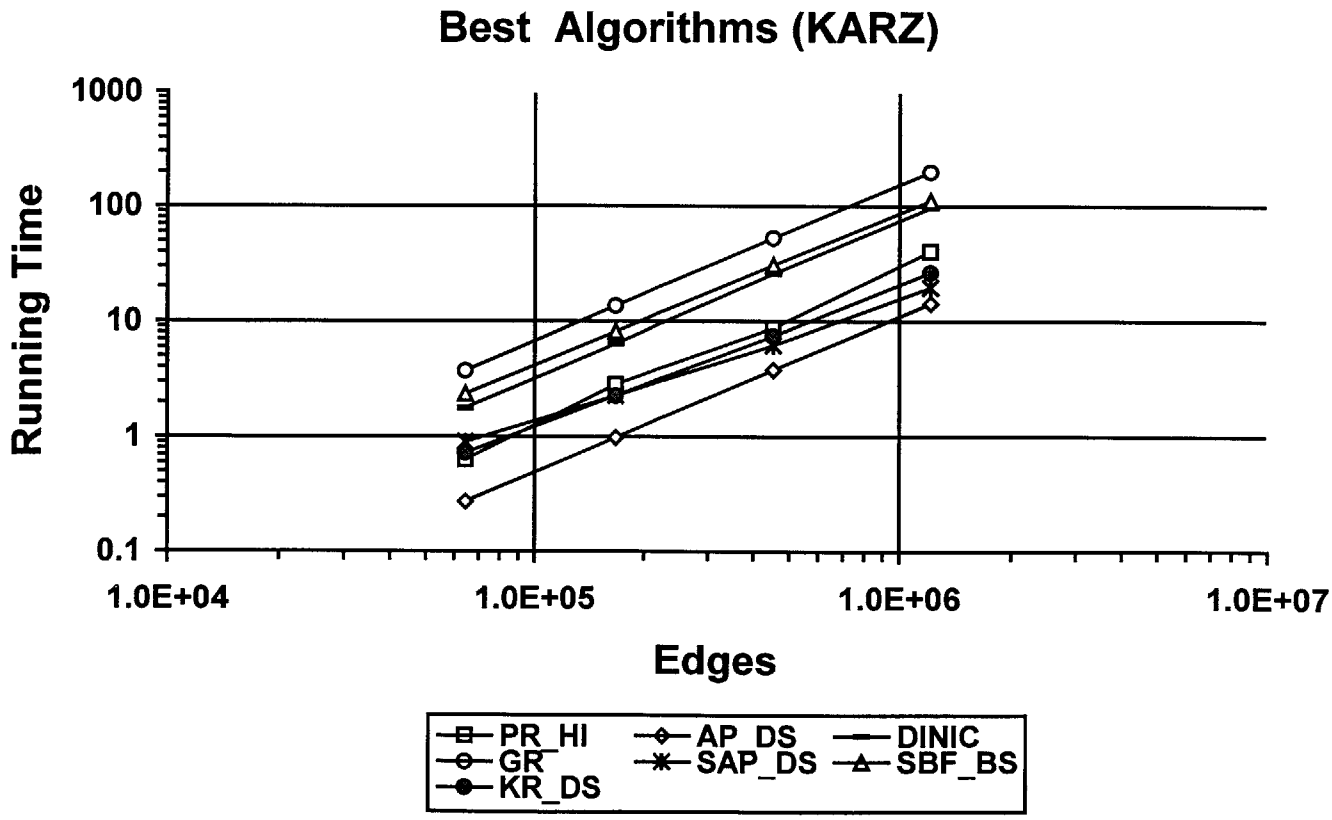

Push-Relabel (KARZ)
1000
100
10
1
0.1
Running Time
1.0E+04
1.0E+05
1.0E+06
1.0E+07
Edges
PR_FIFO
PR_HI
PR_LIFO
PR_LO


Aug. Paths (KARZ)
1000
100
10
1
0.1
Running Time
1.0E+04
1.0E+05
1.0E+06
1.0E+07
Edges
AP_AR
AP_BS
AP_DS
AP_LD


K-L Sparse Aug. Paths (KARZ)
Running Time
100
10
1
0.1
1.0E+04
1.0E+05
1.0E+06
1.0E+07
Edges
SAP_AR
SAP_BF
SAP_BS
SAP_DS
SAP_LD


K-L Sparse Blocking Flows (KARZ)
Running Time
1000
100
10
1
1.0E+04
1.0E+05
1.0E+06
1.0E+07
Edges
SBF_AR
SBF_BF
SBF_BS
SBF_DS
SBF_LD


Randomized Aug. Paths (KARZ)
Running Time
1000
100
10
1
0.1
1.0E+04
1.0E+05
1.0E+06
1.0E+07
Edges
KR_AR
KR_BF
KR_BS
KR_DS
KR_LD


Best Algorithms (KARZ)
Running Time
1000
100
10
1
0.1
1.0E+04
1.0E+05
1.0E+06
1.0E+07
Edges
PR_HI
AP_DS
DINIC
GR
SAP_DS
SBF_BS
KR_DS

Code	Running Time on KARZ
PR_FIFO	$4.84 \times 10^{-7} \times m^{1.39}$
PR_HI	$1.46 \times 10^{-7} \times m^{1.38}$
PR_LIFO	$5.33 \times 10^{-7} \times m^{1.38}$
PR_LO	$5.46 \times 10^{-7} \times m^{1.38}$
AP_AR	$3.98 \times 10^{-7} \times m^{1.39}$
AP_BS	$1.24 \times 10^{-7} \times m^{1.35}$
AP_DS	$9.18 \times 10^{-8} \times m^{1.35}$
AP_LD	$1.05 \times 10^{-7} \times m^{1.38}$
DINIC	$4.87 \times 10^{-7} \times m^{1.36}$
GR	$1.13 \times 10^{-6} \times m^{1.36}$
SAP_AR	$6.47 \times 10^{-6} \times m^{1.10}$
SAP_BF	$1.13 \times 10^{-5} \times m^{1.06}$
SAP_BS	$8.39 \times 10^{-6} \times m^{1.05}$
SAP_DS	$8.56 \times 10^{-6} \times m^{1.04}$
SAP_LD	$8.41 \times 10^{-6} \times m^{1.04}$
SBF_AR	$1.25 \times 10^{-6} \times m^{1.31}$
SBF_BF	$1.31 \times 10^{-6} \times m^{1.30}$
SBF_BS	$1.10 \times 10^{-6} \times m^{1.32}$
SBF_DS	$1.02 \times 10^{-6} \times m^{1.32}$
SBF_LD	$1.03 \times 10^{-6} \times m^{1.32}$
KR_AR	$5.64 \times 10^{-7} \times m^{1.39}$
KR_BF	$8.28 \times 10^{-7} \times m^{1.34}$
KR_BS	$6.43 \times 10^{-7} \times m^{1.27}$
KR_DS	$9.67 \times 10^{-7} \times m^{1.22}$
KR_LD	$3.97 \times 10^{-7} \times m^{1.31}$

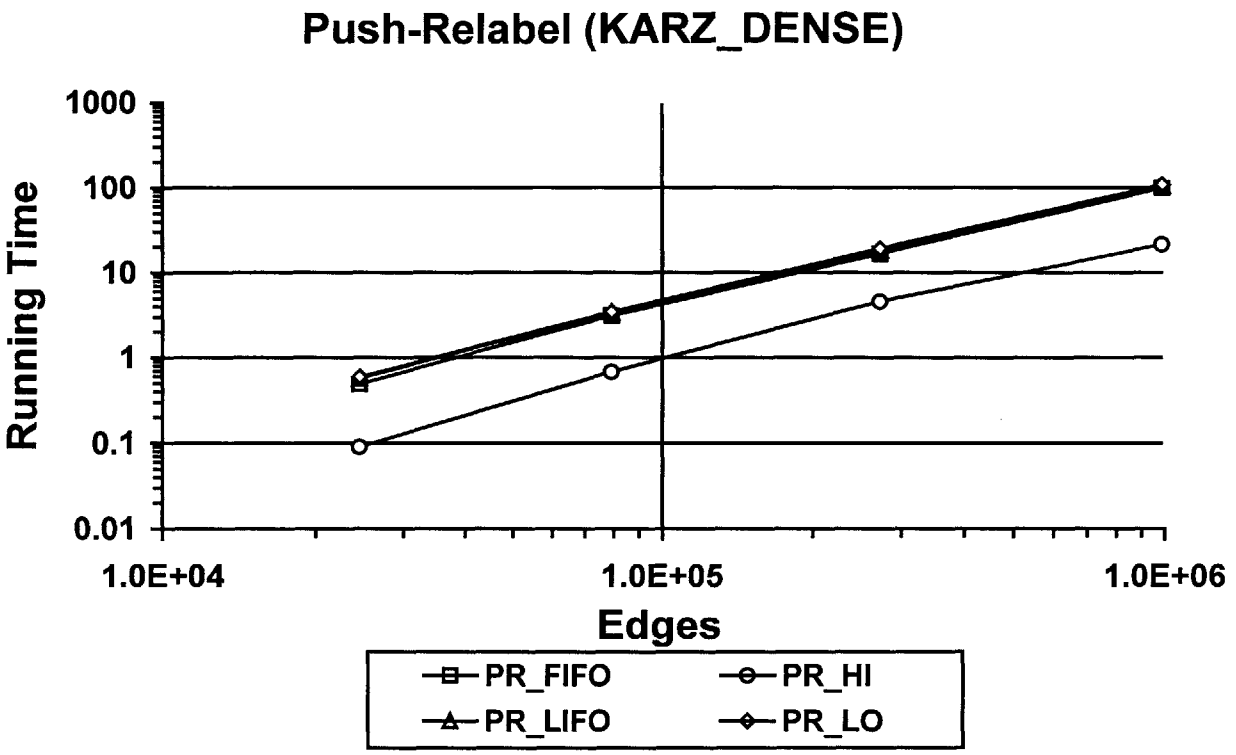
For the KARZ_DENSE family, we see that highest-level vertex selection once again proves to be the best push-relabel approach by a sizeable margin and that the other push-relabel approaches perform virtually identically. The augmenting paths algorithms show widely varied performance (both in terms of constants and growth) with label-directed search taking the lead and augment-relabel bringing up the rear. This is very interesting because these two heuristics are fairly similar; the difference in performance may again be attributable to the "strictness" with which the distance labels are followed. It is also interesting to note that although the distance labelling used by label-directed search is not as powerful as it often is (as indicated by the difficulties of the push-relabel algorithms), the label-directed search still outperforms depth-first search and breadth-first search.

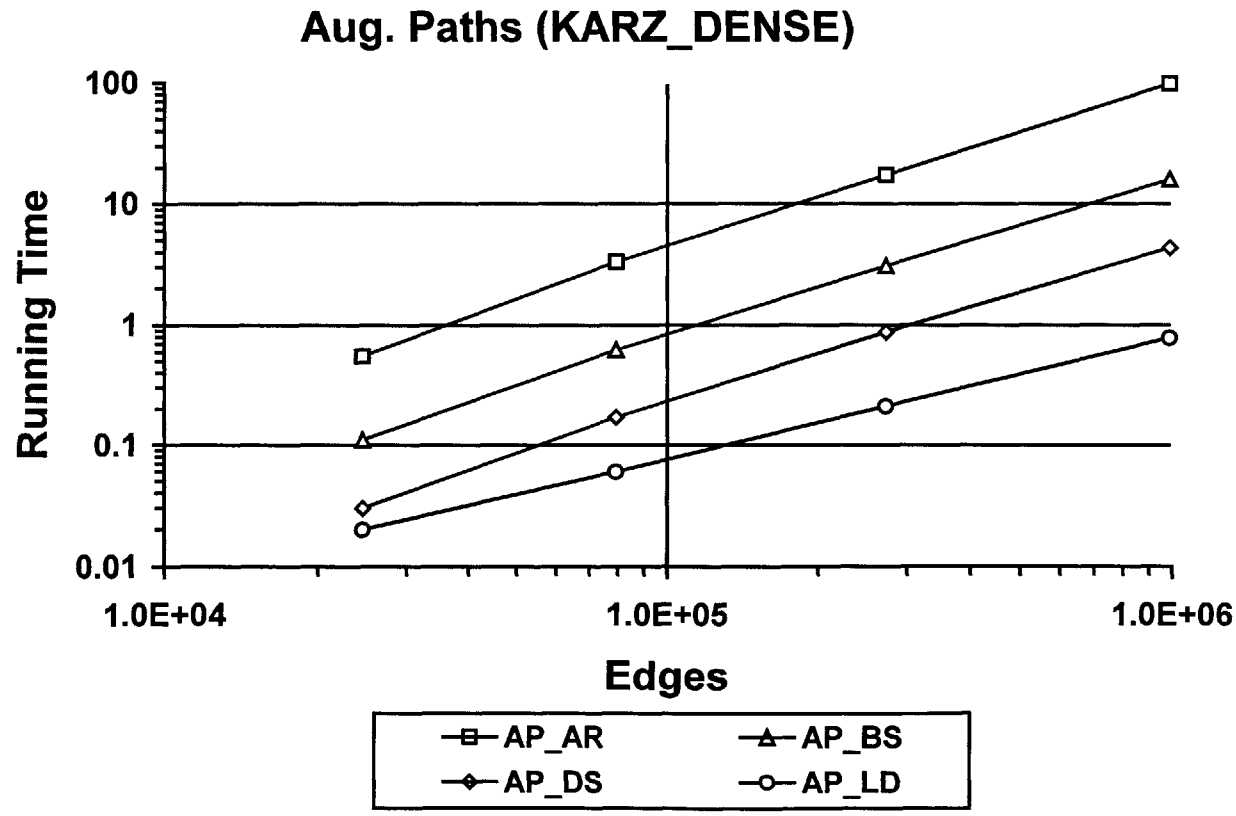
The Karger-Levine sparse augmenting paths implementations display much better asymptotic growth rates than their respective plain-vanilla augmenting paths counterparts and also, interestingly, show much smaller differences between the different implementations. Unfortunately, the running time constants for the Karger-Levine sparse augmenting paths approaches are large. The Karger-Levine sparse blocking flows algorithms all show almost exactly the same performance, as the flow values are not high enough for the execution of the algorithms to differ.

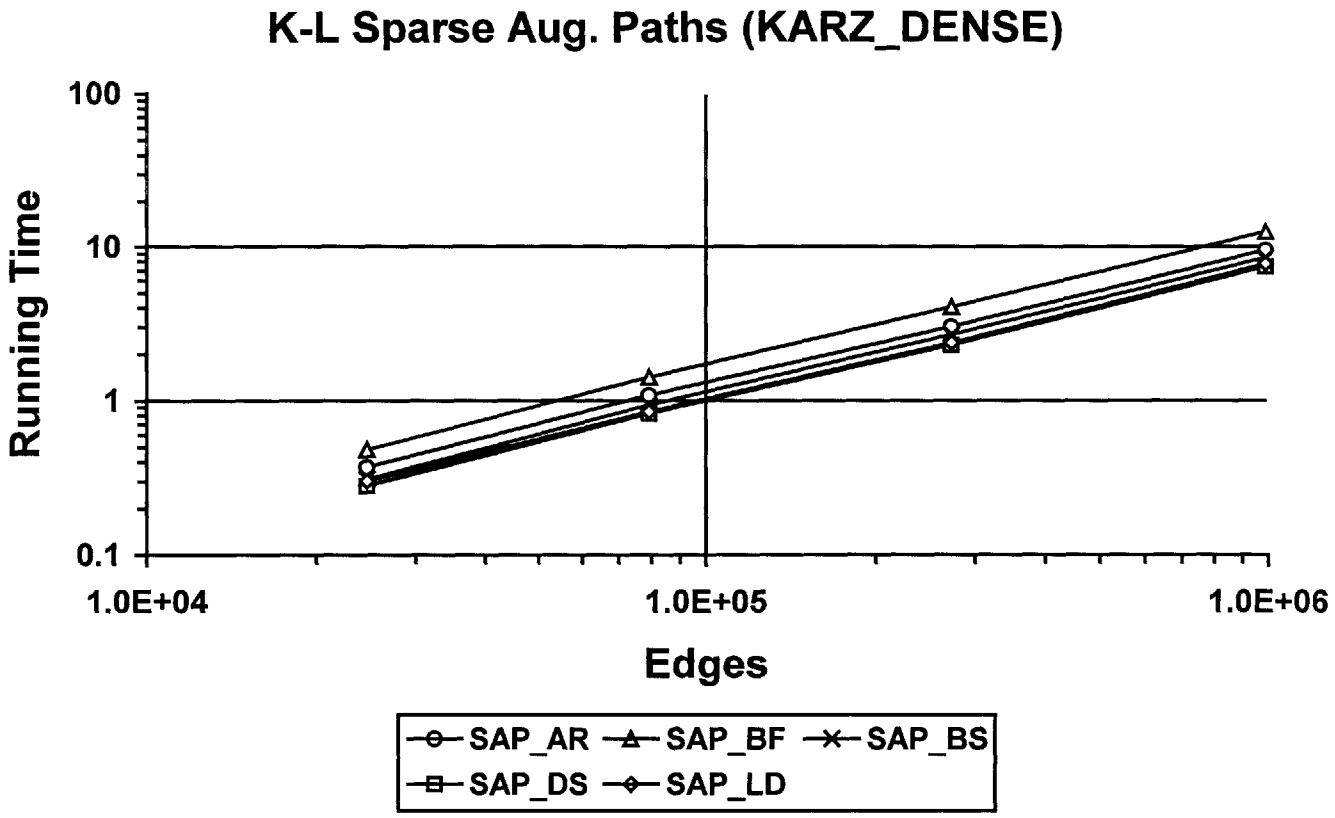
Finally, the Karger randomized augmenting paths algorithms exhibit better asymptotic growth rates than the simple augmenting paths algorithms but not as good as those displayed by the Karger-Levine sparse augmenting paths algorithms. Interestingly, unlike the Karger-Levine sparse augmenting paths algorithms, the differences in the different implementations do not disappear in the Karger randomized augmenting paths versions. Based on this, it seems reasonable to speculate that the Karger-Levine sparse augmenting paths algorithms preserve the structure of the graph much less than the Karger randomized augmenting paths algorithms. It would be interesting to determine to what extent this is true (and how much influence it exerts on the performance of the algorithms). It seems possible that the main effect of sparsification may not be in the sheer reduction in the number of edges, but rather in the alteration of the graph structure (equivalently, the removal of "confusing" edges).

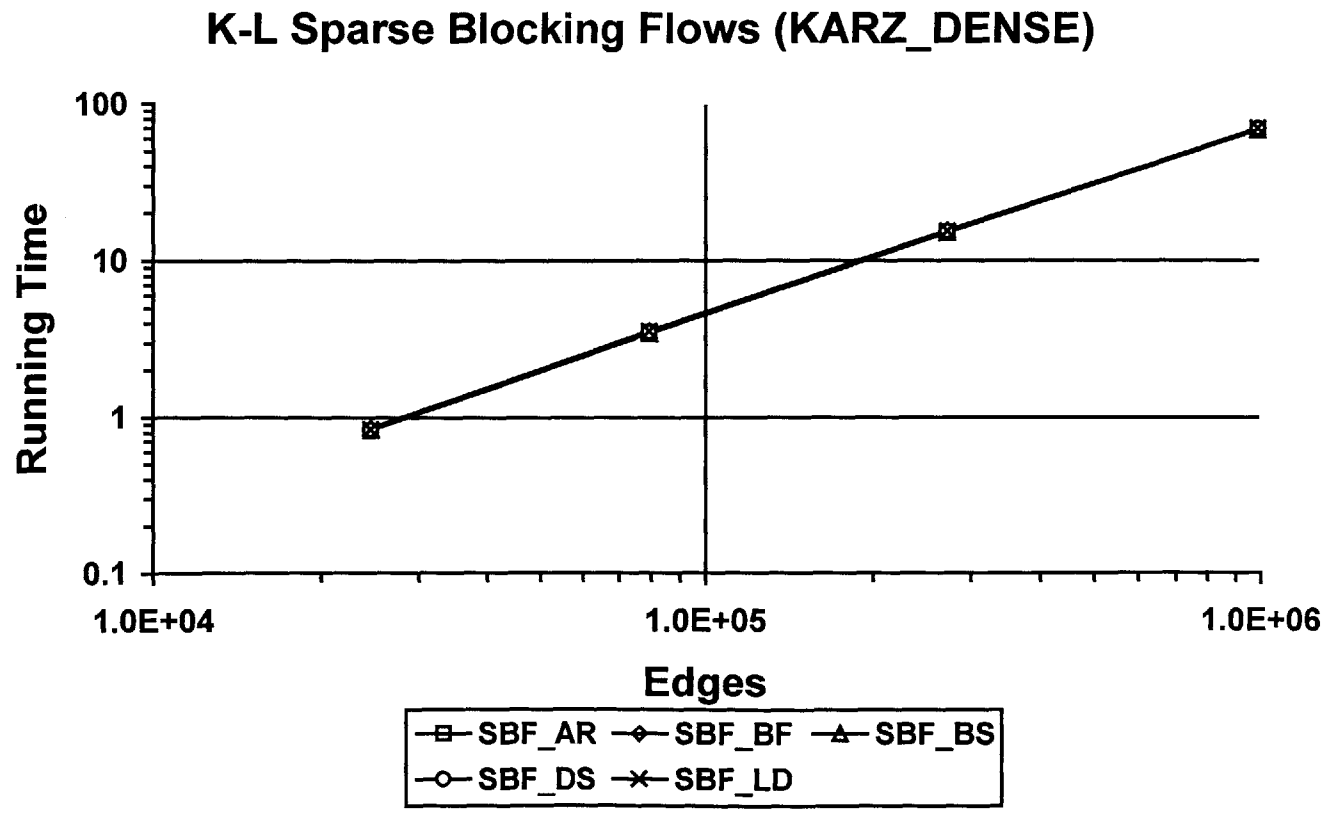
The big picture, then, shows label-directed search as the best algorithm over the range of sizes we covered, but the Karger-Levine sparse depth-first search augmenting paths and Karger randomized label-directed search augmenting paths algorithms seem certain to become the best, in that order, for larger graphs. It is interesting to note that the best subroutine for the three approaches is not the same; it is not clear why this is true nor is it clear whether this happens often or is quirk of this particular problem family. It is interesting to speculate whether the increase in simplicity of the graph structure resulting from sparsification (discussed in the previous paragraph) means that a more complex augmenting paths algorithm (i.e. label-directed search) merely performs extra work without much extra benefit, thus leading to the difference in preferable subroutines.

Examining the two Karz families together is confusing. A natural expectation would be that the denser family would be harder for the algorithms. This, however turns out to not be the case for many of the algorithms; for the label-directed search algorithms as well as all of the Karger-Levine sparse augmenting paths implementations and all of the Karger randomized augmenting paths implementations, the increased edge density actually makes the problem easier. As a result, the gap in performance between the augmenting paths algorithms which use some form of sparsification and those that do not widens, with the exception of the algorithms using label-directed search (where the gap narrows). It is not clear what to make of this result; it may simply be an indication that our attempt to characterize the running time as a function of the number of edges (as opposed to a function of both the number of edges and number of nodes) is an incorrect oversimplification. It may also indicate that the performance of many of the algorithms is not effected by the sheer number of edges as much as by the difficulty of finding flow paths, thus raising the possibility that adding more edges makes it easier to find flow paths and thus makes the problem easier. Thus, the overall role that edge density plays is not clear and, moreover, it is not clear whether these results are indicative of many problem families or specific to those produced by Karz.

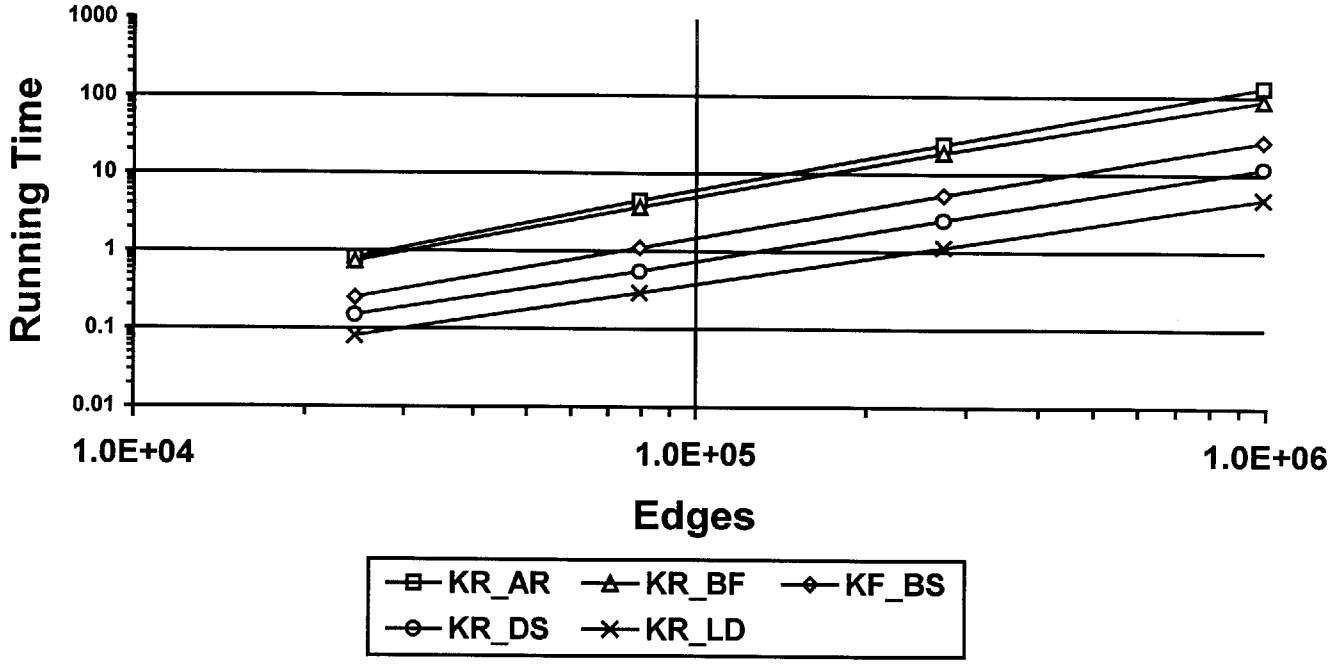

Push-Relabel (KARZ_DENSE)
Running Time
1000
100
10
1
0.1
0.01
1.0E+04
1.0E+05
1.0E+06
Edges
PR_FIFO
PR_HI
PR_LIFO
PR_LO


Aug. Paths (KARZ_DENSE)
Running Time
100
10
1
0.1
0.01
1.0E+04
1.0E+05
1.0E+06
Edges
AP_AR
AP_BS
AP_DS
AP_LD

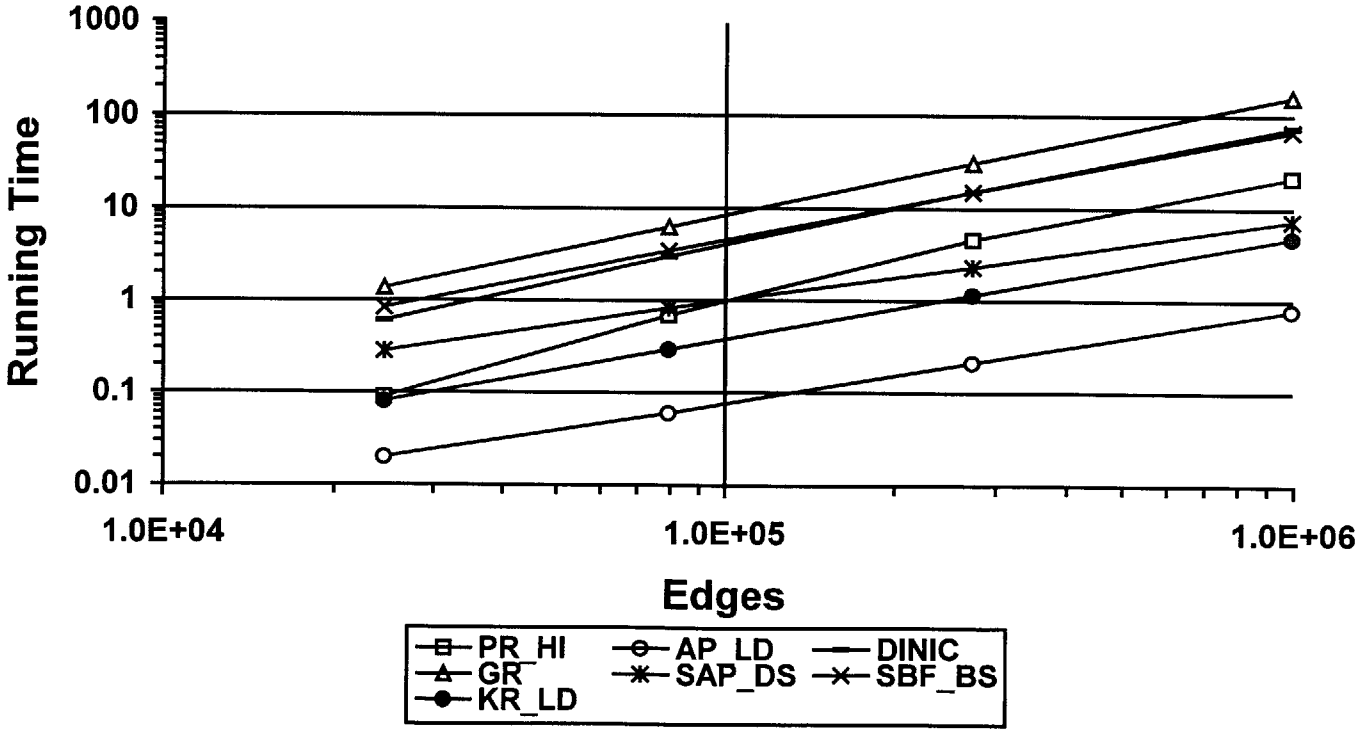

K-L Sparse Aug. Paths (KARZ_DENSE)
Running Time
100
10
1
0.1
1.0E+04
1.0E+05
1.0E+06
Edges
SAP_AR
SAP_BF
SAP_BS
SAP_DS
SAP_LD


K-L Sparse Blocking Flows (KARZ_DENSE)
Running Time
100
10
1
0.1
1.0E+04
1.0E+05
1.0E+06
Edges
SBF_AR
SBF_BF
SBF_BS
SBF_DS
SBF_LD

Randomized Aug. Paths (KARZ_DENSE)



Best Algorithms (KARZ_DENSE)



Code	Running Time on KARZ_DENSE
PR_FIFO	$2.82 \times 10^{-7} \times m^{1.43}$
PR_HI	$3.21 \times 10^{-8} \times m^{1.48}$
PR_LIFO	$4.09 \times 10^{-7} \times m^{1.41}$
PR_LO	$4.10 \times 10^{-7} \times m^{1.41}$
AP_AR	$4.39 \times 10^{-7} \times m^{1.40}$
AP_BS	$1.54 \times 10^{-7} \times m^{1.34}$
AP_DS	$4.22 \times 10^{-8} \times m^{1.34}$
AP_LD	$8.46 \times 10^{-7} \times m^{.99}$
DINIC	$1.19 \times 10^{-6} \times m^{1.30}$
GR	$3.20 \times 10^{-6} \times m^{1.28}$
SAP_AR	$5.49 \times 10^{-5} \times m^{.87}$
SAP_BF	$6.81 \times 10^{-5} \times m^{.88}$
SAP_BS	$3.86 \times 10^{-5} \times m^{.89}$
SAP_DS	$3.85 \times 10^{-5} \times m^{.88}$
SAP_LD	$4.31 \times 10^{-5} \times m^{.88}$
SBF_AR	$5.02 \times 10^{-6} \times m^{1.19}$
SBF_BF	$5.46 \times 10^{-6} \times m^{1.19}$
SBF_BS	$5.06 \times 10^{-6} \times m^{1.19}$
SBF_DS	$4.96 \times 10^{-6} \times m^{1.19}$
SBF_LD	$5.02 \times 10^{-6} \times m^{1.19}$
KR_AR	$8.00 \times 10^{-7} \times m^{1.37}$
KR_BF	$1.55 \times 10^{-6} \times m^{1.30}$
KR_BS	$7.31 \times 10^{-7} \times m^{1.26}$
KR_DS	$8.49 \times 10^{-7} \times m^{1.19}$
KR_LD	$1.08 \times 10^{-6} \times m^{1.11}$

4.2.1.2 Random

As discussed earlier, we used the Random generator to produce two families of graphs, RANDOM and RANDOM_DENSE. We will examine the results for RANDOM, consider RANDOM_DENSE, and finally attempt to draw conclusions relevant to both families.

The RANDOM graphs proved to be relatively easy for all of the algorithms, probably due to their relatively small flow values (our 1.2 million edge graphs had a meager average flow value of 39). All of the push-relabel algorithms ran in about the same time on this problem; PR_LO was the best of the bunch by a hair. Among the augmenting paths algorithms the approaches with global relabelling, AP_AR and AP_LD, performed better than the others. It does appear, however, that AP_DS would have eventually passed the other two algorithms; its rate of growth is better.

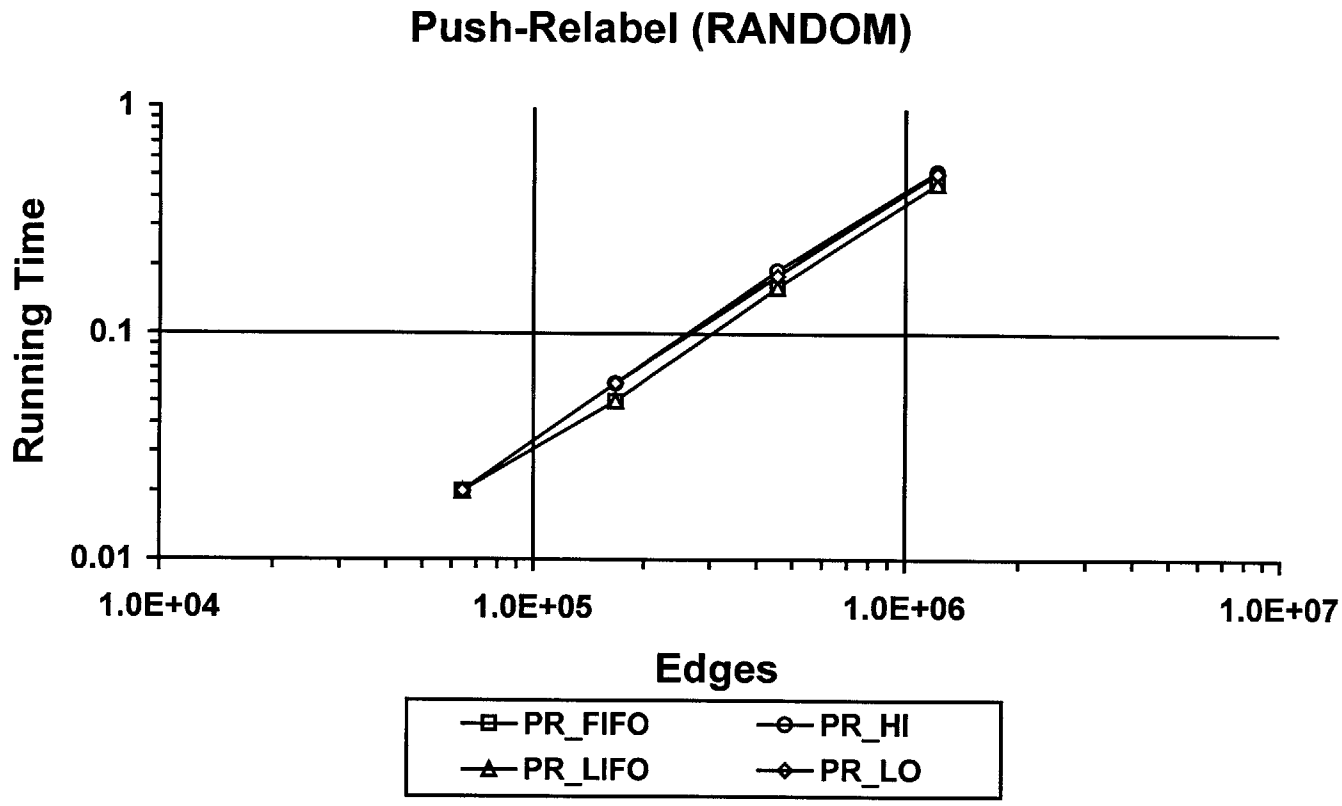
The sparsification-based algorithms (both Karger-Levine sparse augmenting paths and Karger randomized augmenting paths) generally do not do well; both the growth rates of their running times and their constant factors are higher than those for the plain augmenting paths algorithms. This is not only because they perform a fair amount of work to achieve a fairly marginal reduction in the number of edges, but also because the sparsification seems to make the flow-carrying paths harder for the algorithms to find, thus making the running times of the actual flow-finding parts of the algorithms larger than they are without sparsification. We also observe that again the sparsification decreases the difference between the different versions of augmenting paths subroutines.

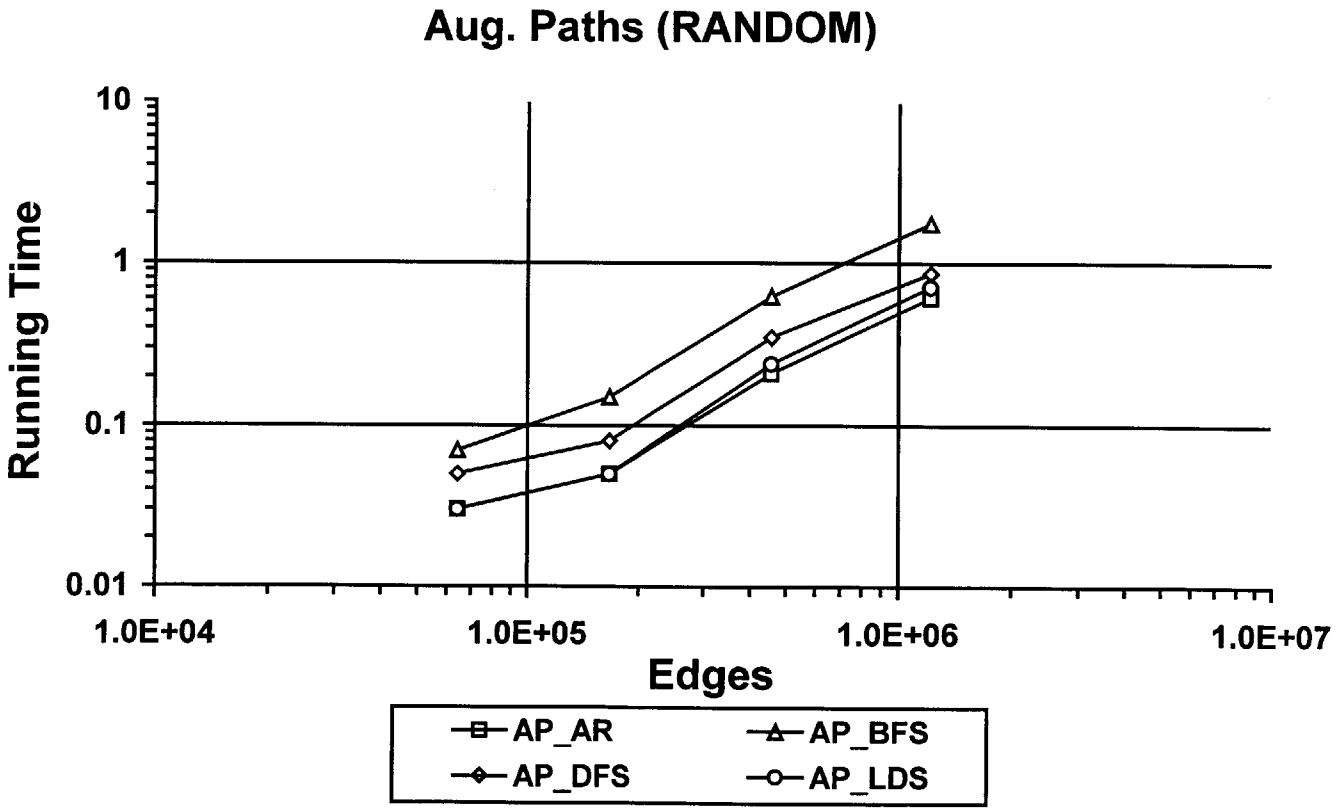
The Karger-Levine sparse blocking flows algorithms are uninteresting; once again the flow value is too low to cause the augmenting paths code to run so all the versions perform the same as Dinic's algorithm. All of our blocking flow algorithms (Dinic's, Goldberg-Rao sparse blocking flows, and Karger-Levine sparse blocking flows) outperform the other algorithms on the RANDOM family in terms of asymptotic growth rates, but, with constant factors that are sometimes bigger by two orders of magnitude, it is hardly noticeable. The strong asymptotic performance is probably due to the low numbers of blocking flows needed.

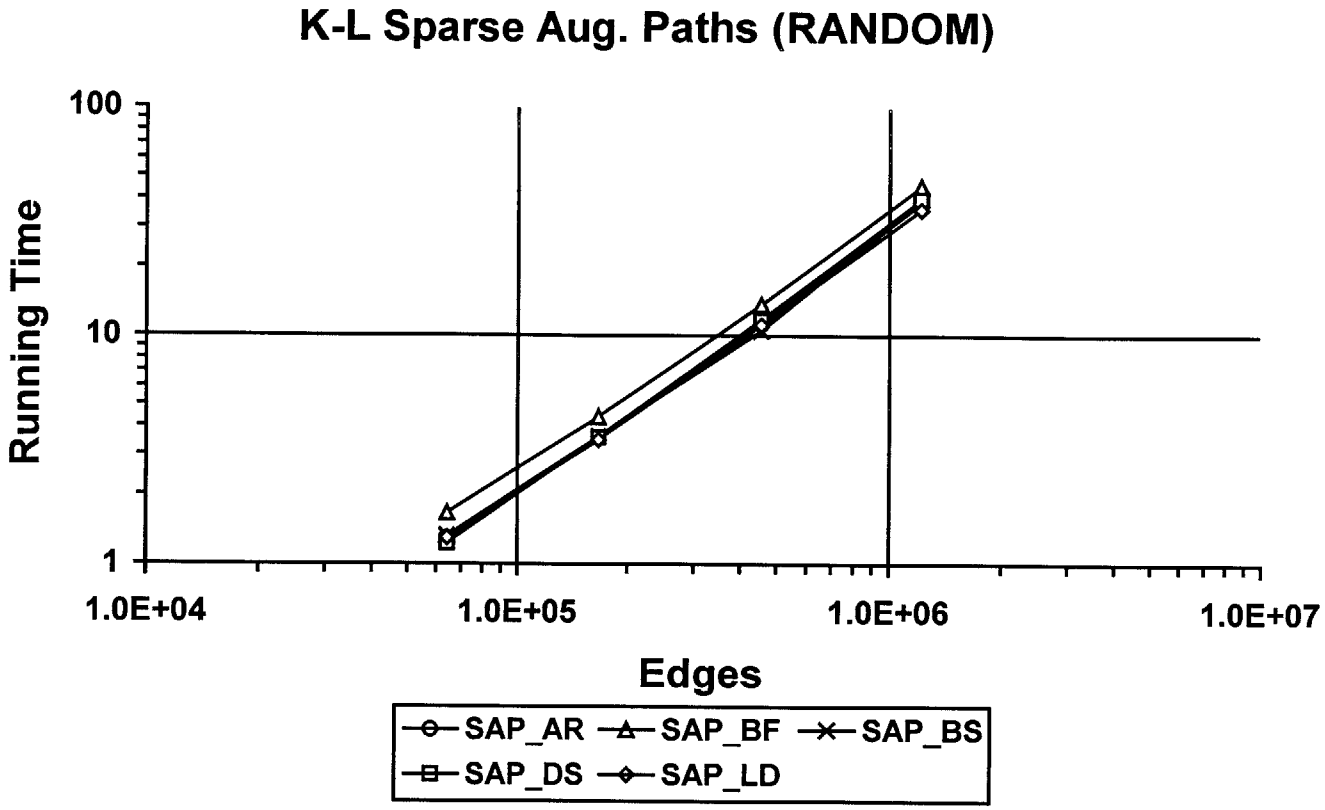
DISCLAIMER

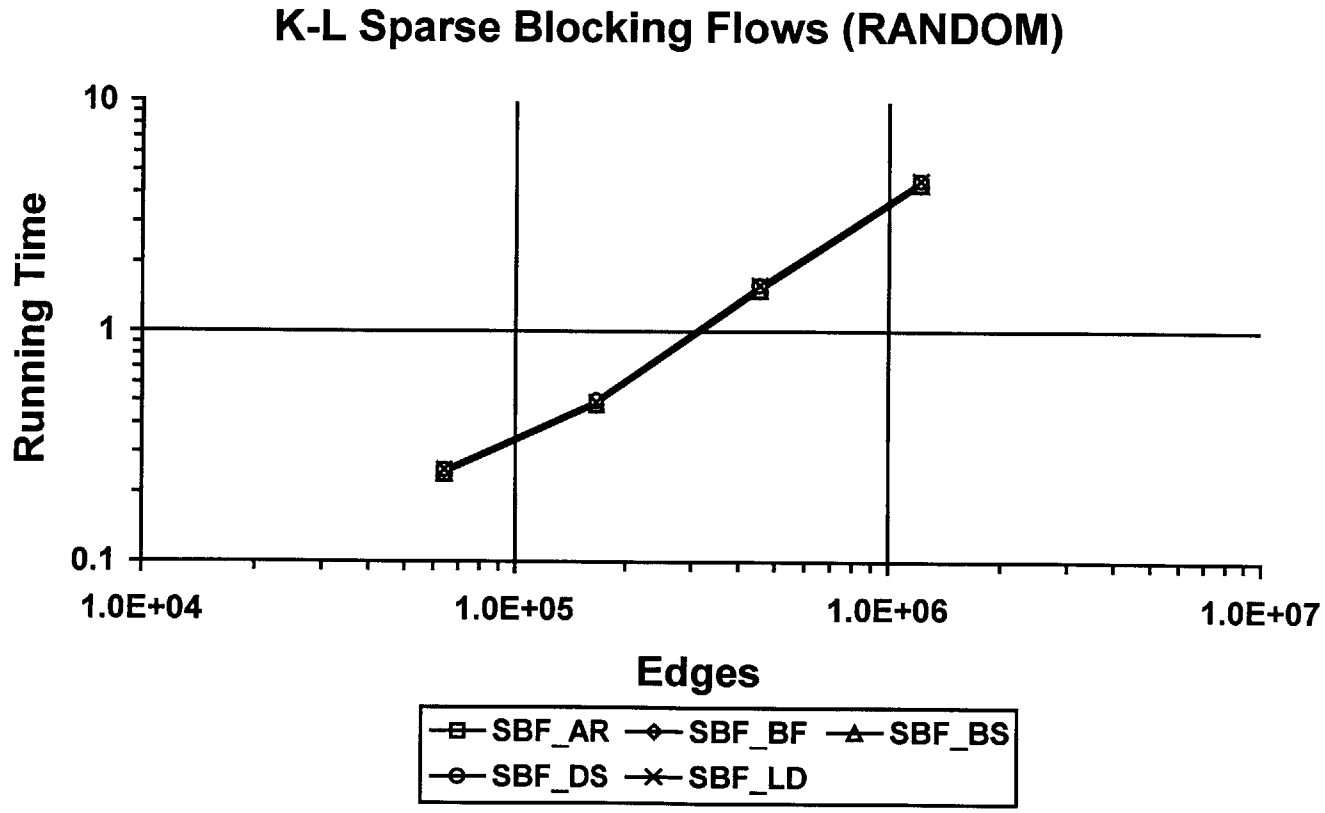
MISSING PAGE(S)

74

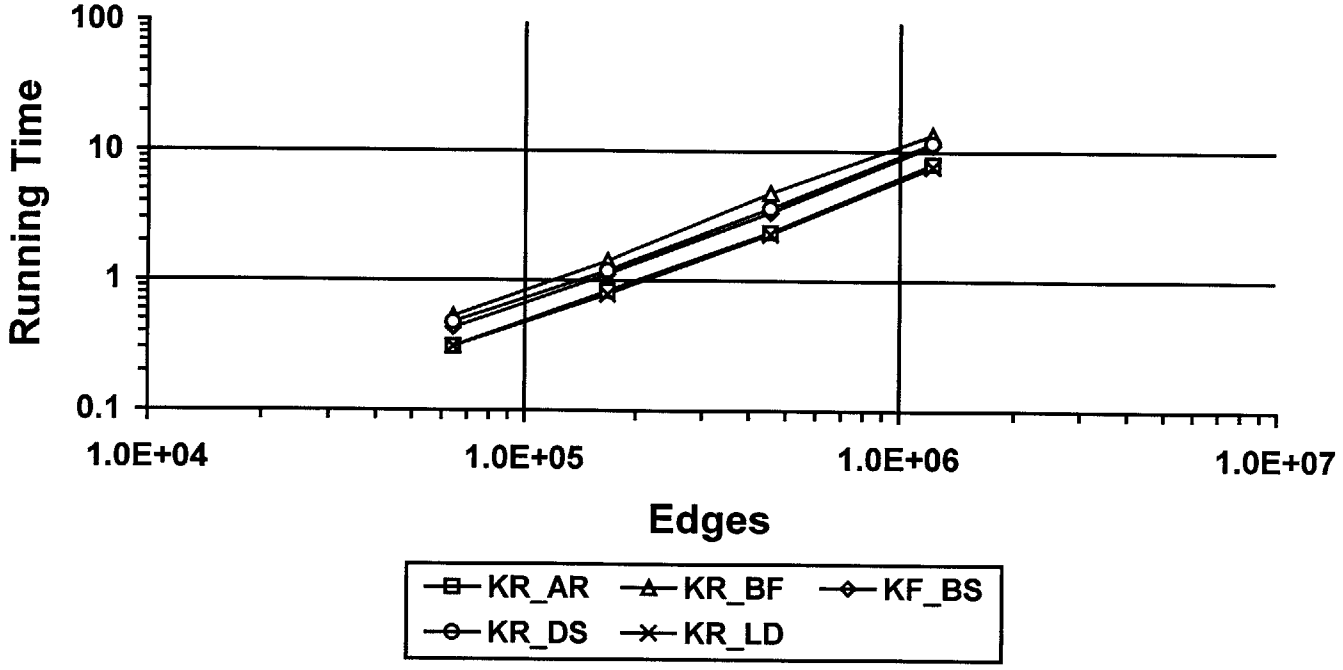

Push-Relabel (RANDOM)
Running Time
1
0.1
0.01
1.0E+04
1.0E+05
1.0E+06
1.0E+07
Edges
PR_FIFO
PR_HI
PR_LIFO
PR_LO


Aug. Paths (RANDOM)
Running Time
10
1
0.1
0.01
1.0E+04
1.0E+05
1.0E+06
1.0E+07
Edges
AP_AR
AP_BFS
AP_DFS
AP_LDS

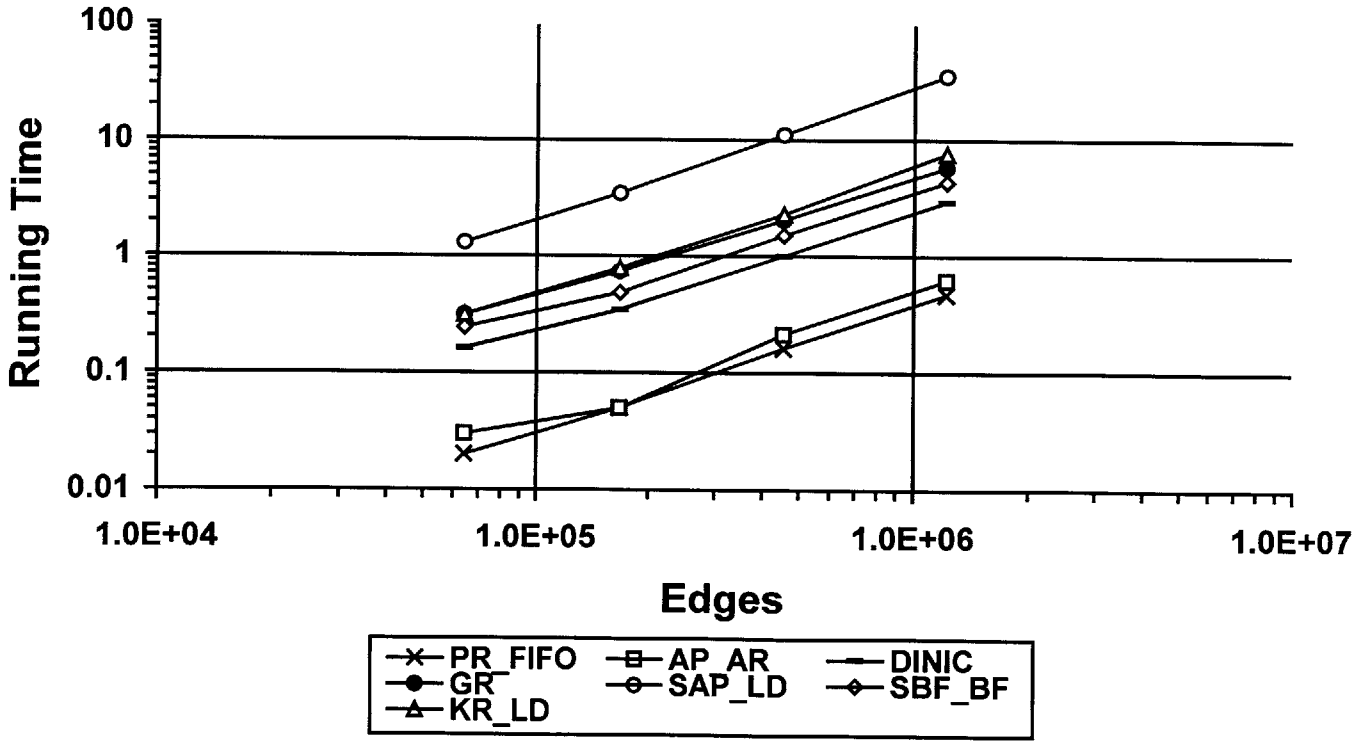

K-L Sparse Aug. Paths (RANDOM)
Running Time
100
10
1
1.0E+04
1.0E+05
1.0E+06
1.0E+07
Edges
SAP_AR
SAP_BF
SAP_BS
SAP_DS
SAP_LD


K-L Sparse Blocking Flows (RANDOM)
Running Time
10
1
0.1
1.0E+04
1.0E+05
1.0E+06
1.0E+07
Edges
SBF_AR
SBF_BF
SBF_BS
SBF_DS
SBF_LD

Randomized Aug. Paths (RANDOM)



Best Algorithms (RANDOM)

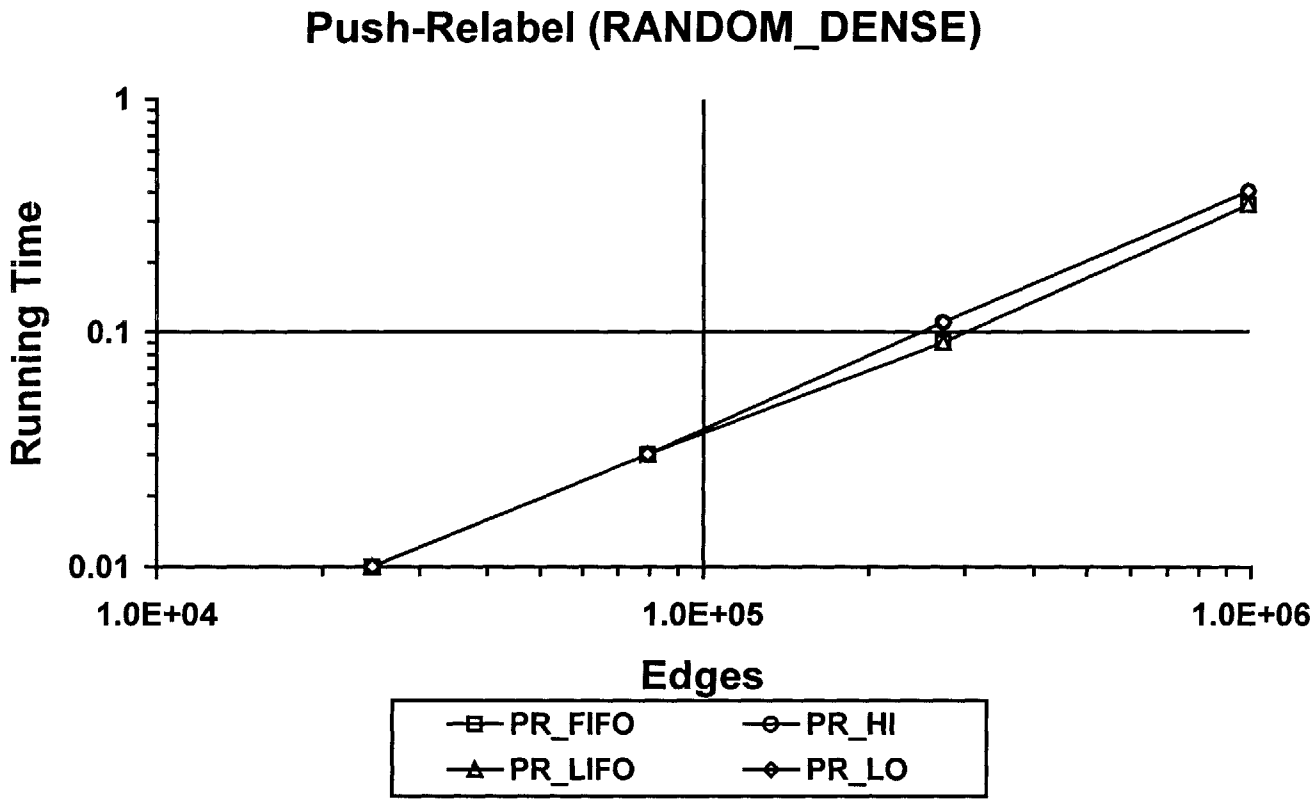


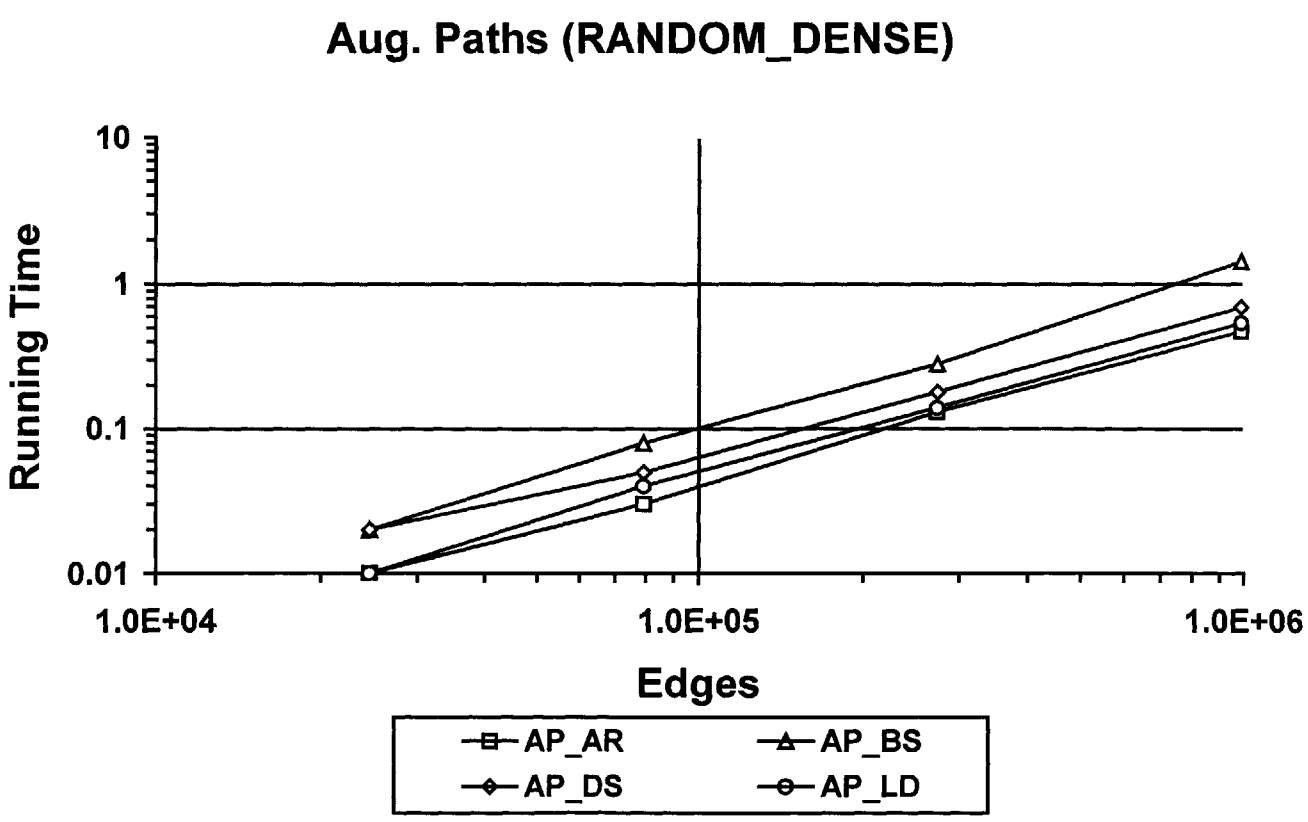
Code	Running Time on RANDOM
PR_FIFO	$1.28 \times 10^{-7} \times m^{1.08}$
PR_HI	$9.20 \times 10^{-8} \times m^{1.11}$
PR_LIFO	$1.28 \times 10^{-7} \times m^{1.08}$
PR_LO	$1.04 \times 10^{-7} \times m^{1.10}$
AP_AR	$1.81 \times 10^{-7} \times m^{1.07}$
AP_BS	$2.35 \times 10^{-7} \times m^{1.13}$
AP_DS	$5.17 \times 10^{-7} \times m^{1.02}$
AP_LD	$9.17 \times 10^{-8} \times m^{1.13}$
DINIC	$2.35 \times 10^{-6} \times m^{1.00}$
GR	$4.67 \times 10^{-6} \times m^{1.00}$
SAP_AR	$3.74 \times 10^{-6} \times m^{1.15}$
SAP_BF	$6.37 \times 10^{-6} \times m^{1.12}$
SAP_BS	$3.77 \times 10^{-6} \times m^{1.15}$
SAP_DS	$2.52 \times 10^{-6} \times m^{1.18}$
SAP_LD	$4.65 \times 10^{-6} \times m^{1.13}$
SBF_AR	$3.33 \times 10^{-6} \times m^{1.00}$
SBF_BF	$3.37 \times 10^{-6} \times m^{1.00}$
SBF_BS	$3.37 \times 10^{-6} \times m^{1.00}$
SBF_DS	$3.47 \times 10^{-6} \times m^{1.00}$
SBF_LD	$3.33 \times 10^{-6} \times m^{1.00}$
KR_AR	$1.52 \times 10^{-6} \times m^{1.10}$
KR_BF	$2.32 \times 10^{-6} \times m^{1.11}$
KR_BS	$1.91 \times 10^{-6} \times m^{1.11}$
KR_DS	$2.69 \times 10^{-6} \times m^{1.09}$
KR_LD	$1.72 \times 10^{-6} \times m^{1.09}$

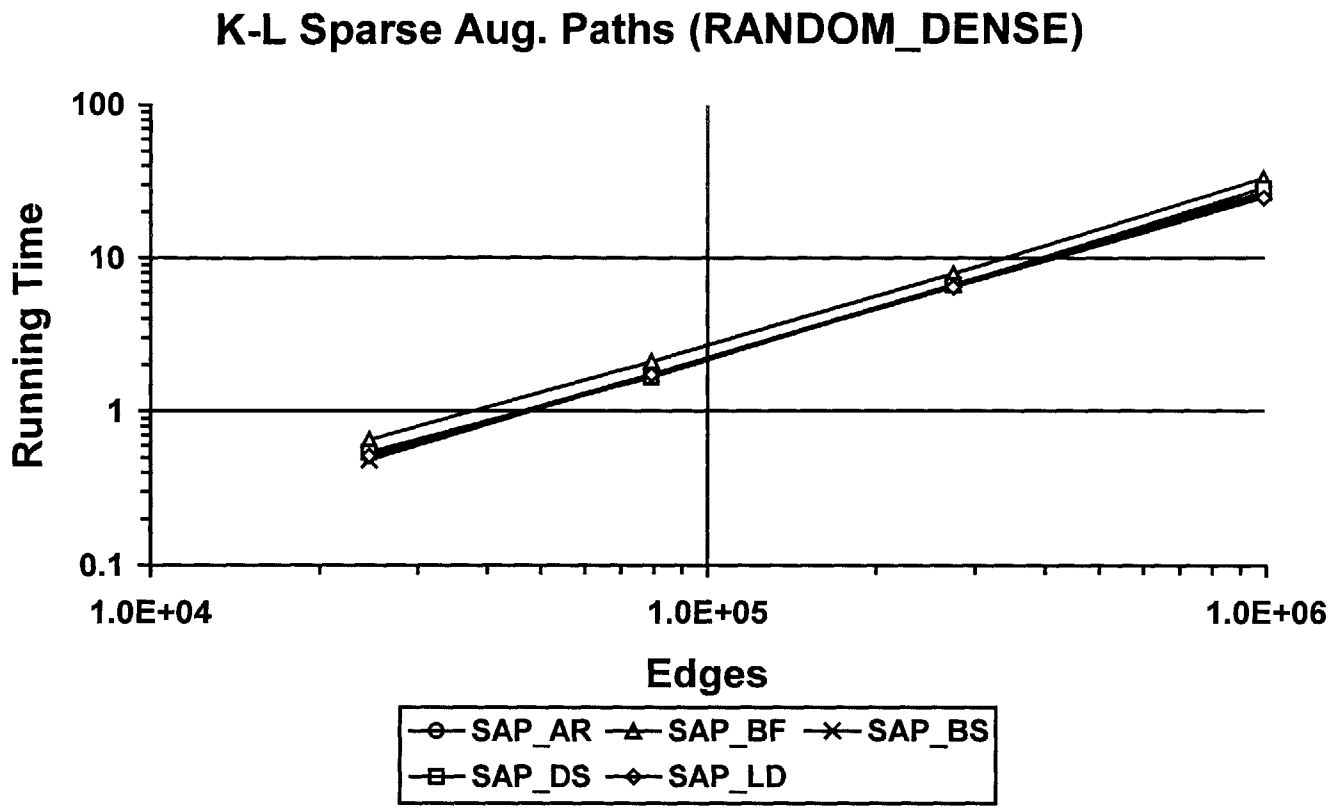
The RANDOM_DENSE graphs proved to be relatively easy for all of the algorithms, probably due to their relatively small flow values (our 988 thousand edge graphs had a meager average flow value of 42). All of the push-relabel algorithms ran in about the same time on this problem; PR_FIFO was the best of the bunch by a hair. Among the augmenting paths algorithms showed the approaches with global relabeling, AP_AR and AP_LD, performed somewhat better than the others. It does appear, however, that AP_DS would have eventually passed the other two algorithms; its rate of growth is somewhat better. The sparsification-based algorithms generally do not fare well; the sparsification seems to make the flow-carrying paths harder for the algorithms to find, thus making the running times of the actual flow-finding parts of the algorithms larger than they are without sparsification.

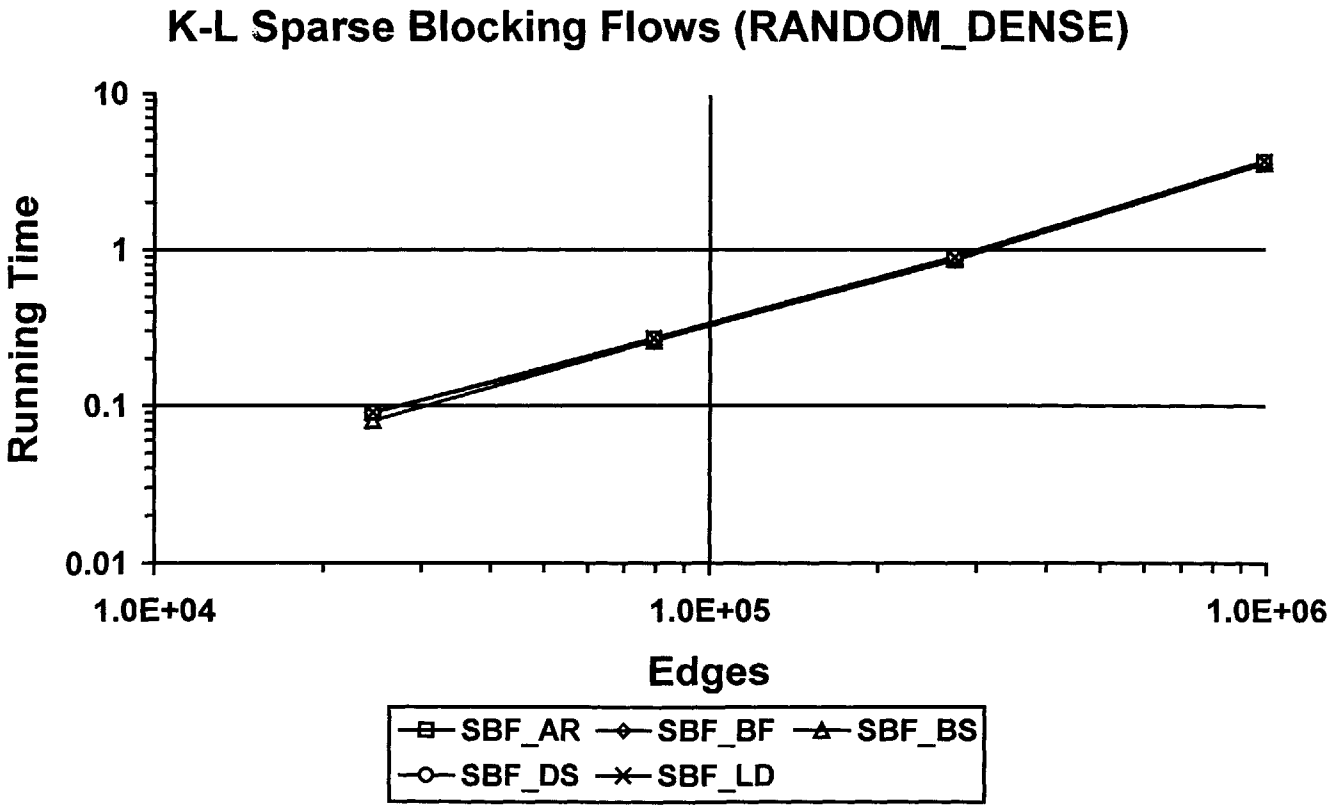
The Karger-Levine sparse blocking flows algorithms are uninteresting; once again the flow value is too low to cause the augmenting paths code to run so all the versions perform the same as Dinic's algorithm. All our blocking flow algorithms (Dinic's, Goldberg-Rao sparse blocking flows, and Karger-Levine sparse blocking flows) outperform most of the other algorithms on the RANDOM_DENSE family in terms of asymptotic growth rates, but, with constant factors that are sometimes bigger by two orders of magnitude, it is hardly noticeable. The strong asymptotic performance is probably due to the low numbers of blocking flows needed.

Overall, then, we found that the RANDOM and RANDOM_DENSE families caused strikingly similar performance in the algorithms, despite their large difference in edge density. We speculate that this is because the defining characteristic of both families is not their density but rather their small flow values.

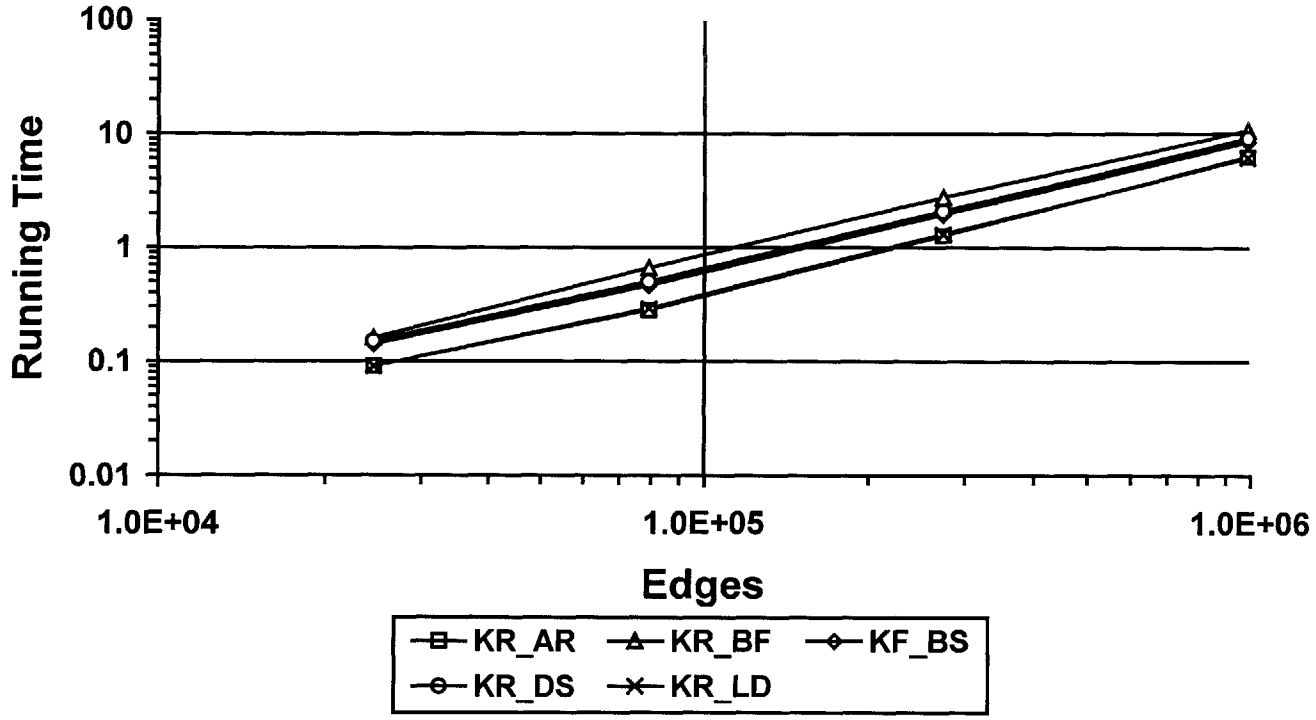

Push-Relabel (RANDOM_DENSE)
Running Time
1
0.1
0.01
1.0E+04
1.0E+05
1.0E+06
Edges
PR_FIFO
PR_HI
PR_LIFO
PR_LO


Aug. Paths (RANDOM_DENSE)
Running Time
10
1
0.1
0.01
1.0E+04
1.0E+05
1.0E+06
Edges
AP_AR
AP_BS
AP_DS
AP_LD

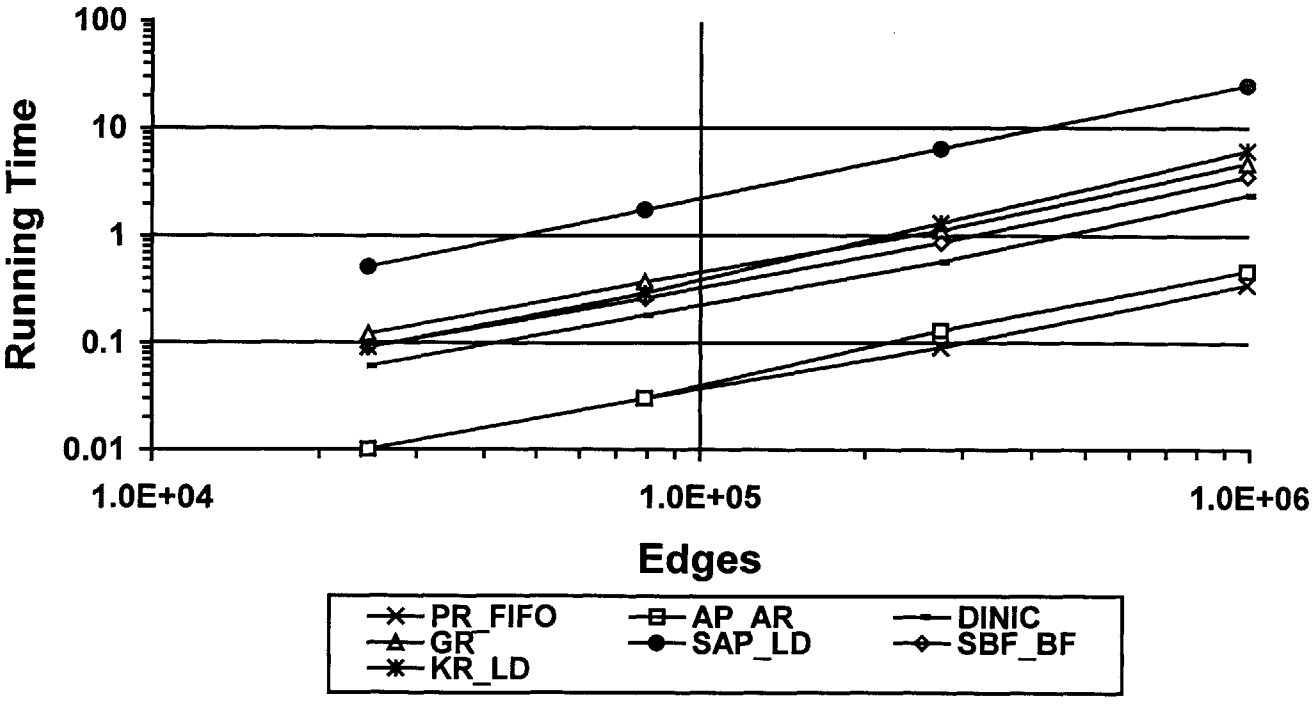

K-L Sparse Aug. Paths (RANDOM_DENSE)
Running Time
100
10
1
0.1
1.0E+04
1.0E+05
1.0E+06
Edges
SAP_AR
SAP_BF
SAP_BS
SAP_DS
SAP_LD


K-L Sparse Blocking Flows (RANDOM_DENSE)
Running Time
10
1
0.1
0.01
1.0E+04
1.0E+05
1.0E+06
Edges
SBF_AR
SBF_BF
SBF_BS
SBF_DS
SBF_LD

Randomized Aug. Paths (RANDOM_DENSE)



Best Algorithms (RANDOM_DENSE)



Code	Running Time on RANDOM_DENSE
PR_FIFO	$6.28 \times 10^{-7} \times m^{.96}$
PR_HI	$3.83 \times 10^{-7} \times m^{1.00}$
PR_LIFO	$6.28 \times 10^{-7} \times m^{.96}$
PR_LO	$3.83 \times 10^{-7} \times m^{1.00}$
AP_AR	$2.21 \times 10^{-7} \times m^{1.06}$
AP_BS	$1.99 \times 10^{-7} \times m^{1.14}$
AP_DS	$1.04 \times 10^{-6} \times m^{.97}$
AP_LD	$2.08 \times 10^{-7} \times m^{1.07}$
DINIC	$2.59 \times 10^{-6} \times m^{.99}$
GR	$5.59 \times 10^{-6} \times m^{.98}$
SAP_AR	$1.16 \times 10^{-5} \times m^{1.06}$
SAP_BF	$1.37 \times 10^{-5} \times m^{1.06}$
SAP_BS	$8.22 \times 10^{-6} \times m^{1.09}$
SAP_DS	$9.44 \times 10^{-6} \times m^{1.08}$
SAP_LD	$1.25 \times 10^{-5} \times m^{1.05}$
SBF_AR	$3.78 \times 10^{-6} \times m^{.99}$
SBF_BF	$3.81 \times 10^{-6} \times m^{.99}$
SBF_BS	$2.60 \times 10^{-6} \times m^{1.02}$
SBF_DS	$3.46 \times 10^{-6} \times m^{1.00}$
SBF_LD	$3.49 \times 10^{-6} \times m^{1.00}$
KR_AR	$6.75 \times 10^{-7} \times m^{1.16}$
KR_BF	$1.66 \times 10^{-6} \times m^{1.14}$
KR_BS	$1.67 \times 10^{-6} \times m^{1.12}$
KR_DS	$1.77 \times 10^{-6} \times m^{1.12}$
KR_LD	$7.37 \times 10^{-7} \times m^{1.15}$

4.2.1.2 Shaded Density

As discussed earlier, we used the Shaded Density generator to produce the SHADED_DENSITY family of graphs.

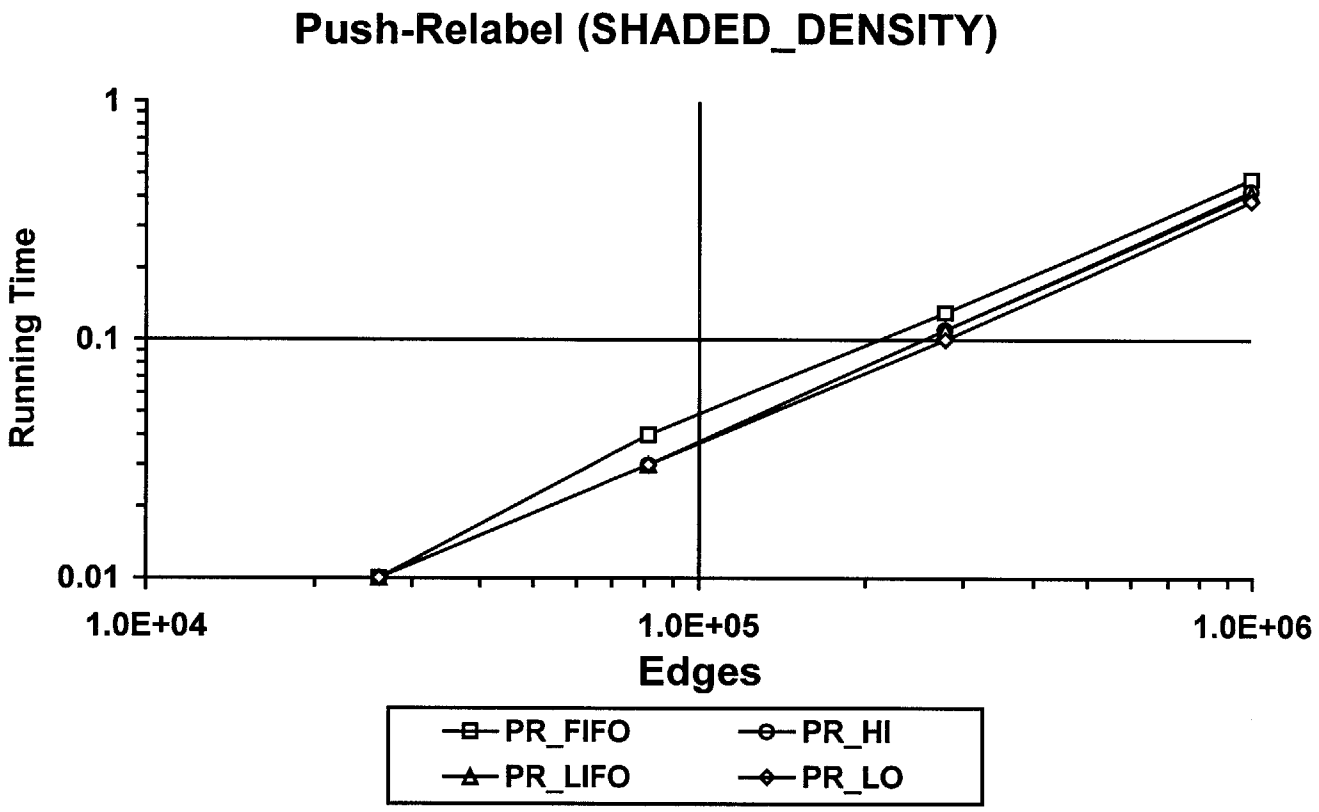
Similar to the graphs produced by the Random Generator, the SHADED_DENSITY graphs were easy for most of the algorithms. This is probably due to the fact that that flow value in these graphs is fairly low. The performance of the push-relabel algorithms is fairly dull; all of them execute in less than a second, with the lowest-level selection heuristic being a tad faster than the others. The performance of the basic augmenting paths algorithms is also surprisingly varied. In particular, breadth-first and depth-first search perform quite a bit worse than usual, probably because the graph is dense enough that it is easy to wander around in it, and stand little chance of progressing forward (there are usually more edges that lead backwards than forwards). Augment-relabel and label-directed search take good advantage of their distance labels and perform fairly nicely.

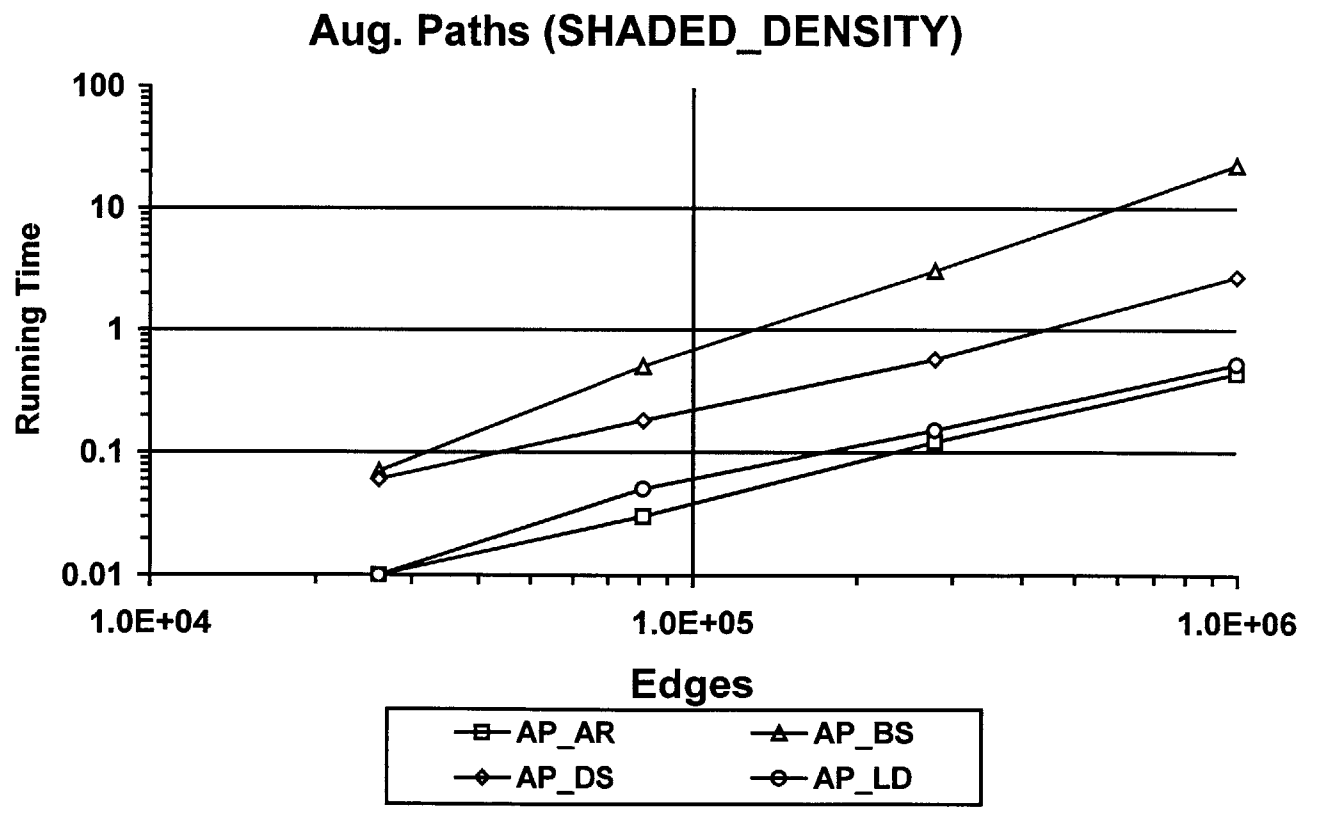
Turning our attention to the sparsifying algorithms, we see that the Karger-Levine sparse augmenting paths algorithms do not perform well on this family, achieving running time growth rates that are generally about the same as the plain augmenting paths algorithms (with the exception of breadth-first search, where the sparse version is considerably better) but with much worse constant factors. This seems to be not only because the sparsification takes a fair amount of time and does not provide much benefit to an algorithm that uses distance labels, but also because the sparsification makes the flow-carrying paths harder to find and thus the flow-finding subroutines actually take more time in the sparsifying algorithm than in the normal algorithm. The Karger-Levine sparse blocking flows implementations perform the same; again, the flow value is too small to cause the augmenting paths code to run.

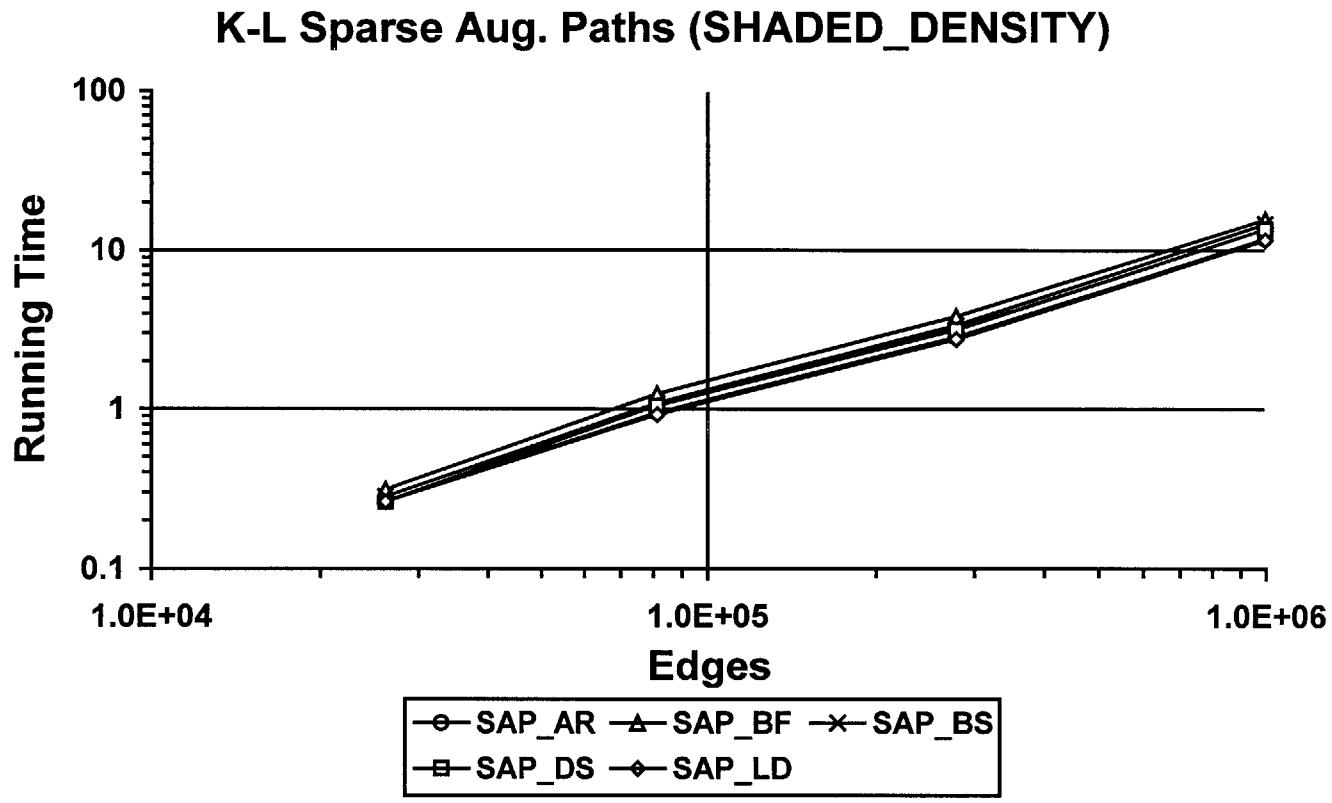
We also observe that the Karger randomized augmenting paths performs worst as compared to the other algorithms on this graph family than on RANDOM and RANDOM_DENSE. This is as we expected; the sparseness of the graph in certain regions means that there are often a few critical edges that all need to be present for an augmenting path to be

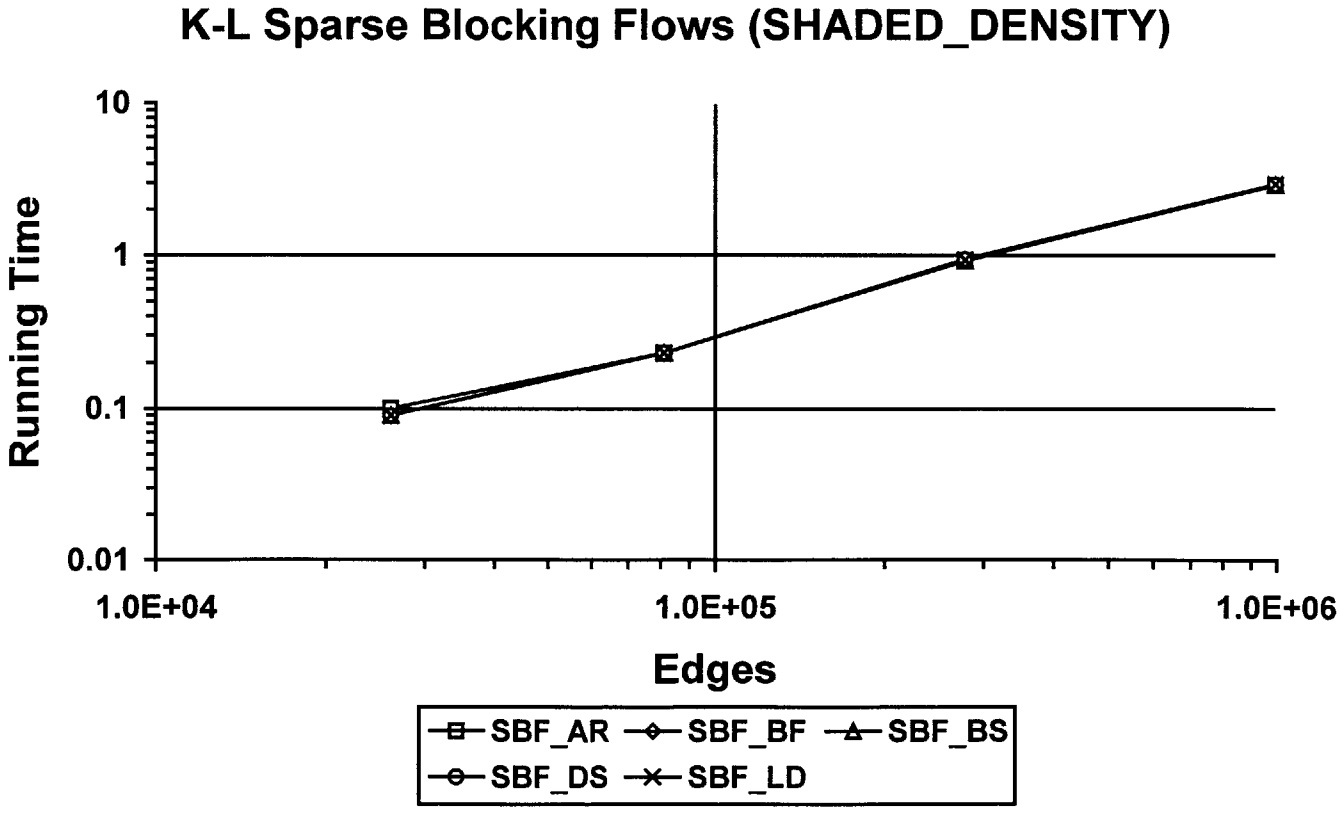
found. This effectively defeats the ideas behind the randomized recursion (although it seems that it is hard to fully take advantage of this without creating a very unnatural graph).

Overall then, the push-relabel and normal augment-relabel implementations perform the best on this family, followed by the blocking flow algorithms, as the graphs seem to require very few blocking flows. The sparsifying algorithms perform poorly. Other than the performance of the Karger randomized augmenting paths algorithm being worst than usual, the performance in this family is essentially the same as in the RANDOM and RANDOM_DENSE families.

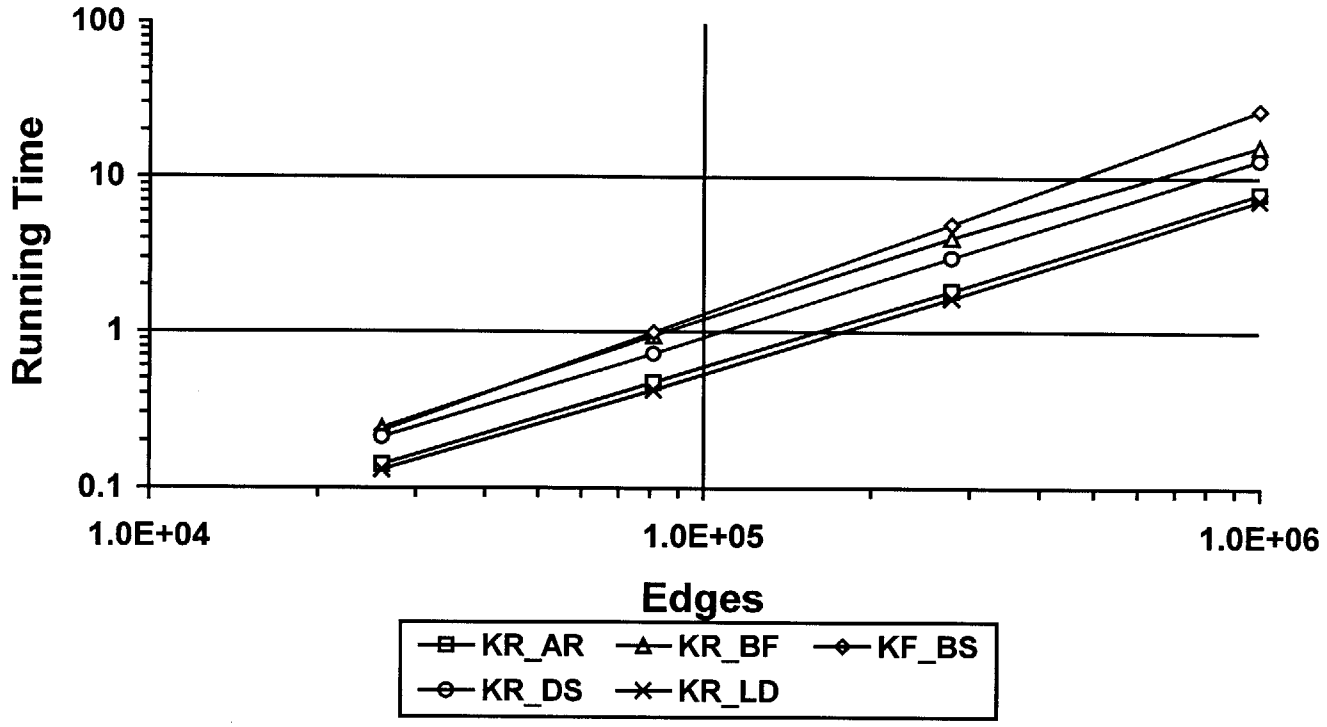

Push-Relabel (SHADED_DENSITY)
Running Time
1
0.1
0.01
1.0E+04
1.0E+05
1.0E+06
Edges
PR_FIFO
PR_HI
PR_LIFO
PR_LO


Aug. Paths (SHADED_DENSITY)
Running Time
100
10
1
0.1
0.01
1.0E+04
1.0E+05
1.0E+06
Edges
AP_AR
AP_BS
AP_DS
AP_LD

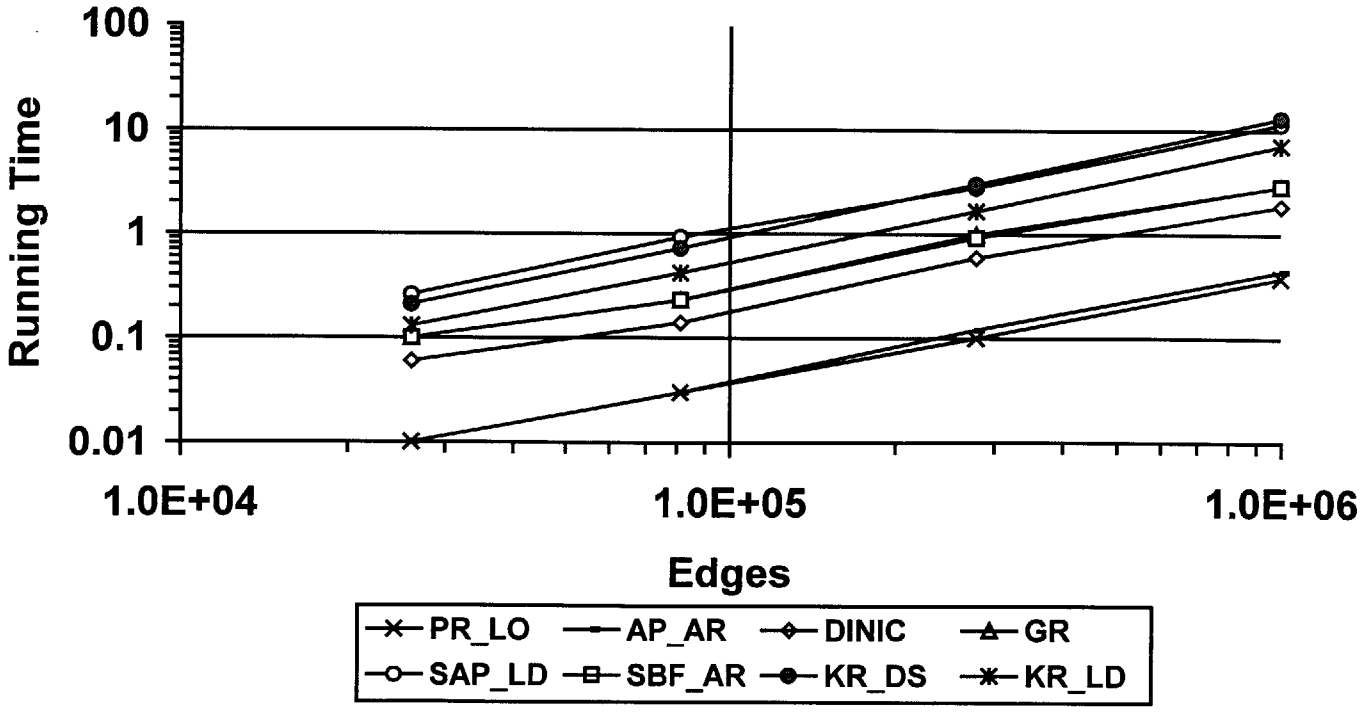

K-L Sparse Aug. Paths (SHADED_DENSITY)
Running Time
100
10
1
0.1
1.0E+04
1.0E+05
1.0E+06
Edges
SAP_AR
SAP_BF
SAP_BS
SAP_DS
SAP_LD


K-L Sparse Blocking Flows (SHADED_DENSITY)
Running Time
10
1
0.1
0.01
1.0E+04
1.0E+05
1.0E+06
Edges
SBF_AR
SBF_BF
SBF_BS
SBF_DS
SBF_LD

Randomized Aug. Paths (SHADED_DENSITY)



Best Algorithms (SHADED_DENSITY)



Code	Running Time on SHADED_DENSITY
PR_FIFO	$2.62 \times 10^{-7} \times m^{1.05}$
PR_HI	$2.75 \times 10^{-7} \times m^{1.03}$
PR_LIFO	$2.94 \times 10^{-7} \times m^{1.02}$
PR_LO	$3.87 \times 10^{-7} \times m^{1.00}$
AP_AR	$2.28 \times 10^{-7} \times m^{1.05}$
AP_BS	$8.51 \times 10^{-9} \times m^{1.57}$
AP_DS	$1.48 \times 10^{-6} \times m^{1.04}$
AP_LD	$2.40 \times 10^{-7} \times m^{1.06}$
DINIC	$3.01 \times 10^{-6} \times m^{.97}$
GR	$6.02 \times 10^{-6} \times m^{.95}$
SAP_AR	$7.47 \times 10^{-6} \times m^{1.03}$
SAP_BF	$7.05 \times 10^{-6} \times m^{1.06}$
SAP_BS	$5.64 \times 10^{-6} \times m^{1.07}$
SAP_DS	$5.44 \times 10^{-6} \times m^{1.07}$
SAP_LD	$7.84 \times 10^{-6} \times m^{1.03}$
SBF_AR	$6.16 \times 10^{-6} \times m^{.95}$
SBF_BF	$4.23 \times 10^{-6} \times m^{.98}$
SBF_BS	$4.38 \times 10^{-6} \times m^{.97}$
SBF_DS	$4.20 \times 10^{-6} \times m^{.98}$
SBF_LD	$4.19 \times 10^{-6} \times m^{.98}$
KR_AR	$1.79 \times 10^{-6} \times m^{1.11}$
KR_BF	$2.00 \times 10^{-6} \times m^{1.15}$
KR_BS	$3.66 \times 10^{-7} \times m^{1.31}$
KR_DS	$1.95 \times 10^{-6} \times m^{1.14}$
KR_LD	$1.72 \times 10^{-6} \times m^{1.10}$

4.2.1.3 Grid

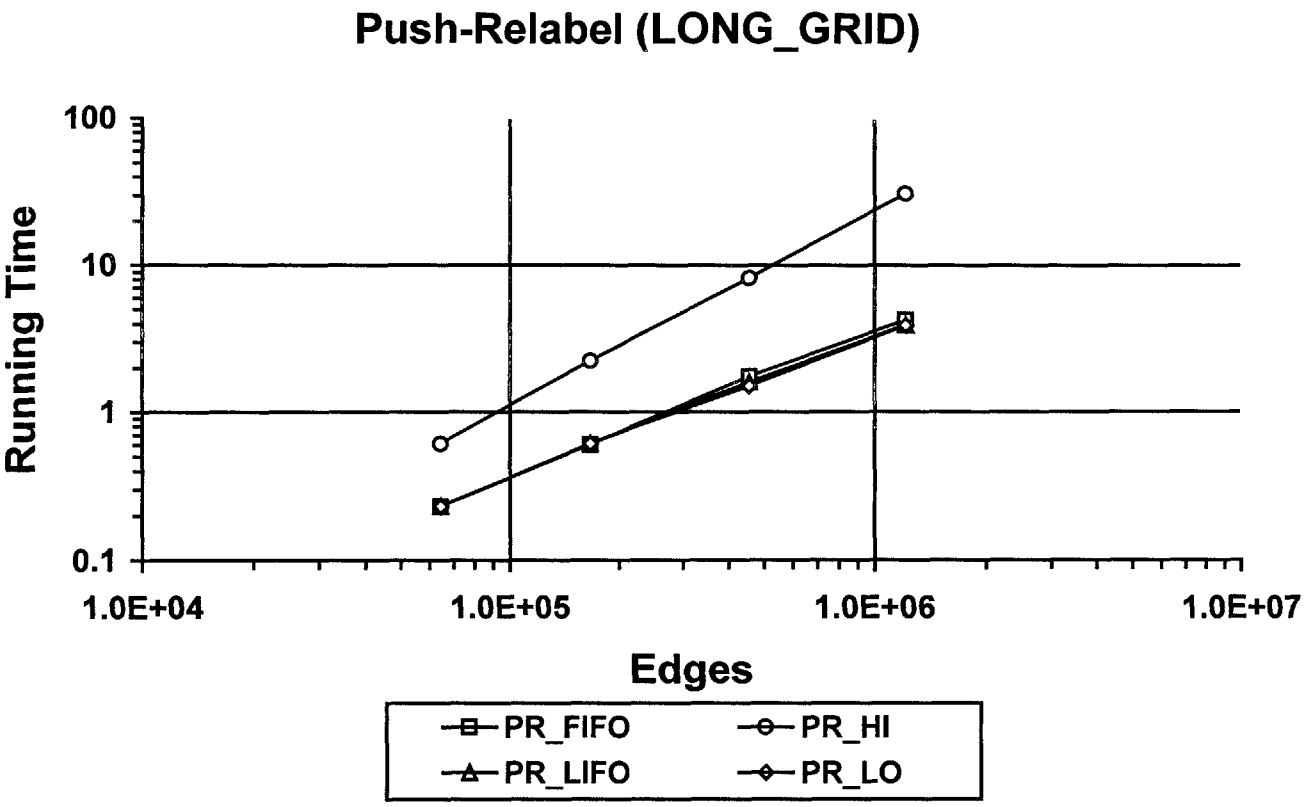
We used the Grid family of generators to produce three different families of graphs, LONG_GRID, CUBE_GRID, and WIDE_GRID. We present discussion, graphs, and tables for these three families, starting with LONG_GRID and ending with WIDE_GRID. These families give a huge advantage to algorithms that use some form of distance labelling, as the distance labellings quickly identify where the flow needs to go, and rarely become inaccurate. The algorithms without distance labels have a tendency to wander around the graphs and, as a result, take significantly longer. Also, as all of these graphs are fairly sparse, it would stand to reason that the sparsification algorithms would not perform particularly well.

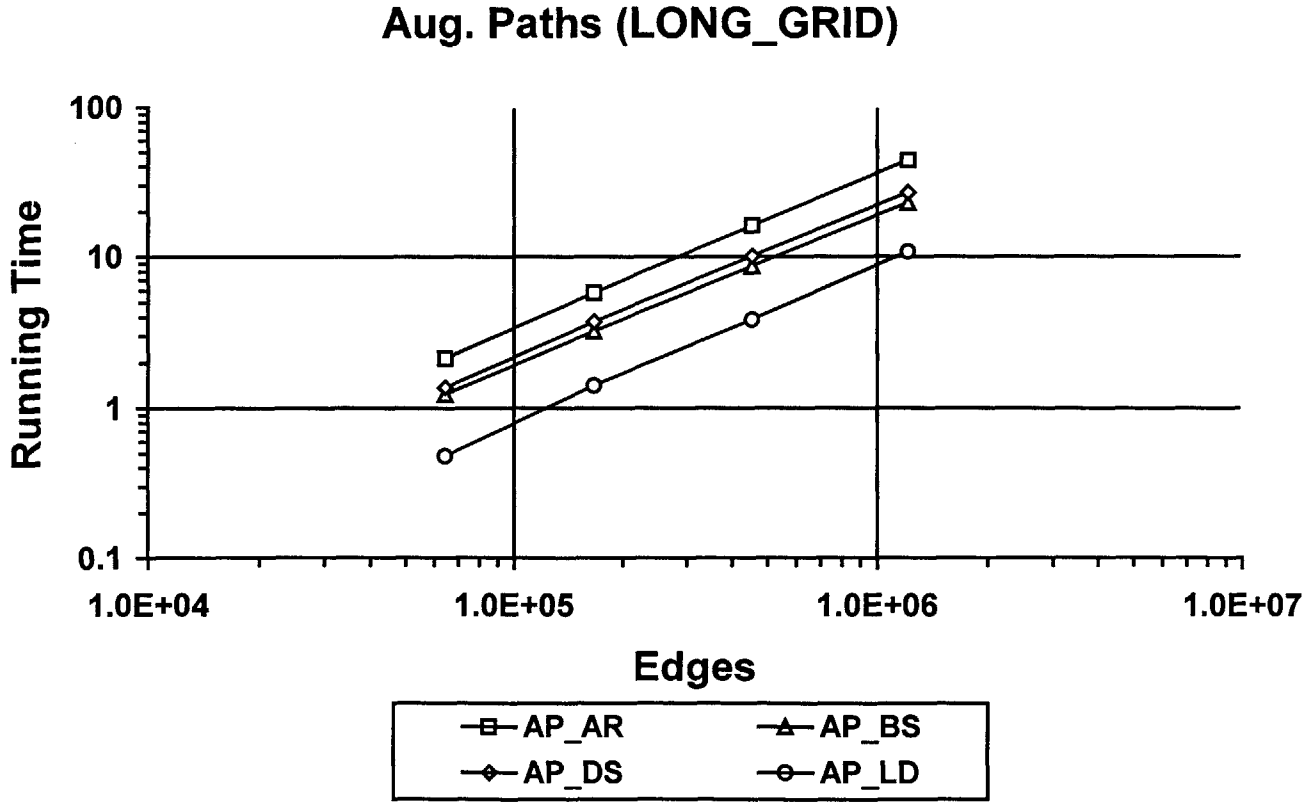
In the LONG_GRID family, we see that three of the push-relabel heuristics perform essentially the same, with lowest-level selection leading the way, and highest level selection performing much worse than the other three. Within the augmenting paths approaches, we see that label-directed search performs the best, with the breadth-first and depth-first searches lagging behind a bit, and finally the augment-relabel approach trailing the rest. As with a few other graph families, it looks like breadth-first and depth-search are slightly asymptotically better than the label-directed search; it is tempting to speculate that this might be a result of the decision to run global relabelling every $O(n)$ times, which may yield much better constants but lead to asymptotically worst performance in the limit. It is unclear why the augment-relabel approach performs so poorly.

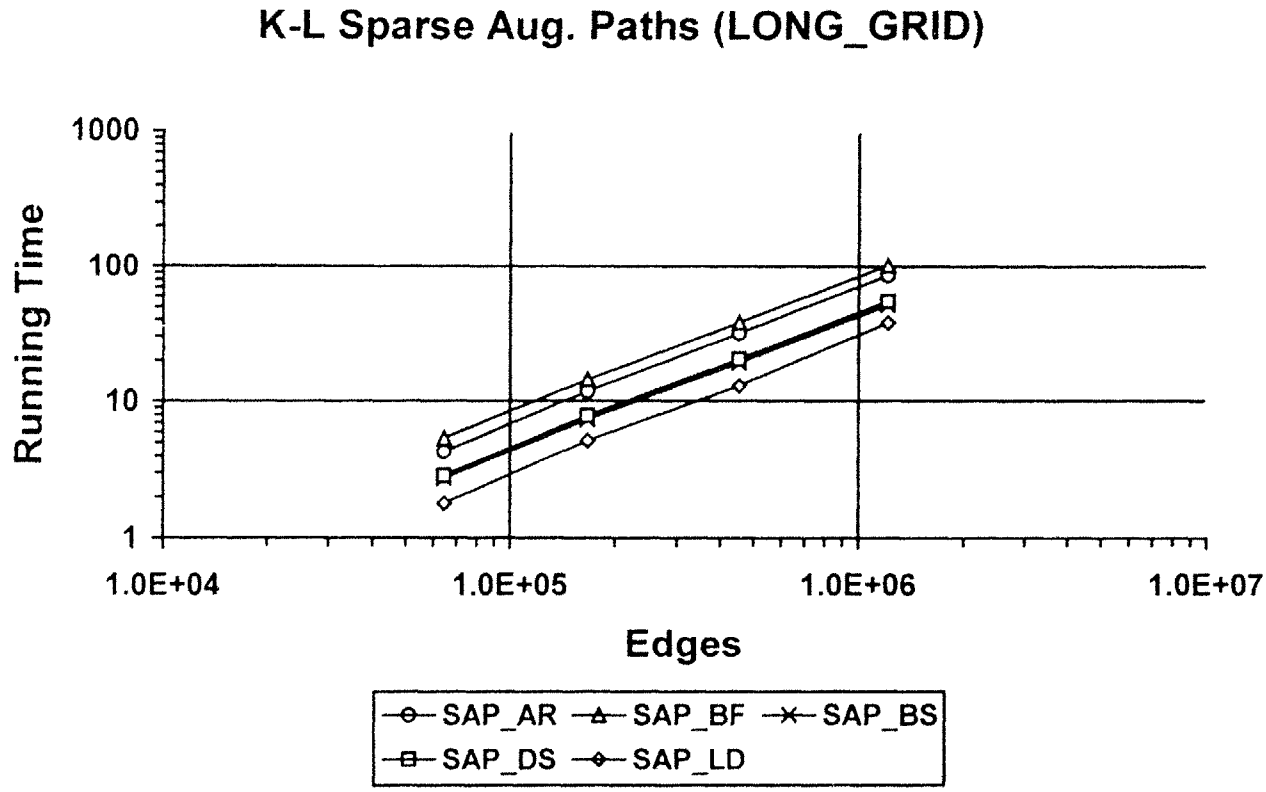
The Karger-Levine sparse augmenting paths and Karger randomized augmenting paths algorithms are unsurprisingly uninteresting; the graph is already sparse enough that there is not much to be gained by sparsification, and the few edges that are removed seem to only make it harder to find paths along which to send flow.

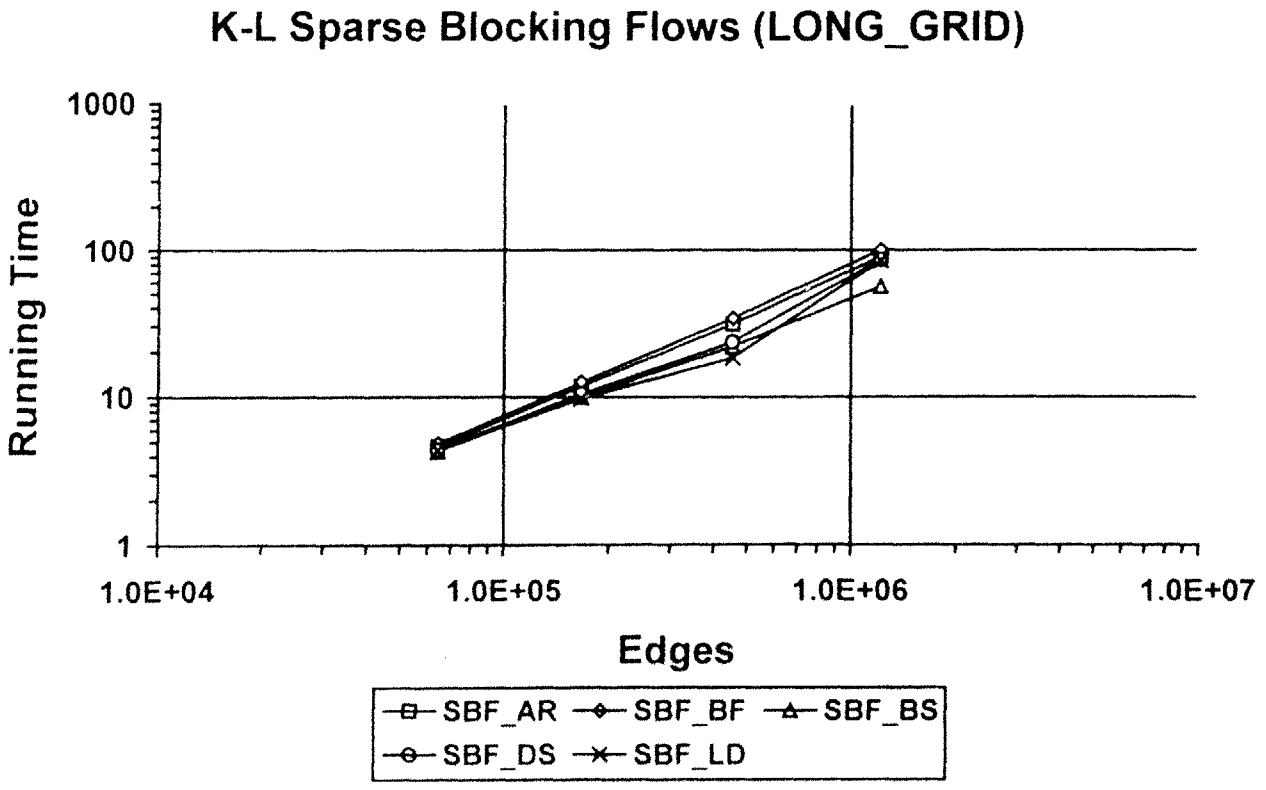
The Karger-Levine sparse blocking flows algorithm performs fairly uninterestingly, with the exception of the breadth-first search version, where it displays a run time growth rate significantly quicker than any of the others. Exactly why this works so well is unclear

Finally, the Goldberg-Rao and Dinic algorithms continue to perform weakly, probably due to the lack of effect of sparsification and the fact that the algorithms seem to only add about one unit of flow per blocking flow computation.

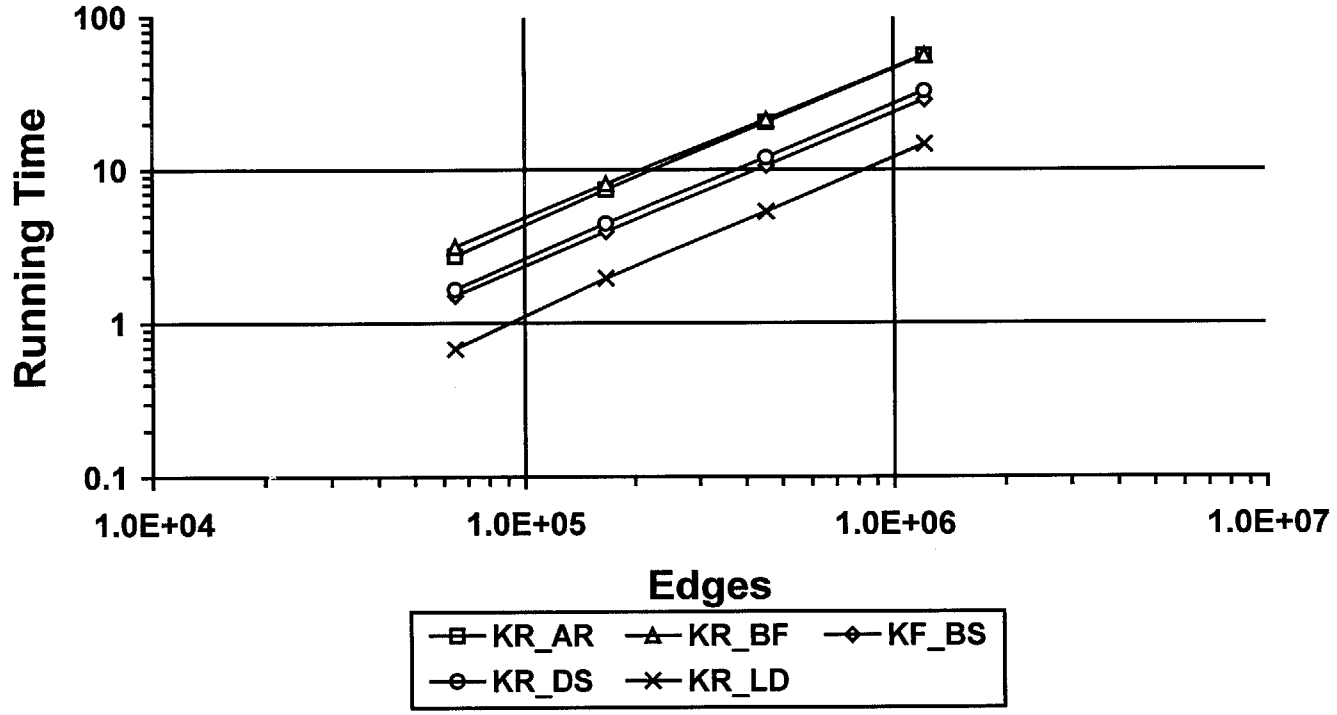

Push-Relabel (LONG_GRID)
Running Time
100
10
1
0.1
1.0E+04
1.0E+05
1.0E+06
1.0E+07
Edges
PR_FIFO
PR_HI
PR_LIFO
PR_LO


Aug. Paths (LONG_GRID)
Running Time
100
10
1
0.1
1.0E+04
1.0E+05
1.0E+06
1.0E+07
Edges
AP_AR
AP_BS
AP_DS
AP_LD

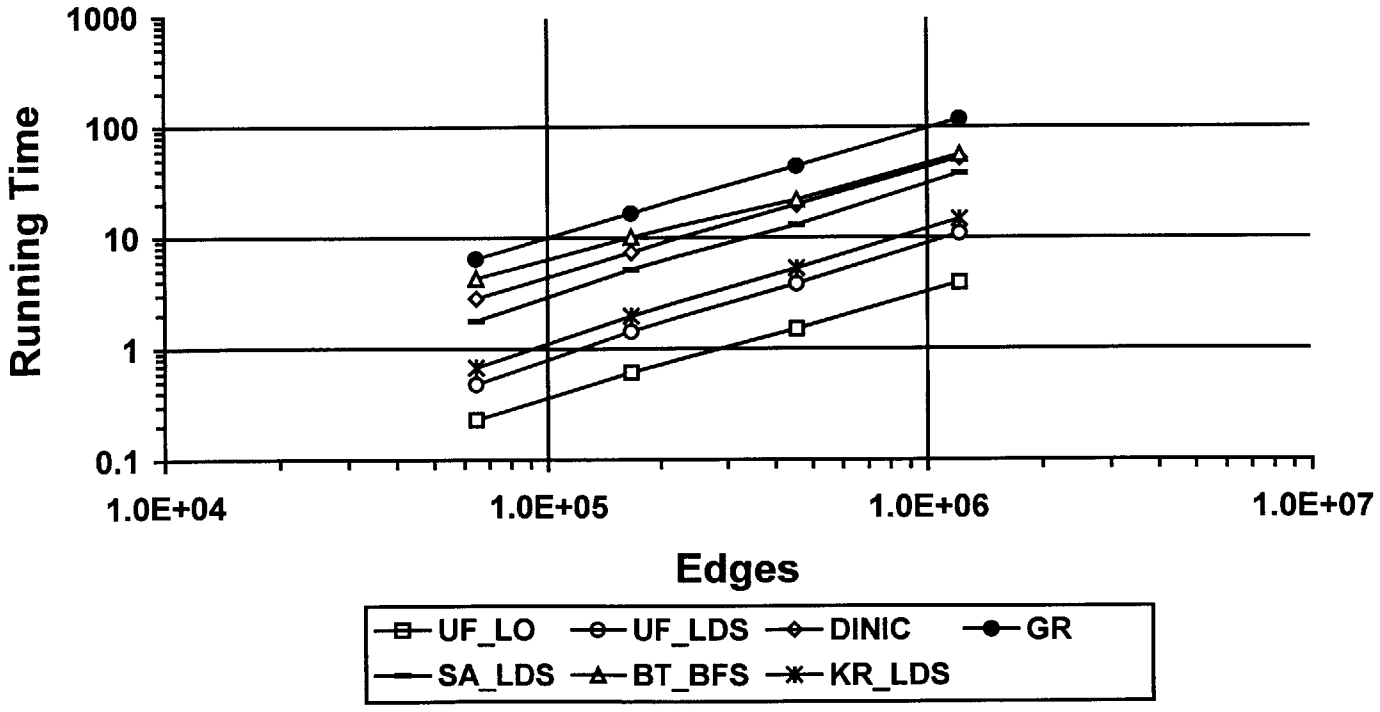

K-L Sparse Aug. Paths (LONG_GRID)
Running Time
1000
100
10
1
1.0E+04
1.0E+05
1.0E+06
1.0E+07
Edges
SAP_AR
SAP_BF
SAP_BS
SAP_DS
SAP_LD


K-L Sparse Blocking Flows (LONG_GRID)
Running Time
1000
100
10
1
1.0E+04
1.0E+05
1.0E+06
1.0E+07
Edges
SBF_AR
SBF_BF
SBF_BS
SBF_DS
SBF_LD

Randomized Aug. Paths (LONG_GRID)



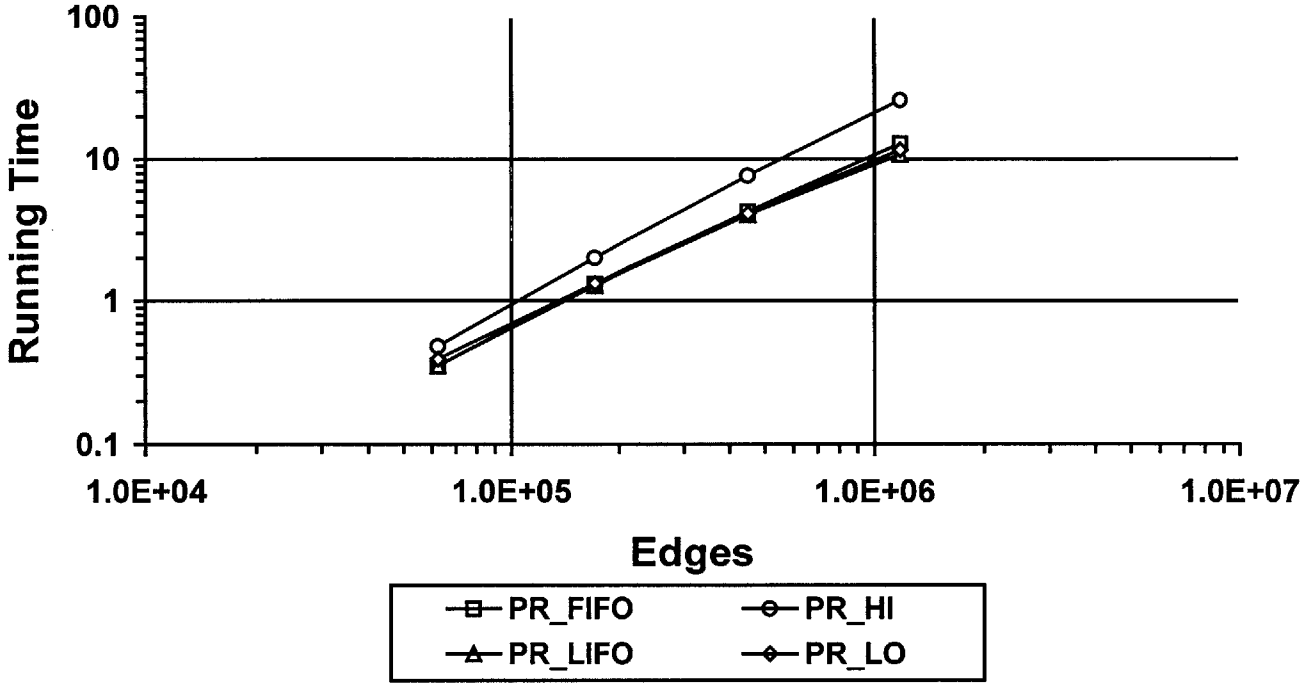
Best Algorithms (LONG_GRID)



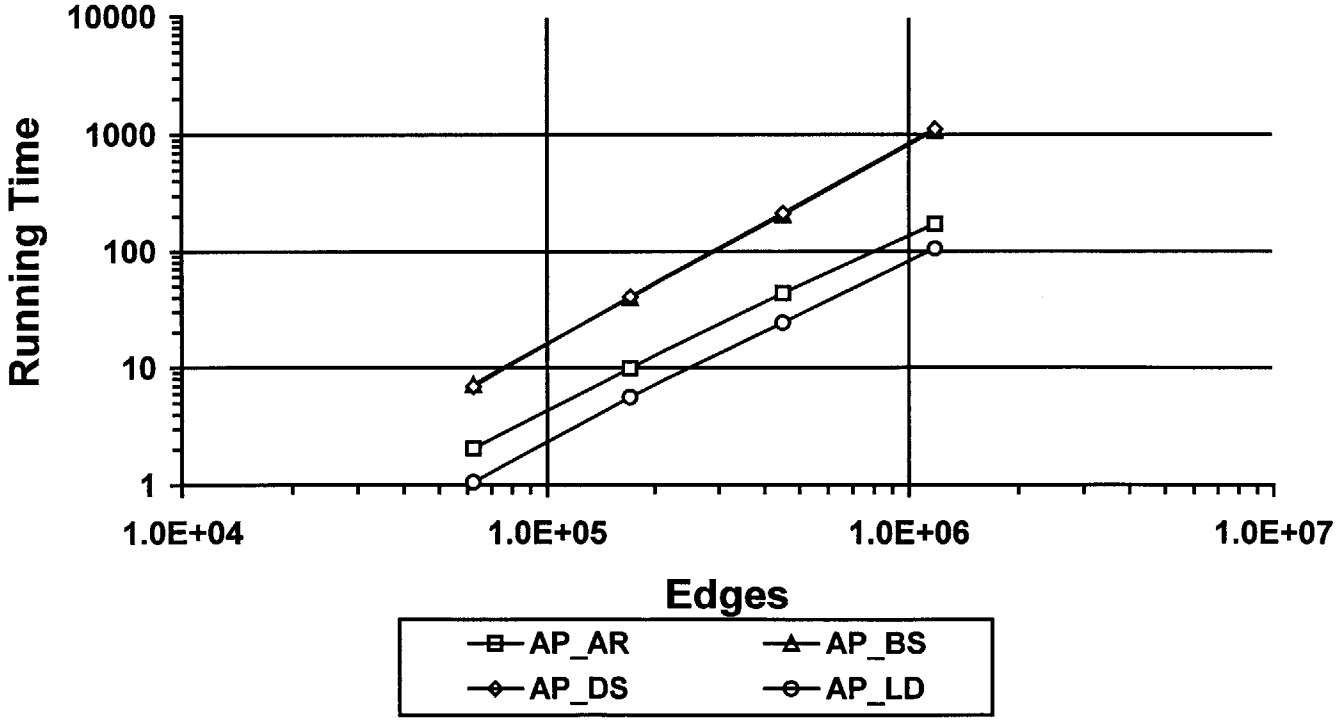
Code	Running Time on LONG_GRID
PR_FIFO	$3.62 \times 10^{-6} \times m^{1.00}$
PR_HI	$2.35 \times 10^{-7} \times m^{1.33}$
PR_LIFO	$5.45 \times 10^{-6} \times m^{.96}$
PR_LO	$5.94 \times 10^{-6} \times m^{.96}$
AP_AR	$2.41 \times 10^{-5} \times m^{1.03}$
AP_BS	$2.04 \times 10^{-5} \times m^{.99}$
AP_DS	$1.92 \times 10^{-5} \times m^{1.01}$
AP_LD	$4.33 \times 10^{-6} \times m^{1.05}$
DINIC	$4.91 \times 10^{-5} \times m^{.99}$
GR	$1.13 \times 10^{-4} \times m^{.99}$
SAP_AR	$5.29 \times 10^{-5} \times m^{1.02}$
SAP_BF	$8.56 \times 10^{-5} \times m^{1.00}$
SAP_BS	$4.33 \times 10^{-5} \times m^{1.00}$
SAP_DS	$4.34 \times 10^{-5} \times m^{1.00}$
SAP_LD	$2.07 \times 10^{-5} \times m^{1.03}$
SBF_AR	$6.84 \times 10^{-5} \times m^{1.00}$
SBF_BF	$5.58 \times 10^{-5} \times m^{1.03}$
SBF_BS	$3.03 \times 10^{-4} \times m^{.86}$
SBF_DS	$7.26 \times 10^{-5} \times m^{.99}$
SBF_LD	$8.60 \times 10^{-5} \times m^{.97}$
KR_AR	$3.31 \times 10^{-5} \times m^{1.02}$
KR_BF	$6.24 \times 10^{-5} \times m^{.98}$
KR_BS	$2.33 \times 10^{-5} \times m^{1.00}$
KR_DS	$2.26 \times 10^{-5} \times m^{1.01}$
KR_LD	$7.08 \times 10^{-6} \times m^{1.04}$

For the CUBE_GRID family, we see that once again the push-relabel algorithms run fairly similarly, with the lowest-level selection heuristic working the best and the highest level selection heuristic working notably worst than the rest. In the augmenting paths algorithms, the two algorithms that utilize distance labels perform significantly better than the two that do not, with label-directed search being the best of the former pair. The Karger-Levine sparse augmenting paths algorithms all have significantly worst constant factors than their plain-vanilla augmenting path counterparts, but also all have slightly better run time growth. The Karger-Levine sparse blocking flow algorithms do not actually run the augmenting paths code, and thus are uninteresting. The same situation is present with the Karger randomized sparse augmenting path algorithms, except the constants are not quite as bad and the run time growth is not quite as good. Taken all together, this makes for a fairly uninteresting picture for the performance of the best algorithms for this instance. The lowest-level push-relabel algorithm dominates all the other algorithms in the picture, both in terms of constant factors and growth rate. Both sparse augmenting paths algorithms have slightly better run time growth than the augmenting paths algorithm, but enough worse constants that the normal augmenting paths algorithm is better for our data ranges. Finally, once again Goldberg-Rao and Dinic's algorithm prove unviable, primarily due to the terrible performance of blocking flows.

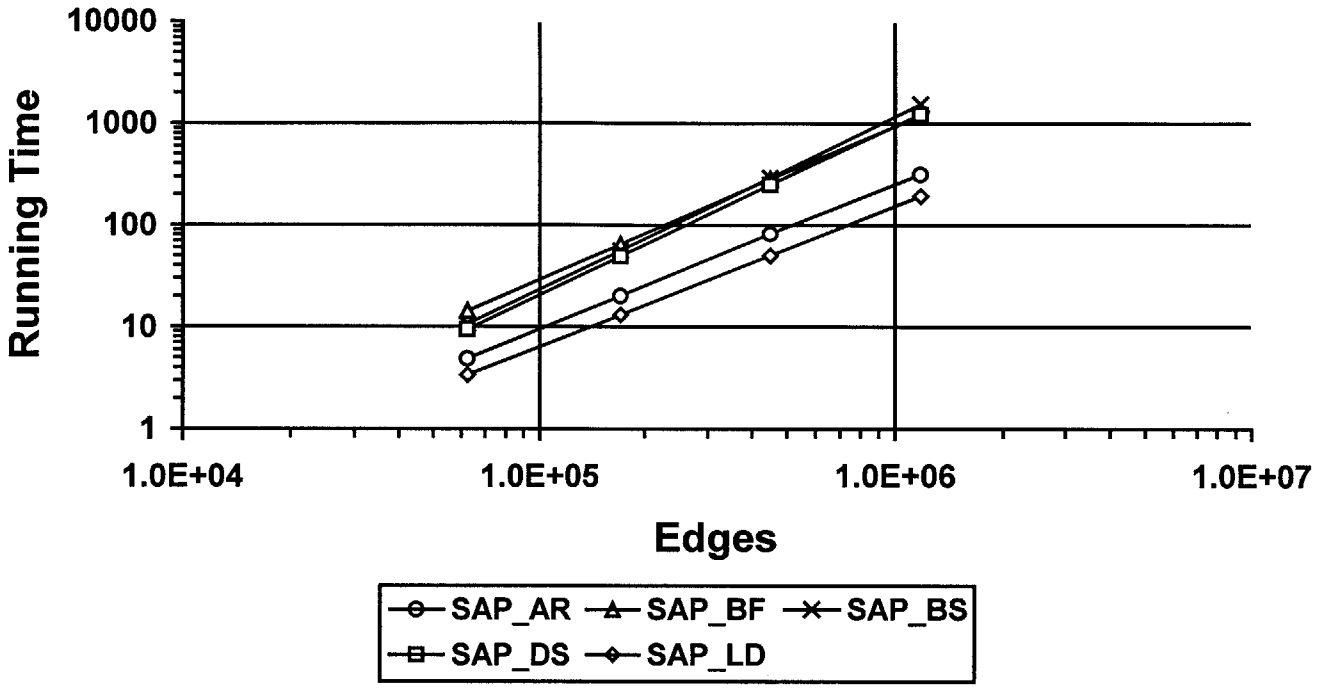
Push-Relabel (CUBE_GRID)



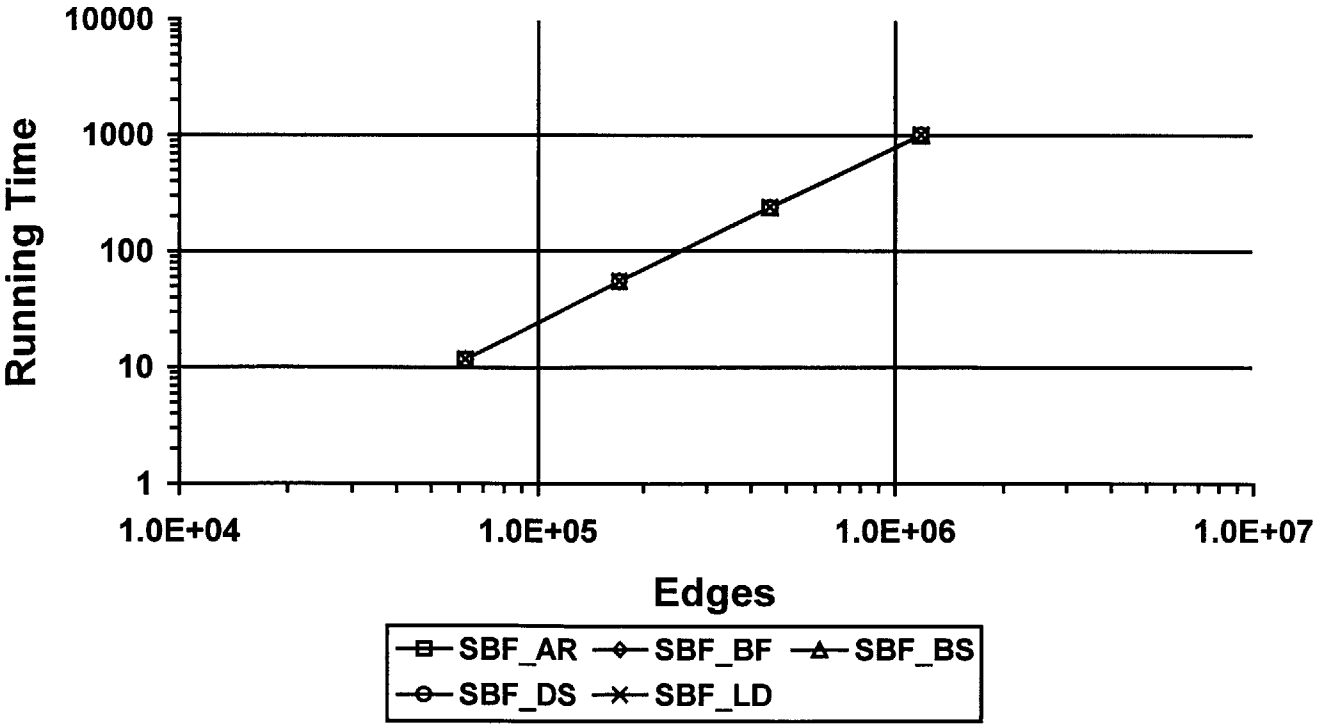
Aug. Paths (CUBE_GRID)



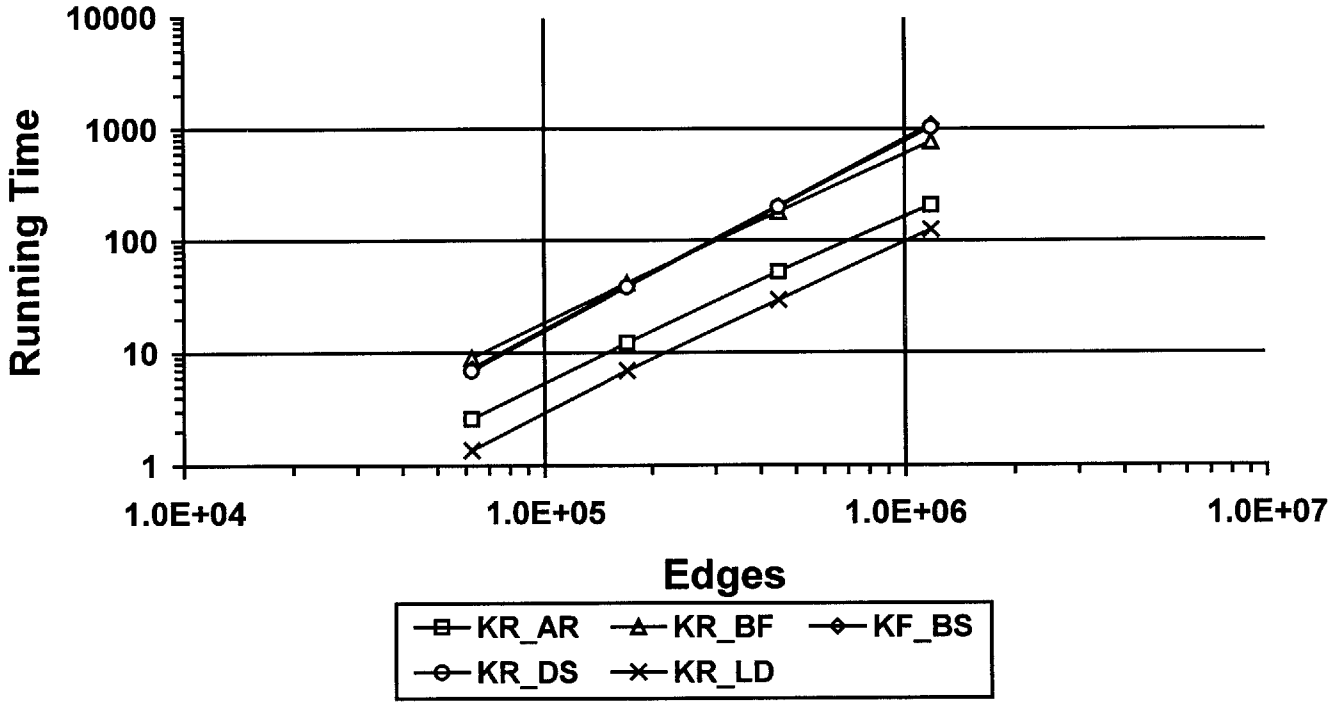
K-L Sparse Aug. Paths (CUBE_GRID)



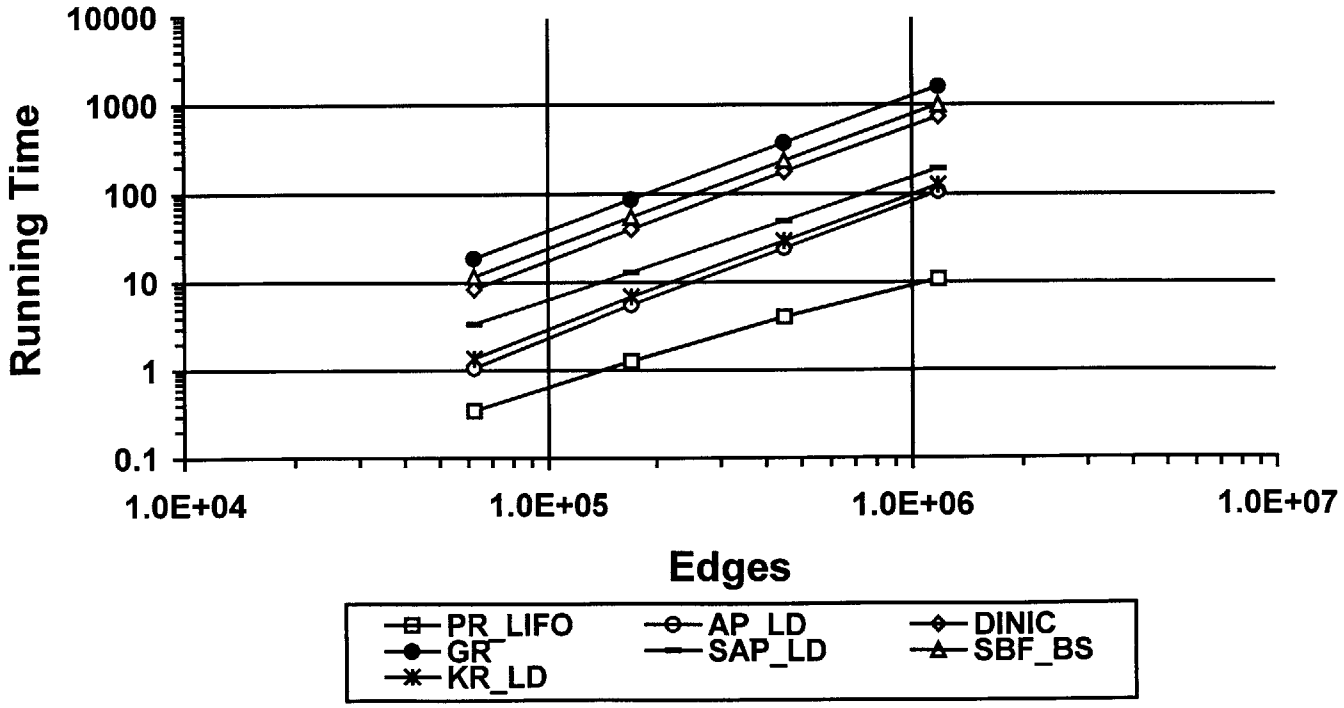
K-L Sparse Blocking Flows (CUBE_GRID)



Randomized Aug. Paths (CUBE_GRID)



Best Algorithms (CUBE_GRID)

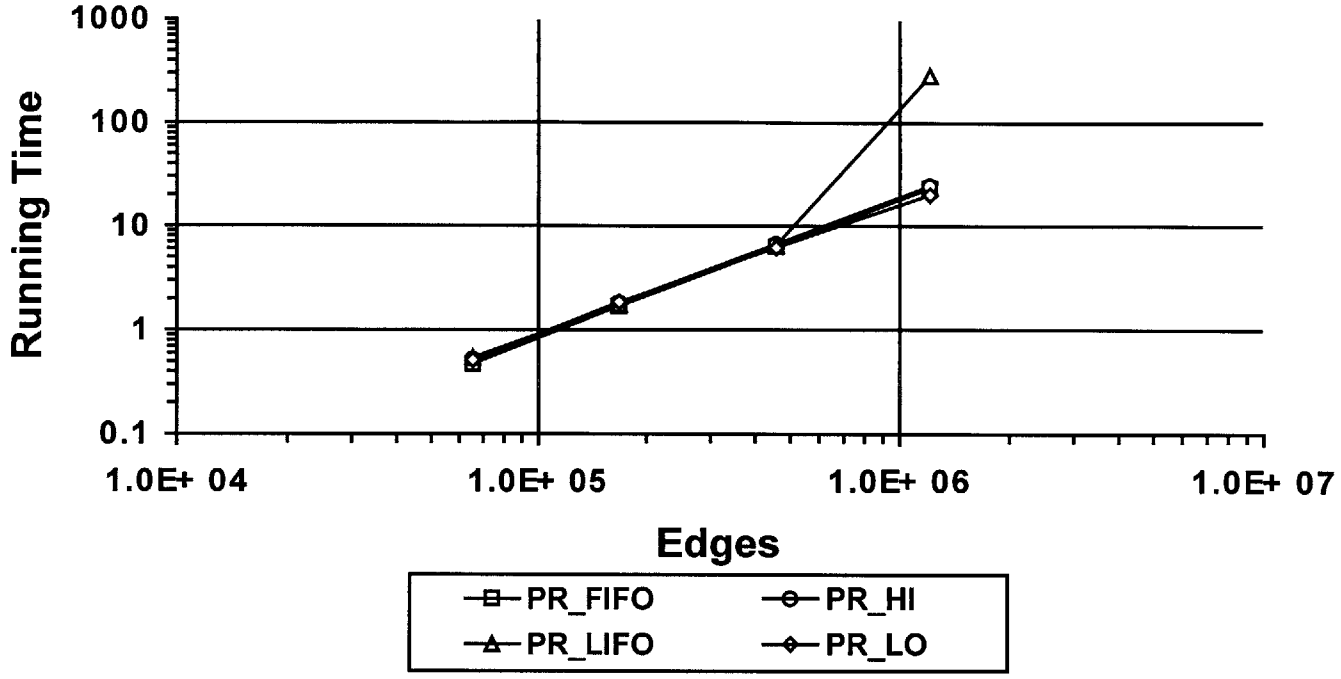


Code	Running Time on CUBE_GRID
PR_FIFO	$4.91 \times 10^{-7} \times m^{1.22}$
PR_HI	$1.50 \times 10^{-7} \times m^{1.36}$
PR_LIFO	$9.20 \times 10^{-7} \times m^{1.17}$
PR_LO	$1.13 \times 10^{-6} \times m^{1.16}$
AP_AR	$1.19 \times 10^{-7} \times m^{1.51}$
AP_BS	$4.55 \times 10^{-8} \times m^{1.71}$
AP_DS	$3.55 \times 10^{-8} \times m^{1.73}$
AP_LD	$3.66 \times 10^{-8} \times m^{1.56}$
DINIC	$4.33 \times 10^{-7} \times m^{1.52}$
GR	$1.09 \times 10^{-6} \times m^{1.51}$
SAP_AR	$7.54 \times 10^{-7} \times m^{1.42}$
SAP_BF	$8.43 \times 10^{-7} \times m^{1.51}$
SAP_BS	$7.97 \times 10^{-8} \times m^{1.69}$
SAP_DS	$1.03 \times 10^{-7} \times m^{1.66}$
SAP_LD	$8.71 \times 10^{-7} \times m^{1.37}$
SBF_AR	$6.40 \times 10^{-7} \times m^{1.51}$
SBF_BF	$6.46 \times 10^{-7} \times m^{1.51}$
SBF_BS	$6.55 \times 10^{-7} \times m^{1.51}$
SBF_DS	$6.41 \times 10^{-7} \times m^{1.52}$
SBF_LD	$6.38 \times 10^{-7} \times m^{1.52}$
KR_AR	$1.81 \times 10^{-7} \times m^{1.49}$
KR_BF	$4.94 \times 10^{-7} \times m^{1.51}$
KR_BS	$4.92 \times 10^{-8} \times m^{1.70}$
KR_DS	$4.83 \times 10^{-8} \times m^{1.70}$
KR_LD	$6.22 \times 10^{-8} \times m^{1.53}$

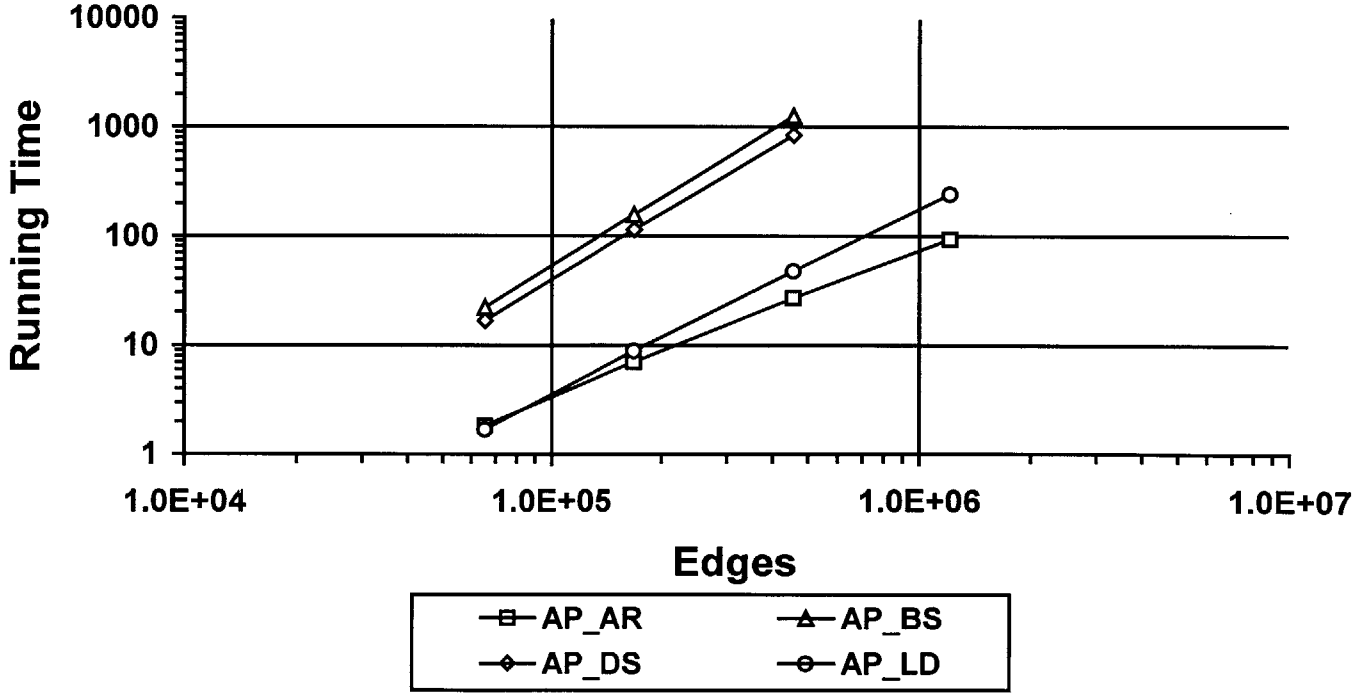
The final family that we used the Grid generator to produce was the WIDE_GRID family. This family produced some very odd data; it seems that some of the instances of the largest problem were very difficult for some of the algorithms. Essentially, we see the same behavior on this class as the other GRID classes, but more accentuated. The push-relabel algorithms perform similarly to each other; the augmenting paths algorithms with distance labels work reasonably well, but the others do not. The sparsified algorithms perform similar to their unsparsified cousins, but with less performance difference, larger running time constant factors, and smaller running time growth. The WIDE_GRID problems proved to be by far the most difficult for our augmenting paths algorithms.

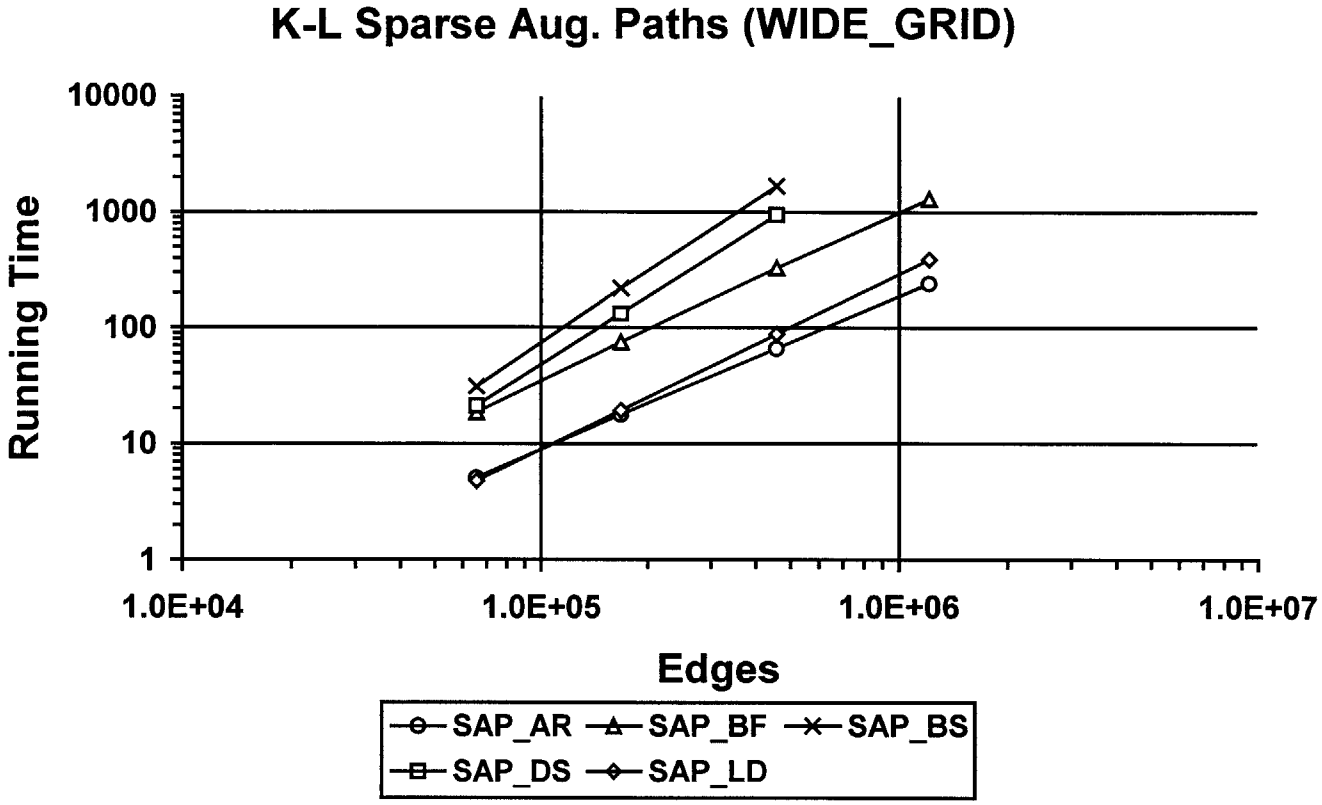
Overall, the families produced by the Grid generator provide an illustration first of the power of the push-relabel algorithms and second of the importance of using some form of distance labels in the algorithms. Given the results of these tests, it would be difficult to recommend any algorithm that did not involve distance labels. Additionally, these results are somewhat encouraging for the viability of the sparsification techniques. Although the graphs are already fairly sparse, so we have no right to expect huge performance improvements, we do in general get some speedup from sparsifying, whether it is the Karger-Levine way or the Karger randomized version.

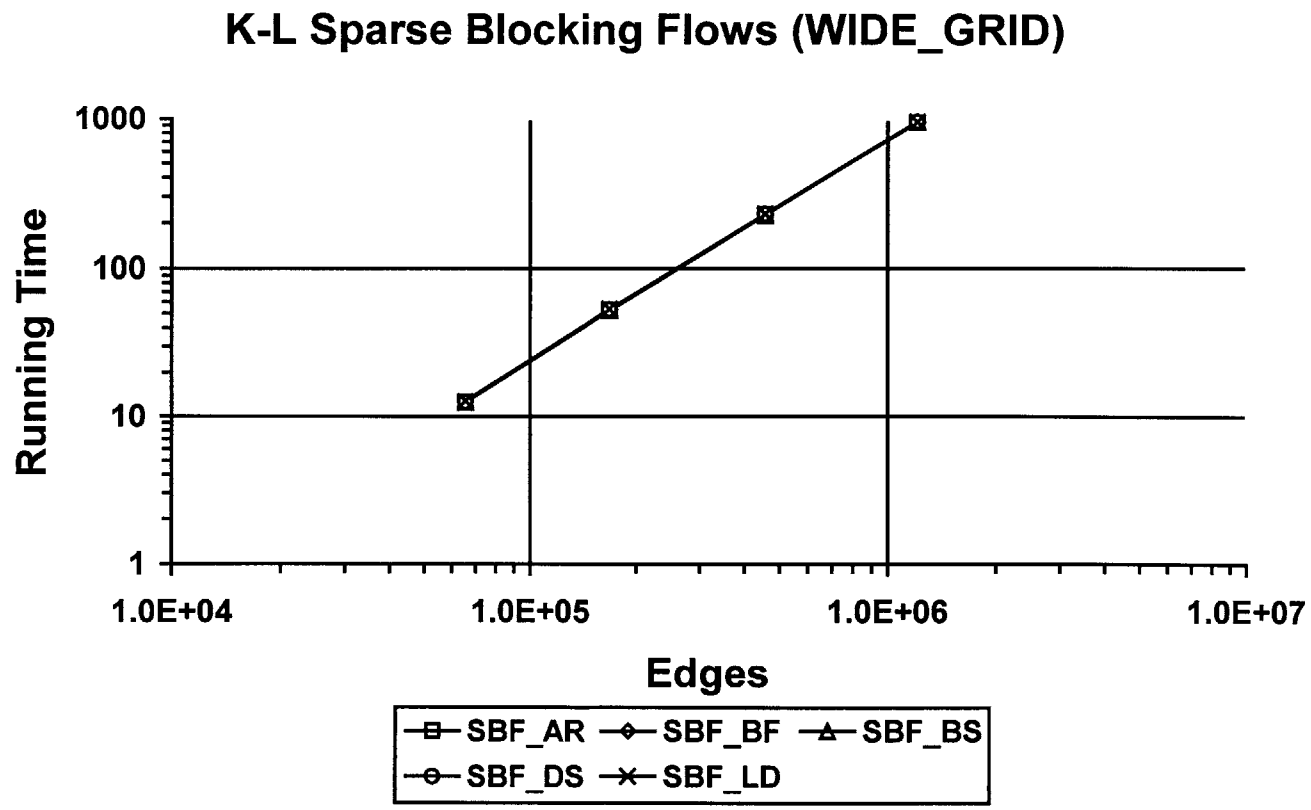
Push-Relabel (WIDE_GRID)



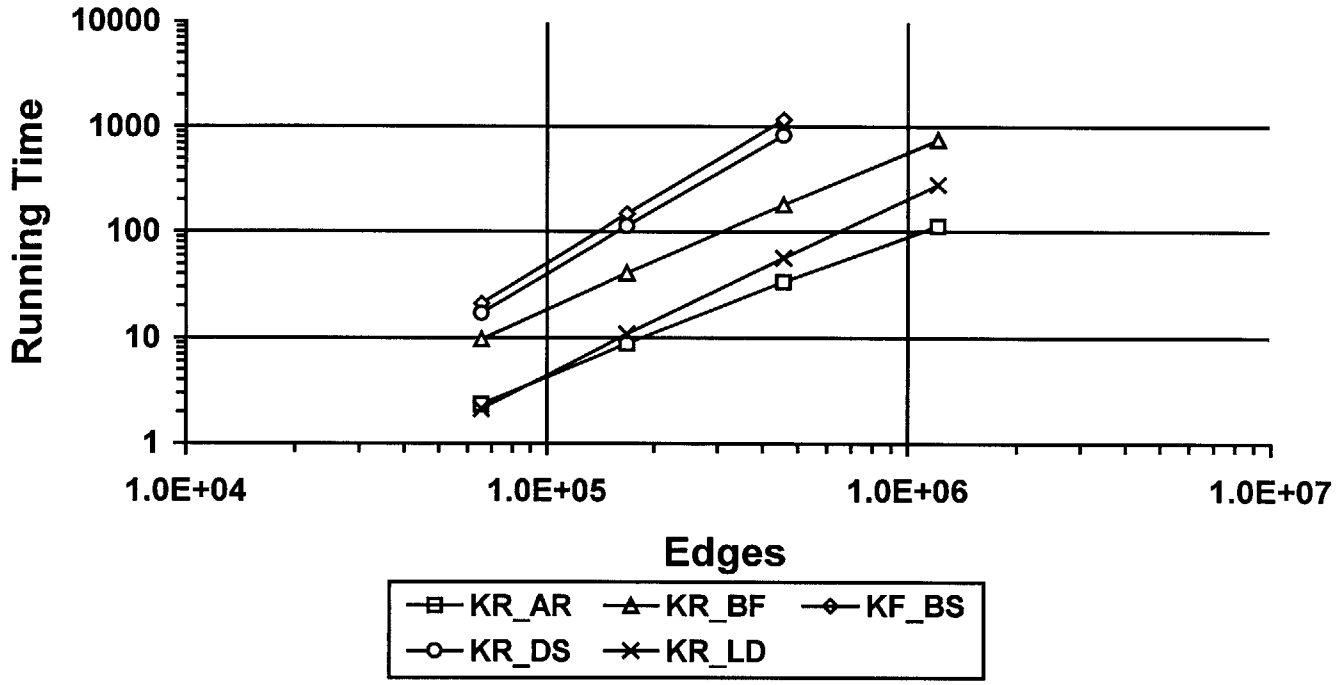
Aug. Paths (WIDE_GRID)



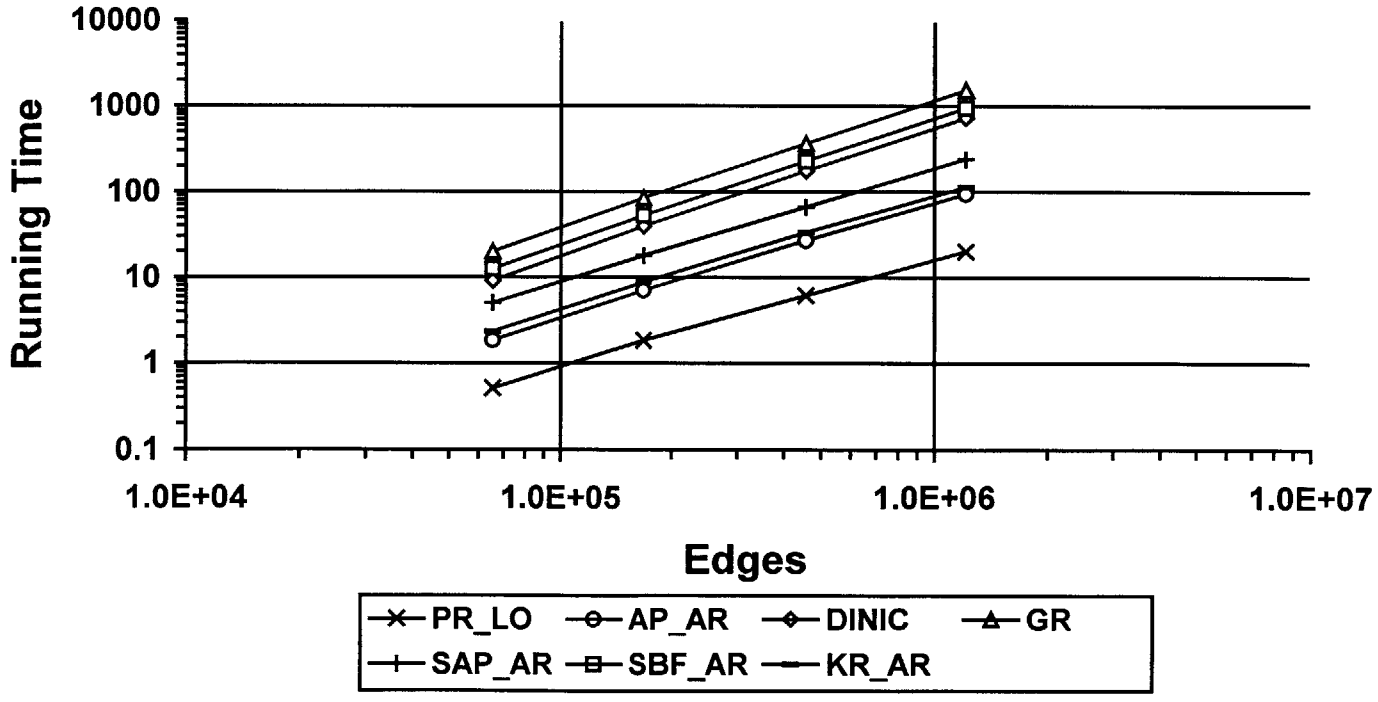

K-L Sparse Aug. Paths (WIDE_GRID)
Running Time
10000
1000
100
10
1
1.0E+04
1.0E+05
1.0E+06
1.0E+07
Edges
SAP_AR
SAP_BF
SAP_BS
SAP_DS
SAP_LD


K-L Sparse Blocking Flows (WIDE_GRID)
Running Time
1000
100
10
1
1.0E+04
1.0E+05
1.0E+06
1.0E+07
Edges
SBF_AR
SBF_BF
SBF_BS
SBF_DS
SBF_LD

Randomized Aug. Paths (WIDE_GRID)



Best Algorithms (WIDE_GRID)



Code	Running Time on WIDE_GRID
PR_FIFO	$1.79 \times 10^{-7} \times m^{1.33}$
PR_HI	$2.40 \times 10^{-7} \times m^{1.32}$
PR_LIFO	$3.76 \times 10^{-11} \times m^{2.06}$
PR_LO	$5.03 \times 10^{-7} \times m^{1.25}$
AP_AR	$6.33 \times 10^{-7} \times m^{1.35}$
AP_BS	$2.12 \times 10^{-9} \times m^{2.08}$
AP_DS	$3.26 \times 10^{-9} \times m^{2.02}$
AP_LD	$1.13 \times 10^{-8} \times m^{1.70}$
DINIC	$5.56 \times 10^{-7} \times m^{1.50}$
GR	$1.44 \times 10^{-6} \times m^{1.48}$
SAP_AR	$2.17 \times 10^{-6} \times m^{1.32}$
SAP_BF	$1.79 \times 10^{-6} \times m^{1.46}$
SAP_BS	$4.16 \times 10^{-9} \times m^{2.05}$
SAP_DS	$7.78 \times 10^{-9} \times m^{1.96}$
SAP_LD	$2.46 \times 10^{-7} \times m^{1.51}$
SBF_AR	$9.36 \times 10^{-7} \times m^{1.48}$
SBF_BF	$9.35 \times 10^{-7} \times m^{1.48}$
SBF_BS	$8.72 \times 10^{-7} \times m^{1.49}$
SBF_DS	$9.09 \times 10^{-7} \times m^{1.48}$
SBF_LD	$9.45 \times 10^{-7} \times m^{1.48}$
KR_AR	$9.72 \times 10^{-7} \times m^{1.33}$
KR_BF	$6.32 \times 10^{-7} \times m^{1.49}$
KR_BS	$2.27 \times 10^{-9} \times m^{2.07}$
KR_DS	$3.96 \times 10^{-9} \times m^{2.00}$
KR_LD	$1.97 \times 10^{-8} \times m^{1.67}$

4.2.2 Results by Algorithm

In this section we summarize the results for each algorithm, outlining where some of the outstanding issues with each algorithm are.

4.2.1.4 Augmenting Paths and Push-Relabel

These algorithms have already been studied elsewhere [6], so we keep our discussion brief. The augmenting paths algorithms can be divided into two classes: those which keep distance labels (augment-relabel and label-directed search) and those which do not (breadth-first search and depth-first search). After seeing the results of the Grid family, it seems that the breadth-first search and depth-first search options are not viable choices unless one has very exact information about the graphs that will be input to the algorithm. The label-directed search and augment-relabel algorithms, as they use distance labels, track the performance of the push-relabel algorithms to some extent, but usually don't seem to be either as incredibly quick or as painfully slow as push-relabel algorithms can be. It is interesting to speculate on why label-directed search often does significantly better than augment-relabel; an intuitively appealing idea is that the label-directed search is a natural combination of the strong points of push-relabel and augmenting paths, and thus is entitled to a certain amount of "hybrid vigor." It would be interesting to see how many of the heuristics used for push-relabel can be adapted to the label-directed search algorithm; our results suggest that this may be of both practical and theoretical importance. The other augmenting paths algorithms, however, do not seem worthy of consideration except for very specific input classes.

The push-relabel algorithms are arguably the best family of algorithms in the study. This should not come as a tremendous surprise; a number of studies have shown them to be practically viable, and Goldberg and Kennedy have proven some interesting and somewhat competitive theoretical bounds for the lowest-label vertex selection heuristic with global relabelling (which happens to be the best push-relabel algorithm in our study).

4.2.2.2 Dinic's Algorithm

Dinic's algorithm is essentially a disappointment. On the RANDOM, RANDOM_DENSE, and SHADED_DENSITY graphs it finishes in the middle of the pack; on the other graphs it places either last or second to last. Although it can be useful it graphs where there are not many different path lengths, it seems as though as a practical approach it must be largely abandoned unless interesting new heuristics can be developed for it.

4.2.2.3 Goldberg-Rao Sparse Blocking Flows

Our study of the Goldberg-Rao sparse blocking flows algorithm was disappointing, and not only because it was the only algorithm which performed worse than Dinic's algorithm. However, the important question is why this happens. The data shows that the problem is the effectiveness of blocking flow algorithms, not the fundamental sparsifying approach. Armed with this knowledge, it would be nice to attempt to use the concepts of the Goldberg-Rao algorithm within the framework of a more heuristically successful algorithm. Goldberg and Rao suggest that the the ideas in their paper are also applicable to an implementation using lowest-level vertex selection push-relabel without losing the theoretical time bounds. Due to time limitations, we were not able to try this idea. It is not clear from our data that it would be interesting (the Goldberg-Rao algorithm seems to in general to run slower than the normal blocking flows algorithm, although sometimes with slightly better growth rates). However, as we have found both lowest label vertex selection push relabel and sparsification to be effective, combining them is an appealing idea.

4.2.2.3 Karger-Levine Sparse Augmenting Paths

From the problem instances we reviewed, the Karger-Levine sparse augmenting paths is fairly promising as practical algorithm. Although it is not often one of the fastest algorithms, this is largely because of the size of its constant factors; with the exception of some of the Grid instances, implementations of the Karger-Levine sparse augmenting paths algorithm almost always have the best growth. Although we experimented with quite a few different details of the algorithm, heuristic improvements remain a distinct possibility. It would be nice to develop some heuristics which determine whether we run the sparsification, or decide that the graph is already sparse to an extent that would make it likely useless. Also, it seems possible that there are some cute variations based on the order of the edges connected to a vertex; in particular, it would be interesting to try to set up the algorithm so it always tries first to search along edges that it has not tried yet (i.e. between successive loops of SPARSEAUGMENT2). Finally, although we tried five different flow-finding subroutines, it still seems a distinct possibility that we can find a better one, as the flow-finding time dominates the sparsifying time on all the problems that the algorithm struggles with. It would be nice if we could find a way to adapt push-relabel strategies to work with this algorithm.

4.2.2.4 Karger-Levine Sparse Blocking Flows

Overall, our study did not provide very solid evidence on the performance of this algorithm. Most of our graph instances did not invoke the augmenting paths code, which reduces this algorithm to simply being Dinic's algorithm. Whether this is because very few interesting graphs will actually cause the augmenting paths code to run or because we were not sufficiently creative in inventing problem generators is not clear. On the one family where we do see the augmenting paths code invoked, the breadth-first search runs quite a bit faster (in terms of asymptotic growth rate) then any of the other algorithms, including the other versions of this one. It is not clear why this should happen. From our data, the most we can say is this algorithm can have interesting performance.

4.2.2.5 Karger Randomized Augmenting Paths

The Karger randomized augmenting paths implementations had similar characteristics to the Karger-Levine sparse augmenting paths algorithm. Both algorithms tended to have higher run time constants and lower run time growths than many of the other algorithms. However, the Karger randomized augmenting paths algorithms generally had slightly lower constants and slightly higher growth than the Karger-Levine sparse augmenting paths algorithms. Although it seems that it should be simple to create bad cases for the randomized algorithm (simply by making long, thin paths to the sink), the algorithm is surprisingly resilient to this; even if this causes it to find very little flow during the recursion, it still often shows up with a competitive running time. This is probably because the amount of time spent on the actual sparsification is small and the algorithm's running time (at least theoretically) should be dominated by the amount of time spent running augmenting paths at the top level of the recursion. Although the implementation of this algorithm is relatively straightforward, there is still reason to hope it can be improved. The choice of flow-finding algorithm is key, and it seems likely that if it was possible to adopt a variant of push-relabel to this problem it might work very well. Also, if we view randomization as a sparsification technique, we could imagine using random edge assignments to sparsify, but making the structure of the overall algorithm more similar to the Karger-Levine sparse augmenting paths algorithm. Similarly, we could use the spanning forests approach to sparsification and then recurse on two subproblems, each comprised of half of the spanning forests. These two ideas are blends of the Karger-Levine sparse augmenting paths algorithms and the Karger randomized augmenting paths algorithms and might have interesting properties.

Chapter 5

Conclusion

Our study implemented several new theoretical discoveries, attempting to determine how they compared in practice to several established algorithms with good theoretical and/or practical behavior. We also explored a number of heuristics and flow-finding subroutines for their suitability for use in conjunction with the new algorithms.

We determined that it is in general inadvisable to use augmenting paths algorithms that do not incorporate distance labels or to use blocking flow-based methods. We also established that the performance of push-relabel algorithms, particularly with the lowest-label vertex selection heuristic, is very strong, thus extending the work of [6] to undirected graphs. Finally, our results establish that sparsification, whether accomplished by randomized means or by spanning forests, is a technique that can result in practical improvements when used carefully.

Future work in this area should focus on ways to combine the fundamental building blocks that we have, namely augmenting paths routines, spanning forest and randomized sparsification, blocking flows, and push-relabel, in new ways to achieve better practical and/or theoretical bounds. In particular, due to the excellent performance in practice of the push-relabel algorithms, a version that incorporates some sparsification could well improve the practical (and possibly theoretical) state of the art for this problem. Finally, there are conjectures and guesses made in the discussion of the results; it would be nice to verify or deny some of these hypothesises.

It would also be nice to extend the set of generators that we considered and further study of the Karger-Levine sparse blocking flow algorithm is definitely needed.

Appendix A

Data Tables

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	2.28	1.76%	27526081	1.78%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	0.63	30.22%	7209207	30.02%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	2.32	2.91%	27916429	2.89%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	2.42	0.29%	28084190	0.02%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	1.88	0.24%	26683512	0.06%	N/A	N/A	N/A	N/A	1.88	0.24%	26683512	0.06%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	0.40	1.38%	3880399	0.40%	N/A	N/A	N/A	N/A	0.40	1.38%	3880399	0.40%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	0.27	2.06%	255908	1.95%	N/A	N/A	N/A	N/A	0.27	2.06%	255908	1.95%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	0.46	3.27%	6428929	2.88%	N/A	N/A	N/A	N/A	0.46	3.27%	6428929	2.88%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	1.77	0.64%	15692293	0.04%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1.77	0.64%	15692293	0.04%	33	N/A	N/A
GR	3.70	0.23%	23808709	0.03%	0.50	4.18%	4122656	0.00%	N/A	N/A	N/A	N/A	3.20	0.56%	19686053	0.03%	32	N/A	N/A
SAP_AR	1.35	1.42%	7484109	0.27%	0.83	1.61%	3738100	0.27%	0.52	3.71%	3746009	0.30%	N/A	N/A	N/A	N/A	N/A	4	7
SAP_BF	1.36	2.60%	5673852	0.28%	0.88	3.88%	3618639	0.21%	N/A	N/A	N/A	N/A	0.48	0.94%	2055212	0.69%	39	4	7
SAP_BS	0.96	2.15%	4190515	0.18%	0.83	2.49%	3667328	0.15%	0.13	5.44%	523187	1.14%	N/A	N/A	N/A	N/A	N/A	4	7
SAP_DS	0.90	2.26%	3989171	0.45%	0.84	2.57%	3748658	0.37%	0.06	13.49%	240513	2.02%	N/A	N/A	N/A	N/A	N/A	4	7
SAP_LD	0.91	3.71%	4167255	3.26%	0.82	3.15%	3807373	0.36%	0.08	32.16%	359914	40.82%	N/A	N/A	N/A	N/A	N/A	4	7
SBF_AR	2.50	0.87%	11743052	0.71%	0.59	1.51%	2921844	0.19%	0.21	7.09%	1685406	5.05%	1.69	0.42%	7135813	0.00%	23	2	2
SBF_BF	2.48	0.73%	10884426	0.05%	0.60	3.05%	2920574	0.18%	N/A	N/A	N/A	N/A	1.89	0.24%	7963852	0.04%	35	2	2
SBF_BS	2.33	0.83%	10299681	0.05%	0.59	1.86%	2920842	0.18%	0.05	0.00%	243025	0.01%	1.69	0.72%	7135813	0.00%	23	2	2
SBF_DS	2.35	0.63%	10195161	0.14%	0.59	1.92%	2963792	0.17%	0.02	20.33%	95556	11.92%	1.74	0.32%	7135813	0.00%	23	2	2
SBF_LD	2.37	0.97%	10350283	1.49%	0.59	2.40%	2943181	0.57%	0.04	55.90%	271289	61.21%	1.74	0.32%	7135813	0.00%	23	2	2
KR_AR	2.65	0.21%	27138170	0.05%	0.21	5.53%	1077638	1.49%	2.44	0.29%	26060533	0.03%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	2.27	0.24%	16746536	0.09%	0.21	5.27%	926600	0.46%	N/A	N/A	N/A	N/A	2.06	0.41%	15819937	0.12%	39	N/A	N/A
KR_BS	0.85	2.04%	5636460	0.13%	0.28	2.97%	1383051	0.73%	0.57	2.30%	4253411	0.31%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	0.72	1.25%	4781106	1.30%	0.28	2.97%	1382823	0.66%	0.43	2.06%	3398284	1.57%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	0.81	1.23%	7152645	1.47%	0.19	2.94%	857211	0.93%	0.62	2.15%	6295467	1.75%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

KARZ_64K: $m = 64416$, $n = 7074$, $v = 32$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	8.74	0.89%	105044163	1.01%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	2.83	22.54%	32538947	22.50%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	8.80	1.24%	105875156	1.06%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	9.06	0.21%	105528715	0.01%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	7.15	0.21%	99832630	0.12%	N/A	N/A	N/A	N/A	7.15	0.21%	99832630	0.12%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	1.46	0.00%	14373624	0.29%	N/A	N/A	N/A	N/A	1.46	0.00%	14373624	0.29%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	0.98	1.52%	8977628	1.72%	N/A	N/A	N/A	N/A	0.98	1.52%	8977628	1.72%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	1.74	2.13%	23001129	2.27%	N/A	N/A	N/A	N/A	1.74	2.13%	23001129	2.27%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	6.36	0.59%	57860105	0.03%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	6.36	0.59%	57860105	0.03%	46	N/A	N/A
GR	13.54	0.18%	87569195	0.02%	1.70	1.89%	15021495	0.00%	N/A	N/A	N/A	N/A	11.84	0.26%	72547700	0.02%	45	N/A	N/A
SAP_AR	3.78	2.89%	19937937	1.71%	2.07	3.63%	9435348	2.08%	1.71	2.26%	10502588	2.39%	N/A	N/A	N/A	N/A	N/A	4	8
SAP_BF	3.74	2.33%	14707287	1.60%	2.24	3.11%	9125605	2.10%	N/A	N/A	N/A	N/A	1.50	1.25%	5581683	0.80%	53	4	8
SAP_BS	2.53	2.83%	10724530	1.90%	2.10	3.56%	9259068	2.11%	0.43	3.02%	1465462	0.63%	N/A	N/A	N/A	N/A	N/A	4	8
SAP_DS	2.26	5.10%	9994299	2.46%	2.09	5.88%	9337656	2.66%	0.18	4.70%	656643	3.52%	N/A	N/A	N/A	N/A	N/A	4	8
SAP_LD	2.36	4.79%	10487839	3.06%	2.14	3.77%	9602539	2.19%	0.22	23.84%	885345	30.13%	N/A	N/A	N/A	N/A	N/A	4	8
SBF_AR	8.57	0.40%	39002876	0.08%	1.56	1.28%	7715131	0.39%	0.67	2.68%	4514047	0.03%	6.34	0.30%	26773712	0.00%	33	2	2
SBF_BF	8.50	0.67%	36581021	0.07%	1.55	0.58%	7713036	0.37%	N/A	N/A	N/A	N/A	6.95	0.77%	28867985	0.01%	48	2	2
SBF_BS	8.11	0.61%	35099275	0.08%	1.57	1.52%	7713652	0.37%	0.15	3.75%	611911	0.00%	6.39	0.62%	26773712	0.00%	33	2	2
SBF_DS	8.25	0.74%	34819096	0.07%	1.56	1.61%	7817644	0.38%	0.06	17.82%	227740	11.85%	6.63	0.63%	26773712	0.00%	33	2	2
SBF_LD	8.28	0.25%	34978982	0.54%	1.57	1.16%	7764809	0.49%	0.08	38.21%	440461	42.72%	6.63	0.36%	26773712	0.00%	33	2	2
KR_AR	9.95	0.24%	100909831	0.03%	0.54	1.54%	2780881	0.29%	9.40	0.21%	98128950	0.04%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	8.21	0.30%	60648421	0.06%	0.54	2.13%	2439293	0.67%	N/A	N/A	N/A	N/A	7.67	0.43%	58209129	0.05%	52	N/A	N/A
KR_BS	2.79	0.36%	19070133	0.07%	0.72	2.31%	3702020	0.41%	2.07	0.55%	15368114	0.10%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	2.26	0.85%	15226764	0.66%	0.73	1.78%	3689966	0.39%	1.53	1.36%	11536799	0.90%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	2.85	3.10%	25582072	3.37%	0.50	0.00%	2262089	0.36%	2.35	3.76%	23320028	3.69%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

KARZ_167K: $m = 166905$, $n = 14042$, $v = 45$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	34.78	0.27%	418627146	0.02%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	8.67	29.40%	100828804	29.55%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	34.58	0.99%	420555682	0.77%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	35.52	0.44%	418553514	0.01%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	28.52	0.34%	393402399	0.04%	N/A	N/A	N/A	N/A	28.52	0.34%	393402399	0.04%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	5.65	0.30%	56356751	0.10%	N/A	N/A	N/A	N/A	5.65	0.30%	56356751	0.10%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	3.75	0.77%	33541838	0.86%	N/A	N/A	N/A	N/A	3.75	0.77%	33541838	0.86%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	6.89	2.30%	88029275	2.35%	N/A	N/A	N/A	N/A	6.89	2.30%	88029275	2.35%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	25.57	0.66%	226205120	0.06%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	25.57	0.66%	226205120	0.06%	65	N/A	N/A
GR	52.52	0.28%	341638976	0.04%	6.46	0.86%	58171456	0.00%	N/A	N/A	N/A	N/A	46.06	0.20%	283467520	0.05%	64	N/A	N/A
SAP_AR	11.19	2.55%	55625931	1.03%	5.47	2.51%	24732467	1.93%	5.72	3.45%	30893465	1.02%	N/A	N/A	N/A	N/A	N/A	4	9
SAP_BF	10.11	1.59%	39604213	1.38%	5.76	2.95%	23918587	1.86%	N/A	N/A	N/A	N/A	4.34	1.00%	15685625	0.66%	73	4	9
SAP_BS	6.90	2.19%	28573391	1.57%	5.55	2.91%	24407756	1.80%	1.36	0.84%	4165636	0.43%	N/A	N/A	N/A	N/A	N/A	4	9
SAP_DS	6.13	3.09%	26344957	2.47%	5.50	3.10%	24442508	2.38%	0.63	5.38%	1902449	5.82%	N/A	N/A	N/A	N/A	N/A	4	8
SAP_LD	6.18	3.74%	27845060	2.89%	5.41	2.47%	24720912	1.38%	0.77	16.07%	3124212	18.44%	N/A	N/A	N/A	N/A	N/A	4	8
SBF_AR	32.22	0.25%	142338995	0.27%	4.39	1.10%	21458822	0.31%	2.19	3.16%	14046888	2.75%	25.64	0.32%	106833304	0.00%	48	2	2
SBF_BF	31.46	0.30%	134490467	0.05%	4.39	1.19%	21455690	0.30%	N/A	N/A	N/A	N/A	27.08	0.35%	113034778	0.01%	48	2	2
SBF_BS	30.38	0.35%	130162558	0.05%	4.36	0.62%	21456370	0.30%	0.47	2.41%	1872885	0.00%	25.54	0.32%	106833304	0.00%	48	2	2
SBF_DS	30.97	0.41%	129164025	0.02%	4.40	0.55%	21712154	0.29%	0.20	6.45%	618567	8.04%	26.37	0.46%	106833304	0.00%	48	2	2
SBF_LD	31.29	0.65%	130929045	0.26%	4.40	1.18%	21566096	0.40%	0.46	13.67%	2529645	15.63%	26.44	0.71%	106833304	0.00%	48	2	2
KR_AR	39.33	0.10%	397046396	0.09%	1.46	0.68%	7616829	0.34%	37.87	0.12%	389429568	0.10%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	32.38	0.22%	237503156	0.06%	1.78	1.00%	8919770	0.60%	N/A	N/A	N/A	N/A	30.59	0.25%	228583388	0.05%	79	N/A	N/A
KR_BS	9.73	0.27%	69344463	0.05%	1.97	1.05%	10246319	0.40%	7.76	0.25%	59098145	0.05%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	7.39	0.79%	50931786	0.94%	1.98	0.62%	10249853	0.18%	5.41	1.10%	40681934	1.17%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	10.42	0.70%	95638981	0.90%	1.35	1.10%	6238715	0.34%	9.07	0.81%	89400332	0.96%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

KARZ_454K: $m = 454464$, $n = 28482$, $v = 64$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	135.06	1.42%	1576672692	0.61%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	40.21	18.31%	446372313	18.03%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	134.67	1.38%	1592594013	1.49%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	140.99	2.45%	1593639901	0.05%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	111.18	0.34%	1497004148	0.11%	N/A	N/A	N/A	N/A	111.18	0.34%	1497004148	0.11%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	21.34	0.39%	213455039	0.14%	N/A	N/A	N/A	N/A	21.34	0.39%	213455039	0.14%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	14.07	0.81%	123303432	1.07%	N/A	N/A	N/A	N/A	14.07	0.81%	123303432	1.07%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	26.66	2.01%	325658726	1.53%	N/A	N/A	N/A	N/A	26.66	2.01%	325658726	1.53%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	96.28	0.14%	856624109	0.02%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	96.28	0.14%	856624109	0.02%	91	N/A	N/A
GR	198.27	0.12%	1291497449	0.01%	24.15	0.44%	218561490	0.00%	N/A	N/A	N/A	N/A	174.12	0.10%	1072845959	0.01%	90	N/A	N/A
SAP_AR	34.87	0.39%	164270846	0.49%	17.49	0.39%	76757232	0.28%	17.38	0.57%	87513614	0.78%	N/A	N/A	N/A	N/A	N/A	5	11
SAP_BF	30.85	0.75%	118997023	0.61%	18.28	1.62%	74112216	0.90%	N/A	N/A	N/A	N/A	12.57	0.86%	44884807	0.21%	101	5	11
SAP_BS	21.29	1.32%	86177416	1.09%	17.19	1.69%	74662565	1.25%	4.11	0.47%	11514851	0.33%	N/A	N/A	N/A	N/A	N/A	5	11
SAP_DS	19.38	1.66%	81503292	1.33%	17.39	1.68%	76025420	1.28%	1.98	2.48%	5477871	3.88%	N/A	N/A	N/A	N/A	N/A	5	11
SAP_LD	20.05	34.50%	87295422	2.81%	17.53	1.97%	77205020	0.89%	2.53	18.29%	10090492	21.51%	N/A	N/A	N/A	N/A	N/A	5	11
SBF_AR	116.55	0.46%	509909123	0.04%	12.06	0.41%	58611300	0.18%	7.39	1.71%	44547919	0.32%	97.11	0.64%	406749928	0.00%	68	2	2
SBF_BF	114.46	0.60%	483715672	0.02%	12.12	0.48%	58619383	0.19%	N/A	N/A	N/A	N/A	102.34	0.64%	425096289	0.01%	93	2	2
SBF_BS	110.80	0.63%	471088110	0.02%	12.04	0.63%	58623916	0.19%	1.47	1.47%	5714265	0.00%	97.29	0.71%	406749928	0.00%	68	2	2
SBF_DS	114.45	0.49%	467898013	0.03%	12.15	0.56%	59276007	0.11%	0.64	4.74%	1872078	7.87%	101.65	0.51%	406749928	0.00%	68	2	2
SBF_LD	115.13	1.05%	471336748	0.80%	12.13	0.47%	58894544	0.36%	1.06	64.10%	5692276	69.53%	101.95	0.54%	406749928	0.00%	68	2	2
KR_AR	156.67	0.02%	1512332026	0.00%	4.82	0.01%	28118634	0.00%	151.85	0.02%	1484213393	0.00%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	114.93	0.41%	887097363	0.04%	4.80	0.65%	24062131	0.58%	N/A	N/A	N/A	N/A	110.13	0.42%	863035234	0.04%	105	N/A	N/A
KR_BS	35.86	0.28%	255818929	0.04%	6.30	0.91%	32658687	0.08%	21.65	0.18%	180594583	0.04%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	26.20	0.48%	179851813	0.18%	6.32	0.73%	32667639	0.46%	19.88	0.54%	147184175	0.29%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	38.45	0.57%	348951695	0.35%	4.42	0.43%	22240765	0.37%	34.03	0.65%	326711022	0.40%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

KARZ_1215K: $m = 1214730$, $n = 56432$, $v = 90$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	0.49	3.37%	8818396	3.09%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	0.09	14.27%	1636144	11.01%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	0.57	10.30%	10260913	9.03%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	0.59	2.59%	10429258	1.73%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	0.55	1.52%	10723448	0.01%	N/A	N/A	N/A	N/A	0.55	1.52%	10723448	0.01%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	0.11	3.99%	1353232	0.19%	N/A	N/A	N/A	N/A	0.11	3.99%	1353232	0.19%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	0.03	15.97%	363719	10.87%	N/A	N/A	N/A	N/A	0.03	15.97%	363719	10.87%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	0.02	34.23%	242603	3.69%	N/A	N/A	N/A	N/A	0.02	34.23%	242603	3.69%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	0.60	2.73%	5576570	0.03%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.60	2.73%	5576570	0.03%	33	N/A	N/A
GR	1.36	0.40%	8661050	0.02%	0.22	8.21%	1566752	0.00%	N/A	N/A	N/A	N/A	1.14	1.15%	7094298	0.03%	32	N/A	N/A
SAP_AR	0.37	3.31%	2110432	0.64%	0.27	4.79%	1346969	0.44%	0.10	4.56%	763463	1.38%	N/A	N/A	N/A	N/A	N/A	3	5
SAP_BF	0.48	3.90%	2130175	0.28%	0.31	5.47%	1304418	0.50%	N/A	N/A	N/A	N/A	0.17	5.14%	825757	0.09%	37	3	5
SAP_BS	0.31	6.17%	1497916	0.46%	0.27	6.42%	1323612	0.51%	0.04	10.65%	174304	0.07%	N/A	N/A	N/A	N/A	N/A	3	5
SAP_DS	0.28	5.34%	1388962	0.17%	0.27	5.86%	1337817	0.34%	0.01	39.12%	51145	9.05%	N/A	N/A	N/A	N/A	N/A	3	5
SAP_LD	0.30	4.32%	1426456	0.62%	0.28	7.30%	1361024	0.41%	0.02	46.48%	65464	5.75%	N/A	N/A	N/A	N/A	N/A	3	5
SBF_AR	0.83	1.79%	4020210	0.14%	0.21	4.18%	1141595	0.49%	0.02	20.33%	181387	0.09%	0.59	0.76%	2697232	0.00%	24	2	2
SBF_BF	0.86	1.04%	4074749	0.15%	0.22	9.60%	1141595	0.50%	N/A	N/A	N/A	N/A	0.65	2.01%	2933154	0.02%	35	2	2
SBF_BS	0.83	0.00%	3883323	0.15%	0.22	3.21%	1141595	0.50%	0.01	37.27%	44496	0.01%	0.60	0.75%	2697232	0.00%	24	2	2
SBF_DS	0.84	1.55%	3866610	0.13%	0.22	5.98%	1146674	0.48%	0.01	91.29%	22704	7.06%	0.62	0.72%	2697232	0.00%	24	2	2
SBF_LD	0.84	1.88%	3879913	0.24%	0.22	8.06%	1145151	0.49%	0.01	91.29%	37530	17.05%	0.61	0.73%	2697232	0.00%	24	2	2
KR_AR	0.81	1.40%	11166979	0.04%	0.06	9.78%	241481	1.71%	0.76	1.10%	10925499	0.03%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	0.73	0.75%	5800230	0.06%	0.05	0.00%	215545	1.42%	N/A	N/A	N/A	N/A	0.68	0.80%	5584686	0.04%	35	N/A	N/A
KR_BS	0.25	1.80%	1833816	0.22%	0.07	7.40%	393951	0.37%	0.17	3.15%	1439866	0.21%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	0.15	3.75%	1001503	3.57%	0.08	8.84%	394770	0.73%	0.07	17.28%	606734	6.09%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	0.08	6.52%	525721	2.43%	0.05	0.00%	206766	2.29%	0.03	16.11%	318988	2.78%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

KARZ_DENSE_24K: $m = 24480$, $n = 5282$, $v = 32$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	3.10	3.55%	48237269	3.47%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	0.68	39.99%	10559691	40.16%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	3.42	3.32%	53165616	3.16%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	3.46	4.33%	53397665	4.30%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	3.31	1.08%	55013068	1.25%	N/A	N/A	N/A	N/A	3.31	1.08%	55013068	1.25%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	0.62	0.72%	6533212	0.06%	N/A	N/A	N/A	N/A	0.62	0.72%	6533212	0.06%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	0.17	7.76%	1821740	8.32%	N/A	N/A	N/A	N/A	0.17	7.76%	1821740	8.32%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	0.06	13.49%	802726	3.88%	N/A	N/A	N/A	N/A	0.06	13.49%	802726	3.88%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	3.00	0.94%	26556522	0.03%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	3.00	0.94%	26556522	0.03%	46	N/A	N/A
GR	6.27	0.07%	40672278	0.02%	0.84	2.73%	7137225	0.00%	N/A	N/A	N/A	N/A	5.43	0.42%	33535053	0.03%	45	N/A	N/A
SAP_AR	1.08	1.54%	5980518	0.50%	0.80	2.10%	3941553	0.29%	0.29	1.55%	2038965	1.03%	N/A	N/A	N/A	N/A	N/A	3	5
SAP_BF	1.42	1.35%	6029437	0.27%	0.84	2.46%	3797863	0.31%	N/A	N/A	N/A	N/A	0.58	2.26%	2231574	0.23%	50	3	5
SAP_BS	0.93	2.79%	4309364	0.27%	0.79	3.28%	3836073	0.30%	0.14	7.14%	473291	0.04%	N/A	N/A	N/A	N/A	N/A	3	5
SAP_DS	0.82	2.77%	3991937	0.25%	0.80	2.86%	3872953	0.14%	0.02	37.27%	118983	7.35%	N/A	N/A	N/A	N/A	N/A	3	5
SAP_LD	0.84	2.23%	4136379	1.34%	0.78	2.10%	3969180	0.32%	0.06	14.43%	167244	28.57%	N/A	N/A	N/A	N/A	N/A	3	5
SBF_AR	3.51	0.58%	16436765	0.03%	0.71	2.23%	3718721	0.13%	0.08	6.52%	691064	0.17%	2.72	0.16%	12026985	0.00%	32	2	2
SBF_BF	3.59	0.41%	16483929	0.03%	0.69	2.14%	3718635	0.13%	N/A	N/A	N/A	N/A	2.90	0.24%	12765294	0.01%	48	2	2
SBF_BS	3.44	0.38%	15895950	0.03%	0.68	0.66%	3718654	0.13%	0.04	11.77%	150311	0.01%	2.73	0.33%	12026985	0.00%	32	2	2
SBF_DS	3.53	0.76%	15817990	0.05%	0.70	2.86%	3735981	0.15%	0.01	37.27%	55025	8.03%	2.82	0.35%	12026985	0.00%	32	2	2
SBF_LD	3.57	0.37%	15871134	0.11%	0.72	0.98%	3733187	0.12%	0.02	20.33%	110963	15.63%	2.83	0.56%	12026985	0.00%	32	2	2
KR_AR	4.50	0.16%	56493395	0.07%	0.19	2.94%	813027	1.74%	4.31	0.13%	55680	0.06%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	3.70	0.12%	27838163	0.03%	0.24	3.67%	1095776	0.58%	N/A	N/A	N/A	N/A	3.46	0.24%	26742388	0.03%	52	N/A	N/A
KR_BS	1.11	0.49%	8196826	0.06%	0.26	4.32%	1351629	0.32%	0.85	0.83%	6845198	0.05%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	0.55	3.71%	3739141	4.24%	0.28	2.97%	1354633	0.29%	0.27	5.86%	2384510	6.53%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	0.29	1.92%	1778702	2.50%	0.18	3.93%	696589	1.66%	0.11	5.17%	1082160	3.42%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

KARZ_DENSE_79K: $m = 79302$, $n = 10504$, $v = 45$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	16.63	4.91%	256092348	4.83%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	4.59	44.97%	67636950	45.36%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	18.24	1.59%	277831757	1.54%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	19.07	1.49%	280383303	1.52%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	17.39	1.21%	28377412	1.24%	N/A	N/A	N/A	N/A	17.39	1.21%	28377412	1.24%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	3.07	0.23%	33327146	0.06%	N/A	N/A	N/A	N/A	3.07	0.23%	33327146	0.06%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	0.87	8.27%	8744534	8.93%	N/A	N/A	N/A	N/A	0.87	8.27%	8744534	8.93%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	0.21	2.15%	2799292	1.29%	N/A	N/A	N/A	N/A	0.21	2.15%	2799292	1.29%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	15.09	1.20%	134442053	0.04%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	15.09	1.20%	134442053	0.04%	65	N/A	N/A
GR	30.83	0.18%	204099013	0.03%	3.95	0.75%	35102784	0.00%	N/A	N/A	N/A	N/A	5.43	0.42%	33535053	0.03%	45	N/A	N/A
SAP_AR	3.02	3.28%	17573857	2.26%	2.22	3.86%	12201916	2.52%	0.80	2.40%	5371941	1.96%	N/A	N/A	N/A	N/A	N/A	3	5
SAP_BF	4.06	2.37%	17861843	1.89%	2.32	4.19%	11699888	2.68%	N/A	N/A	N/A	N/A	1.74	0.94%	6161955	0.42%	69	3	5
SAP_BS	2.68	3.00%	13116094	2.56%	2.22	3.19%	11783583	2.81%	0.45	2.96%	1332511	0.34%	N/A	N/A	N/A	N/A	N/A	3	5
SAP_DS	2.30	3.23%	12143367	2.99%	2.23	3.75%	11842110	2.82%	0.08	17.65%	301257	15.05%	N/A	N/A	N/A	N/A	N/A	3	5
SAP_LD	2.38	4.32%	12814194	3.18%	2.22	3.90%	12238500	2.54%	0.16	12.50%	575757	24.26%	N/A	N/A	N/A	N/A	N/A	3	5
SBF_AR	15.18	0.22%	69968645	0.03%	2.13	0.71%	11932518	0.21%	0.30	1.50%	2300912	0.20%	12.75	0.23%	55735221	0.00%	42	2	2
SBF_BF	15.51	0.40%	69941956	0.03%	2.15	1.64%	11932700	0.20%	N/A	N/A	N/A	N/A	13.36	0.35%	58009256	0.00%	67	2	2
SBF_BS	14.99	0.24%	68166121	0.04%	2.12	0.98%	11932756	0.21%	0.15	6.67%	498143	0.01%	12.73	0.28%	55735221	0.00%	42	2	2
SBF_DS	15.31	0.25%	67857968	0.03%	2.12	1.31%	11977925	0.19%	0.03	16.11%	144821	6.48%	13.16	0.24%	55735221	0.00%	42	2	2
SBF_LD	15.32	0.15%	67983887	0.08%	2.13	1.10%	11962900	0.24%	0.06	7.21%	285766	19.57%	13.13	0.12%	55735221	0.00%	42	2	2
KR_AR	23.80	0.97%	290089349	0.99%	0.85	1.34%	4525193	0.42%	22.94	1.00%	285564156	1.01%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	18.74	0.20%	141221043	0.03%	1.04	1.05%	5242742	0.34%	N/A	N/A	N/A	N/A	17.70	0.19%	135978303	0.03%	79	N/A	N/A
KR_BS	5.34	0.38%	40987819	0.03%	1.12	2.79%	6044246	0.13%	4.22	0.27%	34943574	0.04%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	2.57	3.08%	17694288	3.81%	1.11	1.37%	6054439	0.21%	1.46	5.48%	11639850	5.86%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	1.14	0.48%	6934985	1.45%	0.79	1.06%	3662854	0.73%	0.34	1.59%	3272196	2.35%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

KARZ_DENSE_274K: $m = 274240$, $n = 21314$, $v = 64$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	Avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	99.50	5.15%	1353233052	5.15%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	21.43	45.27%	281261185	45.57%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	105.64	0.74%	1431948359	0.70%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	109.24	0.56%	1433876289	0.53%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	98.43	0.59%	1439810328	0.58%	N/A	N/A	N/A	N/A	98.43	0.59%	1439810328	0.58%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	15.98	0.10%	172680256	0.05%	N/A	N/A	N/A	N/A	15.98	0.10%	172680256	0.05%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	4.32	5.67%	43807630	5.85%	N/A	N/A	N/A	N/A	4.32	5.67%	43807630	5.85%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	0.78	1.14%	10160335	0.26%	N/A	N/A	N/A	N/A	0.78	1.14%	10160335	0.26%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	74.28	0.49%	693818476	0.02%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	74.28	0.49%	693818476	0.02%	91	N/A	N/A
GR	156.35	0.12%	1047501712	0.01%	19.61	0.63%	177829650	0.00%	N/A	N/A	N/A	N/A	136.74	0.05%	869672062	0.01%	90	N/A	N/A
SAP_AR	9.52	0.67%	56762751	0.28%	7.22	0.77%	42250037	0.25%	2.30	1.36%	14512714	1.16%	N/A	N/A	N/A	N/A	N/A	3	6
SAP_BF	12.53	0.41%	57169361	0.21%	7.63	1.01%	40421372	0.30%	N/A	N/A	N/A	N/A	4.90	0.76%	16747989	0.14%	96	3	6
SAP_BS	8.50	0.73%	44346955	0.23%	7.17	0.91%	40662873	0.26%	1.33	0.53%	36874082	0.14%	N/A	N/A	N/A	N/A	N/A	3	6
SAP_DS	7.41	0.44%	41586358	0.27%	7.21	0.41%	40834947	0.22%	0.20	6.59%	751411	9.69%	N/A	N/A	N/A	N/A	N/A	3	6
SAP_LD	7.72	0.34%	44320799	0.20%	7.29	0.34%	74491516	0.17%	0.43	1.95%	2029373	4.30%	N/A	N/A	N/A	N/A	N/A	3	6
SBF_AR	68.09	0.42%	311735862	0.03%	7.14	0.70%	41221638	0.19%	1.22	1.22%	8876217	0.11%	59.73	0.41%	261638018	0.00%	54	2	2
SBF_BF	69.11	0.38%	310215680	0.02%	7.13	0.66%	41220103	0.19%	N/A	N/A	N/A	N/A	61.98	0.37%	268995577	0.00%	93	2	2
SBF_BS	67.44	0.11%	304637561	0.03%	7.10	0.77%	41220056	0.19%	0.56	0.99%	1779487	0.00%	59.78	0.11%	261638018	0.00%	54	2	2
SBF_DS	69.40	0.32%	303399956	0.04%	7.09	0.65%	41358564	0.19%	0.09	14.82%	403374	17.59%	62.22	0.34%	261638018	0.00%	54	2	2
SBF_LD	69.41	0.38%	303490752	0.07%	7.14	0.69%	41322844	0.16%	0.13	14.39%	529891	41.38%	62.14	0.38%	261638018	0.00%	54	2	2
KR_AR	131.38	0.00%	1454609605	0.00%	3.86	0.00%	22700549	0.00%	127.52	0.00%	1431909056	0.00%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	88.24	0.34%	718383175	0.01%	3.84	0.67%	19300235	0.34%	N/A	N/A	N/A	N/A	84.41	0.36%	699082941	0.01%	105	N/A	N/A
KR_BS	26.43	0.12%	207360478	0.03%	4.78	1.12%	26765896	0.08%	21.65	0.18%	180594583	0.04%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	11.99	2.02%	83939229	2.75%	4.81	1.20%	26787526	0.12%	7.18	3.78%	57151704	4.04%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	4.82	0.31%	29025589	0.48%	3.52	0.55%	17920789	0.25%	1.30	0.94%	11104891	1.22%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

KARZ_DENSE_988K: $m = 987942$, $n = 42280$, $v = 90$

	Running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	0.02	0.00%	131269	1.03%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	0.02	22.82%	131870	1.90%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	0.02	0.00%	131118	0.94%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	0.02	22.82%	130805	1.07%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	0.03	40.82%	309633	32.47%	N/A	N/A	N/A	N/A	0.03	40.82%	309633	32.47%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	0.07	35.24%	43221	37.15%	N/A	N/A	N/A	N/A	0.07	35.24%	43221	37.15%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	0.05	37.27%	298171	31.51%	N/A	N/A	N/A	N/A	0.05	37.27%	298171	31.51%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	0.03	26.31%	339734	34.59%	N/A	N/A	N/A	N/A	0.03	26.31%	339734	34.59%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	0.16	15.93%	1212887	14.33%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.16	15.93%	1212887	14.33%	4	N/A	N/A
GR	0.31	13.86%	1803681	13.43%	0.07	20.20%	462470	14.99%	N/A	N/A	N/A	N/A	0.24	16.11%	1341211	14.30%	3	N/A	N/A
SAP_AR	1.30	10.69%	49234498	9.51%	1.14	11.74%	4120032	10.29%	0.16	10.40%	803466	15.00%	N/A	N/A	N/A	N/A	N/A	4	7
SAP_BF	1.66	11.63%	5809715	11.70%	1.20	9.94%	3993729	10.61%	N/A	N/A	N/A	N/A	0.46	16.70%	1815986	15.29%	16	4	7
SAP_BS	1.32	13.54%	4800567	11.94%	1.15	11.99%	4208762	10.52%	0.18	28.76%	591805	27.05%	N/A	N/A	N/A	N/A	N/A	4	7
SAP_DS	1.23	26.02%	4654259	26.18%	1.08	24.33%	4004003	24.36%	0.14	39.77%	650256	37.88%	N/A	N/A	N/A	N/A	N/A	4	6
SAP_LD	1.30	10.94%	4887963	9.64%	1.16	11.05%	4156505	10.28%	0.14	14.29%	731471	16.71%	N/A	N/A	N/A	N/A	N/A	4	7
SBF_AR	0.24	20.74%	1145971	14.84%	0.02	55.90%	102821	55.90%	0.00	0.00%	0	0.00%	0.22	18.46%	1043151	13.18%	4	0	0
SBF_BF	0.24	20.01%	1145971	14.84%	0.02	60.86%	102821	55.90%	N/A	N/A	N/A	N/A	0.22	18.46%	1043151	13.18%	4	0	0
SBF_BS	0.24	19.57%	1145971	14.84%	0.02	59.27%	102821	55.90%	0.00	0.00%	0	0.00%	0.22	18.69%	1043151	13.18%	4	0	0
SBF_DS	0.25	20.00%	1145971	14.84%	0.02	59.27%	102821	55.90%	0.00	0.00%	0	0.00%	0.23	19.47%	1043151	13.18%	4	0	0
SBF_LD	0.25	21.63%	1145971	14.84%	0.02	61.24%	102821	55.90%	0.00	0.00%	0	0.00%	0.23	18.93%	1043151	13.18%	4	0	0
KR_AR	0.31	4.96%	1591092	8.47%	0.22	3.99%	1133215	0.62%	0.08	13.36%	457878	29.25%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	0.53	14.14%	2822655	16.78%	0.23	3.82%	1016330	0.43%	N/A	N/A	N/A	N/A	0.29	27.86%	1806325	26.25%	17	N/A	N/A
KR_BS	0.43	5.99%	1724821	7.92%	0.29	2.44%	1223649	0.65%	0.14	21.36%	501173	27.35%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	0.48	7.54%	2156142	11.64%	0.30	4.38%	1224048	0.58%	0.18	22.61%	932095	26.53%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	0.31	11.69%	1483525	16.79%	0.20	2.68%	926101	0.50%	0.10	32.07%	557439	44.75%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

RANDOM_64K: $m = 64252$, $n = 7074$, $v = 14$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	0.05	0.00%	337390	0.43%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	0.06	7.21%	337819	0.72%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	0.05	8.60%	338055	0.67%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	0.06	8.56%	337571	0.51%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	0.05	44.72%	451299	56.28%	N/A	N/A	N/A	N/A	0.05	44.72%	451299	56.28%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	0.15	43.46%	670934	64.56%	N/A	N/A	N/A	N/A	0.15	43.46%	670934	64.56%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	0.08	38.21%	475064	42.13%	N/A	N/A	N/A	N/A	0.08	38.21%	475064	42.13%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	0.05	56.57%	472447	63.95%	N/A	N/A	N/A	N/A	0.05	56.57%	472447	63.95%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	0.34	17.92%	2692592	15.89%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.34	17.92%	2692592	15.89%	4	N/A	N/A
GR	0.73	15.29%	4227178	16.63%	0.17	7.58%	1000745	23.55%	N/A	N/A	N/A	N/A	0.55	18.30%	3226433	15.16%	3	N/A	N/A
SAP_AR	3.53	29.88%	12874525	26.31%	3.11	28.66%	11114899	23.72%	0.43	39.34%	1759626	42.90%	N/A	N/A	N/A	N/A	N/A	4	8
SAP_BF	4.38	32.96%	15169116	31.01%	3.26	26.94%	10784850	24.45%	N/A	N/A	N/A	N/A	1.12	51.25%	4384266	47.61%	20	4	8
SAP_BS	3.58	30.06%	12377158	28.64%	3.16	27.99%	11225548	25.69%	0.42	46.82%	1151610	59.13%	N/A	N/A	N/A	N/A	N/A	4	8
SAP_DS	3.55	29.33%	13004757	29.73%	3.18	27.29%	11480106	26.09%	0.37	47.59%	1524651	59.46%	N/A	N/A	N/A	N/A	N/A	4	8
SAP_LD	3.47	28.69%	12718411	25.97%	3.13	27.51%	11179246	23.50%	0.34	41.33%	1539187	44.27%	N/A	N/A	N/A	N/A	N/A	4	8
SBF_AR	0.49	22.56%	2477637	20.15%	0.01	223.6%	66666	223.5%	0.00	0.00%	0	0.00%	0.48	18.05%	2410971	15.97%	4	0	0
SBF_BF	0.48	24.21%	2477637	20.15%	0.01	223.6%	66666	223.5%	N/A	N/A	N/A	N/A	0.46	19.49%	2410971	15.97%	4	0	0
SBF_BS	0.48	23.58%	2477637	20.15%	0.01	223.6%	66666	223.5%	0.00	0.00%	0	0.00%	0.47	18.28%	2410971	15.97%	4	0	0
SBF_DS	0.50	23.04%	2477637	20.15%	0.01	223.6%	66666	223.5%	0.00	0.00%	0	0.00%	0.49	18.55%	2410971	15.97%	4	0	0
SBF_LD	0.49	23.85%	2477637	20.15%	0.01	186.3%	66666	223.5%	0.00	0.00%	0	0.00%	0.48	19.54%	2410971	15.97%	4	0	0
KR_AR	0.83	4.58%	4247248	5.47%	0.57	0.95%	2927100	0.20%	0.25	15.21%	1320149	17.27%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	1.41	12.48%	7643068	13.04%	0.58	0.94%	2609771	0.31%	N/A	N/A	N/A	N/A	0.83	20.78%	5033297	19.92%	19	N/A	N/A
KR_BS	1.11	10.31%	4411947	12.17%	0.80	2.45%	3428643	0.22%	0.31	30.76%	983304	54.80%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	1.19	9.99%	5049044	13.12%	0.78	2.32%	3421491	0.22%	0.41	26.87%	1627554	40.69%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	0.78	7.68%	3636516	11.87%	0.53	1.33%	2387830	0.26%	0.25	22.20%	1248709	34.87%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

RANDOM_167K: $m = 166786$, $n = 14042$, $v = 22$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	0.16	2.83%	921869	1.09%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	0.19	2.94%	919601	0.81%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	0.16	0.00%	918032	0.43%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	0.18	0.00%	919223	0.59%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	0.21	39.27%	1837425	45.82%	N/A	N/A	N/A	N/A	0.21	39.27%	1837425	45.82%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	0.62	42.56%	3143038	58.59%	N/A	N/A	N/A	N/A	0.62	42.56%	3143038	58.59%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	0.35	39.22%	1804427	36.94%	N/A	N/A	N/A	N/A	0.35	39.22%	1804427	36.94%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	0.24	45.00%	2020612	49.94%	N/A	N/A	N/A	N/A	0.24	45.00%	2020612	49.94%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	1.00	11.82%	7539227	10.53%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1.00	11.82%	7539227	10.53%	4	N/A	N/A
GR	2.04	10.52%	11358780	13.55%	0.47	12.01%	2910394	14.04%	N/A	N/A	N/A	N/A	1.58	10.96%	8448386	14.79%	3	N/A	N/A
SAP_AR	11.16	24.95%	39887327	22.41%	9.78	24.15%	34041268	20.59%	1.38	31.32%	5846059	33.96%	N/A	N/A	N/A	N/A	N/A	4	9
SAP_BF	13.63	24.97%	45817624	22.92%	10.27	23.46%	33139365	21.15%	N/A	N/A	N/A	N/A	3.36	31.45%	12678259	29.06%	22	4	9
SAP_BS	10.47	24.61%	35875429	24.06%	9.01	23.53%	31839947	21.86%	1.45	32.13%	4035483	42.68%	N/A	N/A	N/A	N/A	N/A	4	8
SAP_DS	11.70	16.88%	42108976	17.25%	10.46	16.85%	37100325	16.84%	1.25	19.62%	5008650	21.69%	N/A	N/A	N/A	N/A	N/A	4	9
SAP_LD	11.08	24.94%	39149034	21.80%	9.93	24.40%	33800606	20.05%	1.15	30.01%	5348456	33.38%	N/A	N/A	N/A	N/A	N/A	4	9
SBF_AR	1.55	17.06%	7335212	12.31%	0.11	91.36%	545897	91.28%	0.00	0.00%	0	0.00%	1.44	12.52%	6789315	9.93%	4	0	0
SBF_BF	1.50	17.50%	7335212	12.31%	0.11	91.29%	545897	91.28%	N/A	N/A	N/A	N/A	1.39	12.40%	6789315	9.93%	4	0	0
SBF_BS	1.50	17.96%	7335212	12.31%	0.11	91.52%	545897	91.28%	0.00	0.00%	0	0.00%	1.39	13.00%	6789315	9.93%	4	0	0
SBF_DS	1.58	17.70%	7335212	12.31%	0.11	91.29%	545897	91.28%	0.00	0.00%	0	0.00%	1.47	12.89%	6789315	9.93%	4	0	0
SBF_LD	1.58	17.45%	7335212	12.31%	0.11	87.20%	545897	91.28%	0.00	0.00%	0	0.00%	1.47	12.87%	6789315	9.93%	4	0	0
KR_AR	2.36	7.02%	12170923	10.57%	1.54	0.71%	7927127	0.19%	0.82	19.15%	4243797	30.20%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	4.75	7.28%	24247945	6.59%	2.01	0.79%	9491977	0.20%	N/A	N/A	N/A	N/A	2.74	13.03%	14755969	10.77%	38	N/A	N/A
KR_BS	3.34	8.34%	13563758	9.91%	2.15	1.04%	9978543	0.08%	1.19	24.21%	3585216	37.45%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	3.61	9.64%	15933511	13.20%	2.18	1.90%	9990522	0.18%	1.43	25.79%	5942990	35.32%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	2.28	4.90%	10806978	9.07%	1.43	0.80%	6468910	0.23%	0.85	11.89%	4338099	22.33%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

RANDOM_454K: $m = 454720$, $n = 28482$, $v = 29$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	0.46	2.39%	2443317	0.15%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	0.52	1.06%	2446032	0.15%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	0.46	1.18%	2446972	0.32%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	0.51	1.07%	2446746	0.27%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	0.61	40.74%	48331672	45.17%	N/A	N/A	N/A	N/A	0.61	40.74%	48331672	45.17%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	1.75	21.62%	7274378	29.87%	N/A	N/A	N/A	N/A	1.75	21.62%	7274378	29.87%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	0.86	45.63%	4511171	40.26%	N/A	N/A	N/A	N/A	0.86	45.63%	4511171	40.26%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	0.71	45.26%	5383561	49.90%	N/A	N/A	N/A	N/A	0.71	45.26%	5383561	49.90%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	2.92	17.96%	20789172	18.71%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	2.92	17.96%	20789172	18.71%	4	N/A	N/A
GR	5.79	17.23%	30996169	20.21%	1.27	11.40%	7776734	26.14%	N/A	N/A	N/A	N/A	4.52	19.58%	23219435	19.01%	3	N/A	N/A
SAP_AR	37.97	14.98%	131149428	14.27%	33.88	14.57%	11389580	13.56%	4.09	19.25%	17759848	19.99%	N/A	N/A	N/A	N/A	N/A	5	11
SAP_BF	44.76	14.42%	149018396	13.75%	34.90	14.59%	110981210	13.86%	N/A	N/A	N/A	N/A	9.86	15.30%	38037187	15.17%	26	5	11
SAP_BS	39.51	14.64%	132288387	14.52%	34.51	14.14%	117859498	1.87%	5.00	19.12%	14428889	21.33%	N/A	N/A	N/A	N/A	N/A	5	11
SAP_DS	39.38	14.76%	13842629	14.93%	35.06	14.37%	121200586	14.04%	4.31	18.87%	17142043	22.97%	N/A	N/A	N/A	N/A	N/A	5	11
SAP_LD	35.54	15.90%	120075965	14.87%	32.30	15.43%	105061145	14.04%	3.24	21.22%	150114859	21.88%	N/A	N/A	N/A	N/A	N/A	5	10
SBF_AR	4.33	23.61%	20207993	21.49%	0.31	91.33%	1458110	91.28%	0.00	0.00%	0	0.00%	4.02	19.00%	18749883	17.89%	4	0	0
SBF_BF	4.32	23.56%	20207993	21.49%	0.32	91.32%	1458110	91.28%	N/A	N/A	N/A	N/A	4.00	18.84%	18749883	17.89%	4	0	0
SBF_BS	4.32	23.55%	20207993	21.49%	0.32	91.29%	1458110	91.28%	0.00	0.00%	0	0.00%	4.00	18.86%	18749883	17.89%	4	0	0
SBF_DS	4.50	23.27%	20207993	21.49%	0.30	91.30%	1458110	91.28%	0.00	0.00%	0	0.00%	4.20	19.03%	18749883	17.89%	4	0	0
SBF_LD	4.51	24.18%	20207993	21.49%	0.31	91.33%	1458110	91.28%	0.00	0.00%	0	0.00%	4.20	19.77%	18749883	17.89%	4	0	0
KR_AR	8.03	0.05%	41027896	0.05%	5.10	0.01%	29072989	0.00%	2.93	0.16%	11954908	0.16%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	13.48	5.26%	69382084	5.19%	5.16	0.61%	25221045	0.04%	N/A	N/A	N/A	N/A	8.32	8.38%	44161040	8.15%	41	N/A	N/A
KR_BS	11.20	9.72%	42213158	8.97%	6.68	0.48%	29775766	0.02%	4.52	24.50%	12437393	30.46%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	11.62	10.36%	48722996	13.25%	6.78	0.50%	29773900	0.04%	4.84	25.23%	18949096	34.05%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	7.62	7.38%	35139030	9.55%	4.69	0.39%	23042961	0.05%	2.93	18.84%	12096110	27.79%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

RANDOM_1214K: $m = 1215125$, $n = 56432$, $v = 39$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	0.01	55.90%	50263	0.74%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	0.01	0.00%	50368	1.15%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	0.01	55.90%	50348	0.86%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	0.01	0.00%	50389	0.96%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	0.01	37.27%	105506	48.31%	N/A	N/A	N/A	N/A	0.01	37.27%	105506	48.31%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	0.02	38.03%	96594	50.20%	N/A	N/A	N/A	N/A	0.02	38.03%	96594	50.20%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	0.02	71.26%	103904	51.53%	N/A	N/A	N/A	N/A	0.02	71.26%	103904	51.53%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	0.01	104.6%	110899	50.11%	N/A	N/A	N/A	N/A	0.01	104.6%	110899	50.11%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	0.06	27.08%	443182	19.95%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.06	27.08%	443182	19.95%	4	N/A	N/A
GR	0.12	22.82%	668623	18.12%	0.03	43.85%	166644	16.02%	N/A	N/A	N/A	N/A	0.09	24.49%	501979	20.11%	3	N/A	N/A
SAP_AR	0.52	25.70%	1872914	25.24%	0.46	23.91%	1565226	22.08%	0.06	40.75%	307688	45.18%	N/A	N/A	N/A	N/A	N/A	3	6
SAP_BF	0.65	28.41%	2208066	27.69%	0.48	23.84%	1520911	23.03%	N/A	N/A	N/A	N/A	0.17	41.56%	687155	38.96%	12	3	6
SAP_BS	0.48	11.50%	1648770	11.13%	0.42	11.54%	1458746	10.26%	0.06	14.43%	190024	27.49%	N/A	N/A	N/A	N/A	N/A	3	6
SAP_DS	0.54	26.14%	1965421	26.47%	0.47	24.38%	1623500	24.62%	0.07	41.21%	341921	37.22%	N/A	N/A	N/A	N/A	N/A	3	6
SAP_LD	0.51	23.21%	1867305	23.18%	0.46	21.47%	1601986	22.09%	0.05	38.40%	265329	33.69%	N/A	N/A	N/A	N/A	N/A	3	6
SBF_AR	0.09	27.22%	408246	19.61%	0.01	104.6%	29414	91.26%	0.00	0.00%	0	0.00%	0.08	27.81%	378832	17.61%	4	0	0
SBF_BF	0.09	26.06%	408246	19.61%	0.01	91.29%	29414	91.26%	N/A	N/A	N/A	N/A	0.08	24.69%	378832	17.61%	4	0	0
SBF_BS	0.08	29.88%	408246	19.61%	0.01	91.29%	29414	91.26%	0.00	0.00%	0	0.00%	0.08	29.24%	378832	17.61%	4	0	0
SBF_DS	0.09	29.17%	408246	19.61%	0.01	91.29%	29414	91.26%	0.00	0.00%	0	0.00%	0.09	26.77%	378832	17.61%	4	0	0
SBF_LD	0.09	24.64%	408246	19.61%	0.01	91.29%	29414	91.26%	0.00	0.00%	0	0.00%	0.08	21.82%	378832	17.61%	4	0	0
KR_AR	0.09	9.52%	476969	14.47%	0.06	0.00%	277507	0.75%	0.03	26.31%	199463	34.83%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	0.16	14.74%	876389	17.77%	0.07	8.30%	260000	0.57%	N/A	N/A	N/A	N/A	0.10	27.16%	616390	25.34%	9	N/A	N/A
KR_BS	0.14	13.55%	528275	16.70%	0.09	12.13%	363560	0.66%	0.05	22.82%	164716	54.24%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	0.15	11.80%	649397	15.03%	0.10	11.18%	362026	0.45%	0.06	41.11%	287373	33.83%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	0.09	11.11%	423304	16.36%	0.06	7.21%	238475	0.36%	0.03	29.88%	184739	37.35%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

RANDOM_DENSE_24K: $m = 24510$, $n = 5282$, $v = 9$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	0.03	15.97%	160752	0.58%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	0.03	13.98%	160772	0.57%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	0.03	15.97%	160369	0.32%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	0.03	13.98%	160389	0.29%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	0.03	51.35%	271312	56.20%	N/A	N/A	N/A	N/A	0.03	51.35%	271312	56.20%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	0.08	35.55%	376659	48.98%	N/A	N/A	N/A	N/A	0.08	35.55%	376659	48.98%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	0.05	53.93%	266190	53.70%	N/A	N/A	N/A	N/A	0.05	53.93%	266190	53.70%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	0.04	50.46%	290315	61.27%	N/A	N/A	N/A	N/A	0.04	50.46%	290315	61.27%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	0.18	6.80%	1325172	8.12%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.18	6.80%	1325172	8.12%	4	N/A	N/A
GR	0.37	13.74%	1991309	12.95%	0.08	12.50%	475932	36.00%	N/A	N/A	N/A	N/A	0.29	14.75%	1515377	17.06%	3	N/A	N/A
SAP_AR	1.74	24.32%	6188107	22.21%	1.51	24.32%	5283124	21.51%	0.23	25.46%	904983	29.59%	N/A	N/A	N/A	N/A	N/A	4	7
SAP_BF	2.10	24.30%	7183352	23.94%	1.58	23.86%	5127520	22.17%	N/A	N/A	N/A	N/A	0.52	28.00%	2055832	30.22%	14	4	7
SAP_BS	1.73	21.24%	5911805	21.77%	1.53	22.02%	5344072	22.27%	0.20	20.31%	567733	24.94%	N/A	N/A	N/A	N/A	N/A	4	7
SAP_DS	1.66	25.12%	5906956	24.64%	1.49	24.42%	5241239	23.46%	0.17	43.62%	665717	48.75%	N/A	N/A	N/A	N/A	N/A	4	6
SAP_LD	1.73	23.44%	6150373	22.36%	1.55	23.28%	5332858	21.36%	0.18	28.01%	817527	31.80%	N/A	N/A	N/A	N/A	N/A	4	7
SBF_AR	0.26	10.59%	1249163	4.89%	0.01	136.9%	63542	136.9%	0.00	0.00%	0	0.00%	0.25	7.48%	1185621	7.89%	4	0	0
SBF_BF	0.26	10.59%	1249163	4.89%	0.01	136.9%	63542	136.9%	N/A	N/A	N/A	N/A	0.25	7.48%	1185621	7.89%	4	0	0
SBF_BS	0.26	10.24%	1249163	4.89%	0.01	108.3%	63542	136.9%	0.00	0.00%	0	0.00%	0.25	8.74%	1185621	7.89%	4	0	0
SBF_DS	0.27	9.52%	1249163	4.89%	0.01	139.2%	63452	136.9%	0.00	0.00%	0	0.00%	0.26	7.20%	1185621	7.89%	4	0	0
SBF_LD	0.27	9.86%	1249163	4.89%	0.02	138.3%	63542	136.9%	0.00	0.00%	0	0.00%	0.26	8.99%	1185621	7.89%	4	0	0
KR_AR	0.28	5.91%	1270368	6.58%	0.20	2.68%	889106	0.27%	0.07	15.41%	381263	22.40%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	0.66	15.34%	3284351	18.04%	0.30	1.85%	1252703	0.59%	N/A	N/A	N/A	N/A	0.36	26.86%	2031648	29.26%	17	N/A	N/A
KR_BS	0.46	9.93%	1797123	11.35%	0.33	2.74%	1365914	0.71%	0.13	38.03%	431210	46.56%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	0.50	10.04%	2141081	14.03%	0.33	2.74%	1366211	0.22%	0.17	31.40%	774871	38.86%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	0.29	7.25%	1250348	16.16%	0.20	0.00%	764461	0.35%	0.09	24.11%	485900	41.60%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

RANDOM_DENSE_79K: $m = 79321$, $n = 10504$, $v = 12$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	0.09	0.00%	552774	0.46%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	0.11	7.47%	552610	0.51%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	0.09	4.86%	553425	0.72%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	0.11	3.99%	551899	0.24%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	0.13	37.06%	1113647	46.22%	N/A	N/A	N/A	N/A	0.13	37.06%	1113647	46.22%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	0.28	54.75%	1320051	67.92%	N/A	N/A	N/A	N/A	0.28	54.75%	1320051	67.92%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	0.18	60.84%	1012696	55.07%	N/A	N/A	N/A	N/A	0.18	60.84%	1012696	55.07%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	0.14	45.74%	1217917	50.02%	N/A	N/A	N/A	N/A	0.14	45.74%	1217917	50.02%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	0.57	20.51%	4279292	18.21%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.57	20.51%	4279292	18.21%	3	N/A	N/A
GR	1.12	17.41%	6361446	19.40%	0.25	12.04%	1643496	23.56%	N/A	N/A	N/A	N/A	0.87	20.02%	4717950	18.61%	2	N/A	N/A
SAP_AR	6.57	25.42%	23204905	22.72%	5.75	24.93%	19903438	21.00%	0.82	29.53%	3301466	34.39%	N/A	N/A	N/A	N/A	N/A	4	9
SAP_BF	7.89	29.00%	26513641	26.64%	6.00	24.63%	19360558	21.59%	N/A	N/A	N/A	N/A	1.88	43.16%	7153083	40.63%	20	4	9
SAP_BS	6.67	27.94%	22829999	26.05%	5.79	24.93%	20302953	22.94%	0.88	48.63%	2527046	51.95%	N/A	N/A	N/A	N/A	N/A	4	9
SAP_DS	6.63	28.40%	23948222	26.98%	5.89	25.94%	20802370	23.45%	0.74	50.59%	3145851	52.88%	N/A	N/A	N/A	N/A	N/A	4	9
SAP_LD	6.45	25.06%	22867470	22.01%	5.78	24.91%	19902045	20.71%	0.67	26.75%	2965447	32.12%	N/A	N/A	N/A	N/A	N/A	9	4
SBF_AR	0.86	25.70%	4133236	20.85%	0.06	91.51%	328480	91.28%	0.00	0.00%	0	0.00%	0.80	21.05%	3804756	16.81%	3	0	0
SBF_BF	0.86	26.04%	4133236	20.85%	0.07	91.29%	328480	91.28%	N/A	N/A	N/A	N/A	0.79	21.25%	3804756	16.81%	3	0	0
SBF_BS	0.87	26.03%	4133236	20.85%	0.07	91.29%	328480	91.28%	0.00	0.00%	0	0.00%	0.81	21.36%	3804756	16.81%	3	0	0
SBF_DS	0.90	25.87%	4133236	20.85%	0.06	91.51%	328480	91.28%	0.00	0.00%	0	0.00%	0.83	21.36%	3804756	16.81%	3	0	0
SBF_LD	0.89	26.35%	4133236	20.85%	0.06	91.52%	328480	91.28%	0.00	0.00%	0	0.00%	0.83	21.91%	3804756	16.81%	3	0	0
KR_AR	1.27	3.17%	6337700	1.98%	0.93	0.59%	4794960	0.12%	0.34	11.76%	1542741	8.33%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	2.76	15.76%	14098342	16.11%	1.23	1.41%	5685542	0.24%	N/A	N/A	N/A	N/A	1.53	28.18%	8412801	27.05%	34	N/A	N/A
KR_BS	1.96	16.22%	7685629	16.37%	1.31	1.49%	5741194	0.18%	0.65	48.17%	1944436	65.10%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	2.10	5.41%	9017091	20.42%	1.35	0.91%	5739858	0.15%	0.75	42.69%	3277234	56.35%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	1.31	8.55%	5987326	13.76%	0.88	1.48%	3920686	0.24%	0.43	26.05%	2066663	40.15%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

RANDOM_DENSE_274K: $m = 273939$, $n = 21314$, $v = 22$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	0.35	1.55%	1990035	0.27%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	0.40	1.77%	1989936	0.20%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	0.35	1.55%	1994305	0.50%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	0.40	1.12%	1990269	0.22%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	0.47	41.58%	3871157	44.42%	N/A	N/A	N/A	N/A	0.47	41.58%	3871157	44.42%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	1.42	27.63%	6424168	39.54%	N/A	N/A	N/A	N/A	1.42	27.63%	6424168	39.54%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	0.69	53.67%	3728306	43.85%	N/A	N/A	N/A	N/A	0.69	53.67%	3728306	43.85%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	0.54	44.36%	4376153	49.82%	N/A	N/A	N/A	N/A	0.54	44.36%	4376153	49.82%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	2.38	11.02%	17565195	9.61%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	2.38	11.02%	17565195	9.61%	42	N/A	N/A
GR	4.71	8.93%	26655812	10.90%	1.00	9.49%	6718738	16.10%	N/A	N/A	N/A	N/A	3.72	9.66%	19937074	10.53%	3	N/A	N/A
SAP_AR	25.86	23.84%	93423730	20.94%	23.27	22.04%	81965835	17.84%	2.60	41.03%	11457895	44.72%	N/A	N/A	N/A	N/A	N/A	5	10
SAP_BF	32.81	13.00%	112405673	13.28%	25.24	12.31%	828652025	11.59%	N/A	N/A	N/A	N/A	7.57	17.54%	29543648	18.86%	28	5	11
SAP_BS	26.60	21.90%	92108939	21.19%	23.02	20.03%	81661801	18.61%	3.58	35.77%	10447138	43.44%	N/A	N/A	N/A	N/A	N/A	5	10
SAP_DS	28.43	12.03%	103447842	11.86%	25.14	12.76%	89941242	12.61%	3.29	18.28%	13506600	23.78%	N/A	N/A	N/A	N/A	N/A	5	11
SAP_LD	24.62	16.35%	87290050	13.87%	22.45	16.24%	77170228	13.02%	2.17	18.90%	10119865	23.03%	N/A	N/A	N/A	N/A	N/A	5	10
SBF_AR	3.54	18.35%	17496539	13.43%	0.24	91.30%	1185367	91.28%	0.00	0.00%	0	0.00%	3.29	13.23%	16311171	9.95%	4	0	0
SBF_BF	3.53	18.42%	17496539	13.43%	0.24	91.33%	1185367	91.28%	N/A	N/A	N/A	N/A	3.29	13.28%	16311171	9.95%	4	0	0
SBF_BS	3.54	17.88%	17496539	13.43%	0.24	91.40%	1185367	91.28%	0.00	0.00%	0	0.00%	3.30	12.85%	16311171	9.95%	4	0	0
SBF_DS	3.68	17.54%	17496539	13.43%	0.23	91.30%	1185367	91.28%	0.00	0.00%	0	0.00%	3.45	12.80%	16311171	9.95%	4	0	0
SBF_LD	3.68	17.71%	17496539	13.43%	0.23	91.34%	1185367	91.28%	0.00	0.00%	0	0.00%	3.45	12.91%	16311171	9.95%	4	0	0
KR_AR	6.30	0.07%	33077018	0.08%	4.09	0.00%	23601197	0.00%	2.21	0.21%	9475822	0.28%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	10.77	12.21%	57890203	11.45%	4.14	0.20%	20454304	0.09%	N/A	N/A	N/A	N/A	6.63	19.84%	37435900	17.75%	41	N/A	N/A
KR_BS	8.48	7.17%	33757517	8.42%	5.33	0.90%	24659805	0.08%	3.16	20.62%	9097713	31.43%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	9.19	10.49%	39465580	13.06%	5.46	1.16%	24668243	0.07%	3.72	26.15%	14797338	34.88%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	6.15	6.11%	29258848	9.13%	3.75	0.30%	18693846	0.09%	2.39	15.46%	10565045	25.27%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

RANDOM_DENSE_988K: $m = 988062$, $n = 42280$, $v = 42$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	0.01	37.27%	118667	0.00%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	0.01	0.00%	100837	0.00%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	0.01	0.00%	104293	0.00%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	0.01	0.00%	82766	0.00%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	0.01	39.12%	257230	0.00%	N/A	N/A	N/A	N/A	0.01	39.12%	257230	0.00%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	0.07	6.21%	832601	0.00%	N/A	N/A	N/A	N/A	0.07	6.21%	832601	0.00%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	0.06	0.00%	303006	0.00%	N/A	N/A	N/A	N/A	0.06	0.00%	303006	0.00%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	0.01	0.00%	177533	0.00%	N/A	N/A	N/A	N/A	0.01	0.00%	177533	0.00%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	0.06	9.78%	518181	0.00%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.06	9.78%	518181	0.00%	3	N/A	N/A
GR	0.10	4.56%	674613	0.00%	0.02	0.00%	156333	0.00%	N/A	N/A	N/A	N/A	0.08	5.73%	518280	0.00%	2	N/A	N/A
SAP_AR	0.26	2.72%	1530437	0.00%	0.24	4.83%	1358195	0.00%	0.02	22.82%	172242	0.00%	N/A	N/A	N/A	N/A	N/A	4	6
SAP_BF	0.31	5.73%	1626386	0.00%	0.25	8.60%	1305135	0.00%	N/A	N/A	N/A	N/A	0.06	11.79%	321251	0.00%	13	4	6
SAP_BS	0.28	2.97%	1535921	0.00%	0.25	4.90%	1325042	0.00%	0.03	26.15%	210879	0.00%	N/A	N/A	N/A	N/A	N/A	4	6
SAP_DS	0.26	4.32%	1584798	0.00%	0.23	5.62%	1368844	0.00%	0.03	26.15%	215954	0.00%	N/A	N/A	N/A	N/A	N/A	4	6
SAP_LD	0.26	1.73%	1477085	0.00%	0.25	2.23%	1377037	0.00%	0.01	37.27%	100063	0.00%	N/A	N/A	N/A	N/A	N/A	4	6
SBF_AR	0.10	22.36%	418733	0.00%	0.01	0.00%	52045	0.00%	0.00	0.00%	0	0.00%	0.09	24.85%	366688	0.00%	3	0	0
SBF_BF	0.09	5.83%	418733	0.00%	0.01	0.00%	52045	0.00%	N/A	N/A	N/A	N/A	0.08	6.52%	366688	0.00%	3	0	0
SBF_BS	0.09	0.00%	418733	0.00%	0.01	0.00%	52045	0.00%	0.00	0.00%	0	0.00%	0.08	0.00%	366688	0.00%	3	0	0
SBF_DS	0.09	5.83%	418733	0.00%	0.01	55.90%	52045	0.00%	0.00	0.00%	0	0.00%	0.09	6.37%	366688	0.00%	3	0	0
SBF_LD	0.09	4.86%	418733	0.00%	0.01	55.90%	52045	0.00%	0.00	0.00%	0	0.00%	0.08	6.52%	366688	0.00%	3	0	0
KR_AR	0.14	5.89%	1127286	11.30%	0.10	5.71%	613798	0.20%	0.05	19.44%	513489	24.67%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	0.24	9.32%	1692237	9.37%	0.09	5.83%	529995	0.28%	N/A	N/A	N/A	N/A	0.15	13.35%	1162243	13.72%	20	N/A	N/A
KR_BS	0.23	1.93%	1515169	1.63%	0.11	4.80%	620186	0.12%	0.12	7.09%	894983	2.82%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	0.21	3.37%	1285256	1.50%	0.13	8.56%	620112	0.12%	0.08	10.20%	665145	2.81%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	0.13	7.69%	952808	7.74%	0.09	11.11%	484911	0.70%	0.04	30.62%	467914	16.28%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

SHADED_DENSITY_26K: $m = 26072$, $n = 1602$, $v = 15$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	0.04	15.21%	342583	0.71%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	0.03	3.98%	271874	2.18%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	0.03	0.00%	306651	0.93%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	0.03	15.97%	237275	0.23%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	0.03	21.07%	380860	0.14%	N/A	N/A	N/A	N/A	0.03	21.07%	380860	0.14%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	0.50	3.93%	5383602	3.21%	N/A	N/A	N/A	N/A	0.50	3.93%	5383602	3.21%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	0.18	9.23%	1443031	13.04%	N/A	N/A	N/A	N/A	0.18	9.23%	1443031	13.04%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	0.05	22.82%	734312	32.67%	N/A	N/A	N/A	N/A	0.05	22.82%	734312	32.67%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	0.14	26.24%	1226813	28.96%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.14	26.24%	1226813	28.96%	2	N/A	N/A
GR	0.23	34.87%	1519154	35.11%	0.05	19.44%	389544	22.92%	N/A	N/A	N/A	N/A	0.18	39.12%	1129610	39.32%	1	N/A	N/A
SAP_AR	0.95	3.36%	5321388	1.64%	0.91	3.23%	4979356	0.28%	0.04	26.08%	342032	21.44%	N/A	N/A	N/A	N/A	N/A	5	8
SAP_BF	1.24	1.32%	6139649	0.80%	0.95	2.28%	4820243	0.28%	N/A	N/A	N/A	N/A	0.29	2.44%	1319406	2.71%	21	5	8
SAP_BS	1.09	1.76%	5849480	0.62%	0.92	2.10%	4854866	0.26%	0.17	10.44%	994614	2.38%	N/A	N/A	N/A	N/A	N/A	5	8
SAP_DS	1.05	1.74%	6044620	0.42%	0.91	1.44%	5057812	0.25%	0.14	6.06%	986808	1.31%	N/A	N/A	N/A	N/A	N/A	5	8
SAP_LD	0.92	2.26%	5187704	0.17%	0.89	2.64%	5033215	0.26%	0.03	34.40%	154521	2.67%	N/A	N/A	N/A	N/A	N/A	5	8
SBF_AR	0.23	22.87%	1058585	21.14%	0.03	15.97%	162202	0.08%	0.00	0.00%	0	0.00%	0.20	25.25%	896384	24.95%	2	0	0
SBF_BF	0.23	23.81%	1058585	21.14%	0.03	0.00%	162202	0.08%	N/A	N/A	N/A	N/A	0.20	27.39%	896384	24.95%	2	0	0
SBF_BS	0.23	21.81%	1058585	21.14%	0.03	21.07%	162202	0.08%	0.00	0.00%	0	0.00%	0.20	27.39%	896384	24.95%	2	0	0
SBF_DS	0.23	22.89%	1058585	21.14%	0.03	0.00%	162202	0.08%	0.00	0.00%	0	0.00%	0.20	26.29%	896384	24.95%	2	0	0
SBF_LD	0.23	22.87%	1058585	21.14%	0.02	22.82%	162202	0.08%	0.00	0.00%	0	0.00%	0.20	25.14%	896384	24.95%	2	0	0
KR_AR	0.47	11.83%	3777856	14.34%	0.29	3.80%	1912949	0.13%	0.18	30.16%	1864909	29.01%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	0.94	8.44%	6290687	10.62%	0.35	3.70%	2068312	0.08%	N/A	N/A	N/A	N/A	0.59	15.05%	4222377	15.83%	41	N/A	N/A
KR_BS	0.99	3.25%	6898119	3.86%	0.37	3.05%	2142981	0.09%	0.61	5.05%	4755139	5.62%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	0.72	0.76%	4618164	0.68%	0.38	3.88%	2143568	0.23%	0.34	3.81%	2474597	1.36%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	0.42	5.83%	3158795	7.52%	0.26	3.85%	1517258	0.25%	0.16	18.22%	1641571	14.52%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

SHADED_DENSITY_81K: $m = 81172$, $n = 2852$, $v = 32$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	0.13	4.35%	1147650	0.63%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	0.11	4.80%	915490	1.28%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	0.11	4.80%	1046015	0.87%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	0.10	0.00%	820949	2.15%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	0.12	42.08%	1872711	40.79%	N/A	N/A	N/A	N/A	0.12	42.08%	1872711	40.79%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	3.06	8.86%	34910129	8.83%	N/A	N/A	N/A	N/A	3.06	8.86%	34910129	8.83%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	0.57	11.02%	3883172	12.52%	N/A	N/A	N/A	N/A	0.57	11.02%	3883172	12.52%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	0.15	28.49%	2216115	31.82%	N/A	N/A	N/A	N/A	0.15	28.49%	2216115	31.82%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	0.59	18.17%	5091808	19.47%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	0.59	18.17%	5091808	19.47%	3	N/A	N/A
GR	0.99	22.60%	6537987	22.77%	0.19	12.34%	1557233	15.96%	N/A	N/A	N/A	N/A	0.80	25.17%	4980754	24.90%	2	N/A	N/A
SAP_AR	2.86	1.86%	16741568	0.88%	2.72	1.75%	15688995	0.58%	0.14	15.97%	1052573	14.26%	N/A	N/A	N/A	N/A	N/A	5	9
SAP_BF	3.86	2.19%	19151016	0.92%	2.92	1.61%	15160972	0.61%	N/A	N/A	N/A	N/A	0.95	6.46%	3990044	5.33%	31	5	9
SAP_BS	3.39	0.81%	18665094	1.54%	2.75	1.19%	15271699	0.61%	0.64	7.26%	3393395	7.66%	N/A	N/A	N/A	N/A	N/A	5	9
SAP_DS	3.19	1.27%	18709151	0.96%	2.73	0.95%	15869798	0.69%	0.45	3.64%	2839353	3.30%	N/A	N/A	N/A	N/A	N/A	5	9
SAP_LD	2.78	4.06%	16448912	1.80%	2.68	4.14%	15759574	1.82%	0.10	4.38%	689400	9.99%	N/A	N/A	N/A	N/A	N/A	5	9
SBF_AR	0.92	16.51%	4179221	14.89%	0.09	0.00%	556014	0.08%	0.00	0.00%	0	0.00%	0.83	18.30%	3623207	17.17%	3	0	0
SBF_BF	0.93	16.15%	4179221	14.89%	0.09	0.00%	556014	0.08%	N/A	N/A	N/A	N/A	0.84	17.89%	3623207	17.17%	3	0	0
SBF_BS	0.92	16.26%	4179221	14.89%	0.09	0.00%	556014	0.08%	0.00	0.00%	0	0.00%	0.83	18.03%	3623207	17.17%	3	0	0
SBF_DS	0.94	15.98%	4179221	14.89%	0.09	5.08%	556014	0.08%	0.00	0.00%	0	0.00%	0.85	17.11%	3623207	17.17%	3	0	0
SBF_LD	0.93	15.72%	4179221	14.89%	0.08	6.52%	556014	0.08%	0.00	0.00%	0	0.00%	0.85	17.68%	3623207	17.17%	3	0	0
KR_AR	1.84	6.96%	14987205	8.97%	1.17	2.18%	8343723	0.21%	0.67	16.41%	6643483	20.05%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	4.04	6.04%	25718337	6.79%	1.43	1.31%	8468718	0.13%	N/A	N/A	N/A	N/A	2.61	9.39%	17249620	10.12%	76	N/A	N/A
KR_BS	4.96	5.90%	36738328	6.84%	1.47	1.95%	8556757	0.12%	3.48	9.09%	28181571	8.93%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	3.02	2.09%	18274675	1.63%	1.49	1.74%	8557709	0.13%	1.53	5.10%	9716967	2.96%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	1.65	2.42%	12432342	3.11%	1.06	0.67%	6470137	0.30%	0.59	2.15%	6295467	1.75%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

SHADED_DENSITY_278K: $m = 278130$, $n = 5302$, $v = 62$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	0.47	0.95%	4099433	0.27%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	0.42	0.00%	3288263	0.77%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	0.41	2.03%	3799150	0.93%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	0.38	1.46%	3013932	1.59%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	0.44	0.31%	6819754	41.23%	N/A	N/A	N/A	N/A	0.44	0.31%	6819754	41.23%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	22.53	2.95%	263837456	2.90%	N/A	N/A	N/A	N/A	22.53	2.95%	263837456	2.90%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	2.72	10.58%	16930909	12.43%	N/A	N/A	N/A	N/A	2.72	10.58%	16930909	12.43%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	0.52	31.86%	7996279	31.58%	N/A	N/A	N/A	N/A	0.52	31.86%	7996279	31.58%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	1.85	27.26%	15043498	29.00%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	1.85	27.26%	15043498	29.00%	2	N/A	N/A
GR	2.86	34.70%	18627370	35.14%	0.61	18.73%	4777217	22.88%	N/A	N/A	N/A	N/A	2.25	39.04%	13850156	39.36%	1	N/A	N/A
SAP_AR	11.80	1.42%	66758127	0.70%	11.33	1.48%	63405558	0.64%	0.47	3.59%	3352570	3.59%	N/A	N/A	N/A	N/A	N/A	6	12
SAP_BF	15.50	0.61%	76069705	0.49%	12.13	0.85%	61570802	0.68%	N/A	N/A	N/A	N/A	3.37	3.96%	14498903	3.79%	46	6	12
SAP_BS	14.55	0.79%	79561994	0.51%	11.38	0.32%	62272151	0.33%	3.17	3.59%	17289842	3.38%	N/A	N/A	N/A	N/A	N/A	6	12
SAP_DS	13.51	0.87%	77245325	0.55%	11.48	1.27%	65107034	0.94%	2.03	2.75%	12138291	2.01%	N/A	N/A	N/A	N/A	N/A	6	12
SAP_LD	11.50	1.07%	65353688	0.62%	11.20	1.10%	63475832	0.65%	0.31	5.81%	1877988	5.19%	N/A	N/A	N/A	N/A	N/A	6	11
SBF_AR	2.89	22.90%	12953380	21.13%	0.32	1.73%	1990041	0.07%	0.00	0.00%	0	0.00%	2.58	25.92%	10963340	24.95%	2	0	0
SBF_BF	2.93	23.14%	12953380	21.13%	0.31	1.74%	1990041	0.07%	N/A	N/A	N/A	N/A	2.62	25.88%	10963340	24.95%	2	0	0
SBF_BS	2.90	22.98%	12953380	21.13%	0.32	1.41%	1990041	0.07%	0.00	0.00%	0	0.00%	2.58	25.92%	10963340	24.95%	2	0	0
SBF_DS	2.93	23.21%	12953380	21.13%	0.30	1.80%	1990041	0.07%	0.00	0.00%	0	0.00%	2.63	26.04%	10963340	24.95%	2	0	0
SBF_LD	2.94	23.29%	12953380	21.13%	0.30	1.80%	1990041	0.07%	0.00	0.00%	0	0.00%	2.63	26.12%	10963340	24.95%	2	0	0
KR_AR	7.84	0.04%	64638138	0.05%	4.98	0.00%	36265528	0.00%	2.86	0.11%	28372610	0.11%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	15.88	3.66%	103257873	4.30%	5.82	0.84%	35349876	0.09%	N/A	N/A	N/A	N/A	10.05	5.85%	67907998	6.53%	143	N/A	N/A
KR_BS	27.29	4.68%	212776912	5.36%	5.91	1.05%	35018694	0.10%	21.38	6.18%	177758219	6.42%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	13.04	0.97%	75651441	1.13%	5.97	0.90%	35013309	0.13%	7.07	1.39%	40638133	2.12%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	7.12	1.94%	54962451	2.27%	4.42	0.42%	27792810	0.12%	2.70	5.12%	27169774	4.50%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

SHADED_DENSITY_995K: $m = 995237$, $n = 10052$, $v = 132$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	0.23	12.30%	1154755	15.21%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	0.61	13.99%	3342889	15.60%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	0.23	16.53%	1188527	18.27%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	0.23	8.70%	988855	11.40%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	2.13	2.61%	18434183	2.90%	N/A	N/A	N/A	N/A	2.13	2.61%	18434183	2.90%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	1.23	2.54%	6208263	2.37%	N/A	N/A	N/A	N/A	1.23	2.54%	6208263	2.37%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	1.37	3.30%	4757350	2.89%	N/A	N/A	N/A	N/A	1.37	3.30%	4757350	2.89%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	0.48	12.53%	3319523	12.13%	N/A	N/A	N/A	N/A	0.48	12.53%	3319523	12.13%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	2.84	2.65%	17257753	2.64%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	2.84	2.65%	17257753	2.64%	49	N/A	N/A
GR	6.45	2.56%	29470613	2.50%	0.95	3.63%	6205193	2.36%	N/A	N/A	N/A	N/A	5.50	2.53%	23265420	2.55%	48	N/A	N/A
SAP_AR	4.25	3.81%	21906425	3.13%	1.05	11.05%	3625100	10.38%	3.20	3.14%	18281326	2.85%	N/A	N/A	N/A	N/A	N/A	2	4
SAP_BF	5.38	2.57%	19001932	3.12%	1.10	9.18%	3514043	10.69%	N/A	N/A	N/A	N/A	4.29	2.66%	15487890	2.54%	52	2	4
SAP_BS	2.71	5.37%	9777788	4.51%	1.04	11.88%	3618912	10.32%	1.67	2.71%	6158875	2.68%	N/A	N/A	N/A	N/A	N/A	2	4
SAP_DS	2.85	5.21%	8317708	5.95%	1.06	10.34%	3636739	10.81%	1.79	3.10%	4680969	3.56%	N/A	N/A	N/A	N/A	N/A	2	4
SAP_LD	1.78	11.21%	7094194	12.22%	1.03	11.39%	3735862	10.44%	0.75	18.92%	3358380	20.61%	N/A	N/A	N/A	N/A	N/A	2	4
SBF_AR	4.67	1.18%	18380952	2.02%	0.74	2.61%	2857258	1.18%	0.67	8.53%	3779164	9.26%	3.26	0.70%	11744543	0.00%	39	2	2
SBF_BF	4.88	1.96%	17900794	2.08%	0.72	2.54%	2857109	1.17%	N/A	N/A	N/A	N/A	4.16	2.56%	15043685	2.40%	51	2	2
SBF_BS	4.32	1.54%	15942461	1.22%	0.72	3.72%	2857109	1.17%	0.36	11.17%	1340809	11.86%	3.24	0.99%	11744543	0.16%	39	2	2
SBF_DS	4.43	1.29%	15973911	1.33%	0.71	3.22%	2889080	1.15%	0.45	11.61%	1340288	13.13%	3.27	0.83%	11744543	0.16%	39	2	2
SBF_LD	4.38	2.35%	16368406	2.77%	0.72	2.41%	2868069	1.14%	0.38	23.08%	1755795	24.41%	3.28	0.27%	11744543	0.16%	39	2	2
KR_AR	2.72	2.84%	18666581	2.86%	0.09	5.83%	232398	0.00%	2.62	3.09%	18434184	2.90%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	3.12	3.02%	17490151	2.61%	0.10	4.38%	232398	0.00%	N/A	N/A	N/A	N/A	3.01	3.04%	17257754	2.64%	49	N/A	N/A
KR_BS	1.50	3.03%	5965885	4.45%	0.22	2.54%	681568	0.16%	1.29	3.36%	5284318	5.00%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	1.65	2.83%	5032415	335.00%	0.22	2.54%	680779	0.06%	1.43	3.10%	4351637	3.86%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	0.68	0.24%	3551921	11.34%	0.09	4.86%	232398	0.00%	0.59	11.50%	3319572	12.12%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

LONG_GRID_65K: $m = 64450$, $n = 25802$, $v = 48$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	0.60	11.49%	2885149	13.52%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	2.24	15.81%	12113630	16.81%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	0.61	13.02%	2939530	14.61%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	0.61	11.88%	2502373	12.14%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	5.77	1.56%	48076425	1.64%	N/A	N/A	N/A	N/A	5.77	1.56%	48076425	1.64%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	3.21	2.08%	16001341	2.12%	N/A	N/A	N/A	N/A	3.21	2.08%	16001341	2.12%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	3.70	2.42%	12297271	2.82%	N/A	N/A	N/A	N/A	3.70	2.42%	12297271	2.82%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	1.42	15.10%	9362176	16.11%	N/A	N/A	N/A	N/A	1.42	15.10%	9362176	16.11%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	7.33	1.43%	43991489	1.72%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	7.33	1.43%	43991489	1.72%	49	N/A	N/A
GR	16.51	1.66%	75416609	1.77%	2.42	2.05%	15951878	2.11%	N/A	N/A	N/A	N/A	14.09	1.66%	59464731	1.69%	48	N/A	N/A
SAP_AR	11.72	2.91%	57991713	1.94%	3.02	10.49%	10628781	10.41%	8.71	0.89%	47362932	1.05%	N/A	N/A	N/A	N/A	N/A	3	5
SAP_BF	14.32	3.41%	50086138	3.30%	3.15	10.39%	10343945	10.71%	N/A	N/A	N/A	N/A	11.16	2.32%	39742193	2.29%	52	3	5
SAP_BS	7.35	4.80%	26560280	5.26%	3.00	9.59%	10641585	10.85%	4.35	2.72%	15918695	2.97%	N/A	N/A	N/A	N/A	N/A	3	5
SAP_DS	7.82	5.38%	22949835	7.09%	3.05	10.08%	10716921	10.56%	4.77	3.64%	12232913	5.81%	N/A	N/A	N/A	N/A	N/A	3	5
SAP_LD	5.15	6.29%	20436237	7.15%	3.01	9.22%	10953243	10.23%	2.14	9.62%	9483041	11.06%	N/A	N/A	N/A	N/A	N/A	3	5
SBF_AR	11.76	1.17%	48730802	1.28%	1.86	1.96%	7223039	1.18%	3.41	3.10%	18295448	3.43%	6.50	0.55%	23212336	0.33%	30	2	2
SBF_BF	12.58	1.74%	45757552	1.74%	1.86	1.10%	7223587	1.18%	N/A	N/A	N/A	N/A	10.73	1.92%	38533964	1.91%	51	2	2
SBF_BS	10.01	1.26%	36821407	1.20%	1.83	2.16%	7223645	1.18%	1.72	5.97%	6385427	6.09%	6.46	0.93%	23212336	0.33%	30	2	2
SBF_DS	10.52	1.46%	36218759	1.46%	1.87	2.12%	7314295	1.13%	2.09	6.53%	5692128	8.15%	6.56	0.56%	23212336	0.33%	30	2	2
SBF_LD	9.77	2.50%	36827870	3.14%	1.83	2.06%	7261355	1.16%	1.40	16.02%	6354179	17.71%	6.55	0.65%	23212336	0.33%	30	2	2
KR_AR	7.37	1.05%	48676923	1.62%	0.24	2.24%	600498	0.00%	7.13	1.05%	48076426	1.64%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	8.03	1.96%	44591987	1.69%	0.25	1.77%	600498	0.00%	N/A	N/A	N/A	N/A	7.77	2.00%	43991490	1.72%	49	N/A	N/A
KR_BS	3.92	2.02%	15400767	2.61%	0.56	2.97%	1757005	0.16%	3.36	2.64%	13643763	2.93%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	4.43	3.55%	13138802	4.20%	0.59	0.93%	1757930	0.07%	3.84	4.00%	11380873	4.85%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	1.95	13.39%	9962674	15.14%	0.24	2.24%	600498	0.00%	1.70	15.41%	9362225	16.11%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

LONG_GRID_167K: $m = 166800$, $n = 66702$, $v = 48$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	1.75	14.05%	8329302	16.00%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	8.22	14.68%	44797754	15.27%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	1.60	16.51%	7726551	19.44%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	1.50	6.71%	5926962	6.86%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	16.05	1.13%	128860346	0.97%	N/A	N/A	N/A	N/A	16.05	1.13%	128860346	0.97%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	8.63	1.70%	42948925	1.71%	N/A	N/A	N/A	N/A	8.63	1.70%	42948925	1.71%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	10.02	2.16%	33011611	2.38%	N/A	N/A	N/A	N/A	10.02	2.16%	33011611	2.38%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	3.83	7.57%	25025456	8.66%	N/A	N/A	N/A	N/A	3.83	7.57%	25025456	8.66%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	19.81	2.09%	116852544	1.32%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	19.81	2.09%	116852544	1.32%	48	N/A	N/A
GR	44.20	1.33%	200671564	1.26%	6.47	1.22%	42778088	1.71%	N/A	N/A	N/A	N/A	37.73	1.37%	157893476	1.17%	47	N/A	N/A
SAP_AR	31.74	2.94%	152444130	2.26%	7.45	9.85%	25668645	11.46%	24.29	0.97%	126775485	0.80%	N/A	N/A	N/A	N/A	N/A	2	4
SAP_BF	38.10	4.25%	130135294	3.65%	7.77	9.98%	24872504	11.87%	N/A	N/A	N/A	N/A	30.33	3.05%	105262790	2.03%	51	2	4
SAP_BS	19.24	5.02%	68582929	5.41%	7.49	9.44%	25939480	10.74%	11.75	2.54%	42643449	2.59%	N/A	N/A	N/A	N/A	N/A	2	4
SAP_DS	20.44	5.47%	58876253	7.77%	7.35	10.60%	25750445	11.98%	13.09	3.41%	33125808	5.57%	N/A	N/A	N/A	N/A	N/A	2	4
SAP_LD	12.93	10.64%	51071388	12.03%	7.28	9.68%	26407556	11.53%	5.65	12.50%	24663878	13.03%	N/A	N/A	N/A	N/A	N/A	2	4
SBF_AR	30.80	0.83%	140504904	0.85%	5.09	1.78%	19439270	1.46%	17.50	1.30%	91786836	1.06%	8.22	0.53%	29278834	0.25%	14	2	2
SBF_BF	33.86	1.96%	122121608	1.56%	5.09	1.85%	19444423	1.46%	N/A	N/A	N/A	N/A	28.77	2.03%	102677185	1.60%	50	2	2
SBF_BS	21.80	1.33%	80182396	1.57%	5.09	0.72%	19444437	1.46%	8.45	3.00%	31459125	3.13%	8.25	0.98%	29278834	0.25%	14	2	2
SBF_DS	23.36	1.72%	74522004	2.12%	5.09	2.25%	19722100	1.39%	9.91	3.20%	25521071	5.18%	8.36	0.78%	29278834	0.25%	14	2	2
SBF_LD	18.25	2.71%	70990406	3.06%	5.08	1.35%	19590876	1.43%	4.86	9.12%	22120696	9.78%	8.31	0.92%	29278834	0.25%	14	2	2
KR_AR	20.31	0.91%	130496744	0.96%	0.66	0.00%	1636398	0.00%	19.65	0.94%	128860347	0.97%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	21.08	1.75%	118488942	1.31%	0.70	0.64%	1636398	0.00%	N/A	N/A	N/A	N/A	20.38	1.81%	116852545	1.32%	48	N/A	N/A
KR_BS	10.56	1.45%	42071891	1.61%	1.54	1.03%	4794792	0.07%	9.02	1.68%	37277100	1.82%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	11.96	2.39%	35055549	3.37%	1.58	1.48%	4791892	0.07%	10.38	2.89%	30262658	3.90%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	5.27	6.77%	26661854	8.13%	0.66	0.83%	1636398	0.00%	4.62	7.71%	25025504	8.66%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

LONG_GRID_455K: $m = 454550$, $n = 181802$, $v = 47$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	4.24	11.25%	19577206	13.79%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	31.10	11.12%	167500596	4.26%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	3.91	9.88%	18860372	11.84%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	3.89	6.21%	15298338	5.96%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	43.89	0.94%	343878669	0.91%	N/A	N/A	N/A	N/A	43.89	0.94%	343878669	0.91%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	22.86	1.17%	113721234	1.23%	N/A	N/A	N/A	N/A	22.86	1.17%	113721234	1.23%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	26.80	1.35%	87015820	1.51%	N/A	N/A	N/A	N/A	26.80	1.35%	87015820	1.51%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	10.77	13.32%	69031967	14.90%	N/A	N/A	N/A	N/A	10.77	13.32%	69031967	14.90%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	52.06	1.64%	309781935	1.21%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	52.06	1.64%	309781935	1.21%	47	N/A	N/A
GR	117.88	1.11%	532819215	1.20%	17.26	0.81%	113225985	1.22%	N/A	N/A	N/A	N/A	100.62	1.16%	419593230	1.20%	46	N/A	N/A
SAP_AR	86.15	3.33%	411667110	2.73%	20.96	11.37%	73259118	12.11%	65.20	0.84%	338407992	0.73%	N/A	N/A	N/A	N/A	N/A	3	5
SAP_BF	101.81	3.08%	350476550	3.79%	21.83	11.17%	71143665	12.42%	N/A	N/A	N/A	N/A	79.98	1.76%	279332885	1.82%	51	3	5
SAP_BS	51.61	5.12%	185261969	5.54%	20.74	11.03%	72894823	12.18%	30.87	1.76%	112367146	1.98%	N/A	N/A	N/A	N/A	N/A	3	5
SAP_DS	55.15	5.40%	159591598	7.05%	20.93	11.20%	73985545	12.38%	34.21	2.75%	85606053	4.29%	N/A	N/A	N/A	N/A	N/A	3	5
SAP_LD	37.85	8.59%	148796483	8.52%	20.98	11.59%	75601403	11.94%	16.88	6.81%	73195126	7.05%	N/A	N/A	N/A	N/A	N/A	3	5
SBF_AR	90.11	0.51%	427015003	0.60%	22.80	2.46%	80415841	2.32%	64.71	0.66%	36882347	0.62%	2.59	0.65%	9716865	0.00%	2	4	4
SBF_BF	100.42	0.93%	356585432	0.95%	22.57	0.81%	81228848	0.50%	N/A	N/A	N/A	N/A	77.85	1.13%	275356584	1.17%	52	4	4
SBF_BS	56.11	0.73%	202124842	0.71%	22.73	2.16%	80434432	2.32%	30.79	1.29%	111973546	1.37%	2.59	0.69%	9716865	0.00%	2	4	4
SBF_DS	86.09	0.56%	292466890	0.76%	22.48	0.71%	81954080	0.50%	14.30	3.30%	38142898	5.04%	49.30	0.24%	172369911	0.04%	31	4	4
SBF_LD	84.28	1.03%	307690344	1.58%	22.44	0.59%	81434222	0.49%	12.53	7.64%	53886210	8.75%	49.32	0.47%	172369911	0.04%	31	4	4
KR_AR	55.05	0.72%	348251967	0.90%	1.76	0.25%	4373298	0.00%	53.28	0.75%	343878670	0.91%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	55.35	1.12%	314155233	1.19%	1.86	0.38%	4373298	0.00%	N/A	N/A	N/A	N/A	53.49	1.16%	309781936	1.21%	47	N/A	N/A
KR_BS	28.39	1.20%	113697977	1.26%	4.15	0.65%	12806962	0.01%	24.25	1.41%	100891016	1.42%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	32.48	2.22%	93639985	2.84%	4.25	0.44%	12810064	0.05%	28.23	2.56%	80829922	3.29%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	14.58	11.94%	73405265	14.01%	1.75	0.26%	4373298	0.00%	12.83	13.56%	69032014	14.90%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

LONG_GRID_1215K: $m = 1214800$, $n = 485902$, $v = 48$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	0.35	4.26%	1604235	4.16%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	0.48	2.55%	2371877	3.30%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	0.35	12.36%	1679941	12.71%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PF_LO	0.39	3.36%	1626624	3.55%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	2.04	3.14%	16382595	2.98%	N/A	N/A	N/A	N/A	2.04	3.14%	16382595	2.98%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	7.14	0.48%	32365925	0.30%	N/A	N/A	N/A	N/A	7.14	0.48%	32365925	0.30%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	6.88	0.79%	22529034	0.98%	N/A	N/A	N/A	N/A	6.88	0.79%	22529034	0.98%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	1.06	8.60%	6410174	7.70%	N/A	N/A	N/A	N/A	1.06	8.60%	6410174	7.70%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	8.42	3.47%	517535229	3.28%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	8.42	3.47%	517535229	3.28%	133	N/A	N/A
GR	18.76	2.45%	84641641	3.19%	2.53	2.44%	16529905	2.88%	N/A	N/A	N/A	N/A	16.23	2.48%	68111737	3.27%	132	N/A	N/A
SAP_AR	4.86	2.96%	22840709	2.94%	1.33	6.66%	4559173	8.88%	3.53	2.72%	18281537	2.66%	N/A	N/A	N/A	N/A	N/A	2	5
SAP_BF	14.16	1.58%	48091068	1.79%	1.44	11.85%	4614710	10.73%	N/A	N/A	N/A	N/A	12.73	2.26%	43476358	2.45%	150	2	5
SAP_BS	10.58	1.12%	33418893	1.12%	1.34	8.63%	4617398	9.16%	9.24	0.25%	28801494	0.36%	N/A	N/A	N/A	N/A	N/A	2	5
SAP_DS	9.37	2.12%	23837522	2.82%	1.33	6.85%	4668098	9.24%	8.04	1.62%	19169424	1.76%	N/A	N/A	N/A	N/A	N/A	2	5
SAP_LD	3.39	6.92%	13263181	6.64%	1.34	8.94%	4753355	8.96%	2.05	6.34%	8510096	6.19%	N/A	N/A	N/A	N/A	N/A	2	5
SBF_AR	11.62	2.70%	40170006	3.04%	0.00	223.6%	14329	219.4%	0.00	0.00%	0	0.00%	11.61	2.77%	40155677	3.11%	133	0	0
SBF_BF	11.73	2.83%	40170006	3.04%	0.01	223.6%	14329	219.4%	N/A	N/A	N/A	N/A	11.72	2.93%	40155677	3.11%	133	0	0
SBF_BS	11.64	2.90%	40170006	3.04%	0.00	223.6%	14329	219.4%	0.00	0.00%	0	0.00%	11.64	2.97%	40155677	3.11%	133	0	0
SBF_DS	11.75	2.51%	40170006	3.04%	0.00	223.6%	14329	219.4%	0.00	0.00%	0	0.00%	11.75	2.55%	40155677	3.11%	133	0	0
SBF_LD	11.76	2.81%	40170006	3.04%	0.00	136.9%	14329	219.4%	0.00	0.00%	0	0.00%	117.75	2.84%	40155677	3.11%	133	0	0
KR_AR	2.58	322.00%	16605073	2.94%	0.09	0.00%	222478	0.00%	2.49	3.26%	16382596	2.98%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	8.92	3..30%	51957707	3.27%	0.10	4.56%	222478	0.00%	N/A	N/A	N/A	N/A	8.83	3.34%	51735230	3.28%	133	N/A	N/A
KR_BS	7.24	0.33%	25226280	0.81%	0.21	2.66%	626134	0.08%	7.04	0.38%	24600147	0.83%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	6.88	0.75%	19808243	0.64%	0.21	8.82%	625238	0.24%	6.67	0.88%	19183006	0.67%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	1.36	7.76%	6632652	7.44%	0.09	4.86%	222478	0.00%	1.27	8.33%	6410445	7.70%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

CUBE_GRID_62K: $m = 62370$, $n = 27002$, $v = 270$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	1.31	1.83%	5611954	1.83%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	2.01	4.72%	9410474	4.71%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	1.27	6.10%	5903920	6.56%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	1.32	2.95%	5370539	2.77%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	9.88	4.46%	72989474	4.55%	N/A	N/A	N/A	N/A	9.88	4.46%	72989474	4.55%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	39.66	0.40%	175045808	0.47%	N/A	N/A	N/A	N/A	39.66	0.40%	175045808	0.47%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	40.31	1.06%	123552200	0.87%	N/A	N/A	N/A	N/A	40.31	1.06%	123552200	0.87%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	5.62	6.03%	32167488	6.24%	N/A	N/A	N/A	N/A	5.62	6.03%	32167488	6.24%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	40.04	3.21%	235764925	3.39%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	40.04	3.21%	235764925	3.39%	222	N/A	N/A
GR	86.83	3.17%	386655750	3.34%	11.48	3.35%	75770168	3.26%	N/A	N/A	N/A	N/A	75.35	3.15%	310885582	3.36%	221	N/A	N/A
SAP_AR	20.00	1.93%	91948891	2.49%	3.61	1.23%	12191882	0.71%	16.39	2.51%	79757009	2.80%	N/A	N/A	N/A	N/A	N/A	2	5
SAP_BF	64.03	1.92%	212911783	1.91%	3.78	2.70%	11895222	1.20%	N/A	N/A	N/A	N/A	60.25	2.09%	201016561	2.03%	249	2	5
SAP_BS	55.46	0.29%	167870307	0.41%	3.62	0.66%	12271587	0.68%	51.84	0.32%	155598719	0.44%	N/A	N/A	N/A	N/A	N/A	2	5
SAP_DS	48.92	0.60%	114515185	0.67%	3.63	1.37%	12486577	0.91%	45.30	0.71%	102028608	0.73%	N/A	N/A	N/A	N/A	N/A	2	5
SAP_LD	13.03	9.40%	50503445	9.86%	3.63	1.80%	12692268	0.62%	9.40	12.92%	37811704	13.27%	N/A	N/A	N/A	N/A	N/A	2	5
SBF_AR	54.55	3.34%	183795067	3.31%	0.04	46.77%	153832	48.92%	0.00	0.00%	0	0.00%	54.51	3.33%	183641235	3.31%	222	0	0
SBF_BF	54.98	3.13%	183795067	3.31%	0.04	60.01%	153832	48.92%	N/A	N/A	N/A	N/A	54.94	3.12%	183641235	3.31%	222	0	0
SBF_BS	54.66	3.11%	183795067	3.31%	0.04	43.24%	153832	48.92%	0.00	0.00%	0	0.00%	54.62	3.12%	183641235	3.31%	222	0	0
SBF_DS	55.29	3.23%	183795067	3.31%	0.03	64.44%	153832	48.92%	0.00	0.00%	0	0.00%	55.26	3.22%	183641235	3.31%	222	0	0
SBF_LD	55.21	3.11%	183795067	3.31%	0.04	46.48%	153832	48.92%	0.00	0.00%	0	0.00%	55.17	3.11%	183641235	3.31%	222	0	0
KR_AR	12.14	4.29%	73599076	4.51%	0.25	0.00%	609602	0.00%	11.89	4.38%	72989475	4.55%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	41.63	3.15%	236374527	3.38%	0.27	2.06%	609602	0.00%	N/A	N/A	N/A	N/A	41.36	3.17%	235764926	3.39%	222	N/A	N/A
KR_BS	39.51	0.85%	132921249	1.54%	0.59	1.53%	1712905	0.07%	38.93	0.86%	131208345	1.56%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	38.38	1.04%	104491410	1.04%	0.59	0.76%	1715266	0.18%	37.79	1.05%	102776145	1.05%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	6.89	6.11%	32777090	6.12%	0.25	0.00%	609602	0.00%	6.64	6.34%	32168016	6.24%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

CUBE_GRID_171K: $m = 170923$, $n = 74090$, $v = 526$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	Avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	4.24	4.89%	16859695	5.69%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	7.65	6.80%	33281153	6.66%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	4.04	10.34%	18316512	10.90%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	4.12	3.94%	16067225	4.89%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	43.28	2.08%	304647100	2.00%	N/A	N/A	N/A	N/A	43.28	2.08%	304647100	2.00%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	206.03	0.28%	883409315	0.32%	N/A	N/A	N/A	N/A	206.03	0.28%	883409315	0.32%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	213.12	0.44%	628476930	0.47%	N/A	N/A	N/A	N/A	213.12	0.44%	628476930	0.47%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	24.15	5.71%	133720171	6.11%	N/A	N/A	N/A	N/A	24.15	5.71%	133720171	6.11%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	175.53	1.18%	999884378	0.37%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	175.53	1.18%	999884378	0.37%	358	N/A	N/A
GR	374.03	0.31%	1641235008	0.32%	48.55	0.50%	321471145	0.34%	N/A	N/A	N/A	N/A	325.48	0.30%	1319763863	0.34%	357	N/A	N/A
SAP_AR	81.20	2.50%	364718053	2.47%	10.05	9.25%	33938547	10.08%	71.15	2.85%	330779506	2.77%	N/A	N/A	N/A	N/A	N/A	2	5
SAP_BF	279.68	2.10%	914870908	2.05%	10.46	8.80%	33157173	10.06%	N/A	N/A	N/A	N/A	269.21	2.07%	881713736	1.95%	410	2	5
SAP_BS	285.77	0.29%	816649291	0.28%	9.70	1.74%	33033572	1.49%	276.06	0.30%	783615719	0.29%	N/A	N/A	N/A	N/A	N/A	2	5
SAP_DS	246.76	0.26%	546054529	0.17%	9.62	1.10%	33223075	0.45%	237.14	0.23%	512831454	0.17%	N/A	N/A	N/A	N/A	N/A	2	5
SAP_LD	49.57	3.53%	190009449	3.73%	9.69	0.93%	33873801	0.43%	39.88	4.30%	156136651	4.55%	N/A	N/A	N/A	N/A	N/A	2	5
SBF_AR	235.50	0.61%	780041820	0.32%	0.08	45.80%	346075	42.25%	0.00	0.00%	0	0.00%	235.42	0.61%	779695746	0.32%	358	0	0
SBF_BF	237.22	0.27%	780041820	0.32%	0.09	43.57%	346075	42.25%	N/A	N/A	N/A	N/A	237.13	0.28%	779695746	0.32%	358	0	0
SBF_BS	235.10	0.54%	780041820	0.32%	0.08	41.75%	346075	42.25%	0.00	0.00%	0	0.00%	235.01	0.54%	779695746	0.32%	358	0	0
SBF_DS	238.36	0.49%	780041820	0.32%	0.08	45.02%	346075	42.25%	0.00	0.00%	0	0.00%	238.27	0.49%	779695746	0.32%	358	0	0
SBF_LD	238.72	0.29%	780041820	0.32%	0.08	41.94%	346075	42.25%	0.00	0.00%	0	0.00%	238.64	0.29%	779695746	0.32%	358	0	0
KR_AR	52.42	1.87%	306251006	1.99%	0.66	0.00%	1603906	0.00%	51.76	1.89%	304647101	2.00%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	179.63	0.91%	1001488284	0.37%	0.70	0.79%	1603906	0.00%	N/A	N/A	N/A	N/A	178.93	0.91%	999884379	0.37%	358	N/A	N/A
KR_BS	203.43	0.39%	655430369	0.69%	1.55	0.35%	4509963	0.05%	201.88	0.39%	650920407	0.70%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	198.70	0.42%	522524298	0.32%	1.58	0.78%	4510728	0.10%	197.12	0.43%	518013571	0.32%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	29.23	5.77%	135324077	6.03%	0.66	0.83%	1603906	0.00%	28.57	5.89%	133721175	6.11%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

CUBE_GRID_450K: $m = 449755$, $n = 195114$, $v = 1003$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	12.83	4.44%	46468927	5.16%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	25.94	3.95%	105809468	4.26%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	10.77	9.95%	46855619	10.64%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	11.63	4.22%	43531161	6.83%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	172.33	4.56%	1152844766	4.72%	N/A	N/A	N/A	N/A	172.33	4.56%	1152844766	4.72%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	1086.69	0.41%	4411752435	0.35%	N/A	N/A	N/A	N/A	1086.69	0.41%	4411752435	0.35%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	1113.00	0.59%	3160323315	0.40%	N/A	N/A	N/A	N/A	1113.00	0.59%	3160323315	0.40%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	105.58	7.70%	553905434	8.13%	N/A	N/A	N/A	N/A	105.58	7.70%	553905434	8.13%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	735.38	1.23%	4156501806	1.64%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	735.38	1.23%	4156501806	1.64%	567	N/A	N/A
GR	1584.71	1.49%	6825519662	1.64%	202.80	1.86%	1336432565	1.66%	N/A	N/A	N/A	N/A	1381.90	16.11%	1341211	14.30%	566	N/A	N/A
SAP_AR	312.85	3.45%	1377805587	3.73%	29.86	7.57%	103021561	9.18%	283.00	3.15%	1274784026	3.41%	N/A	N/A	N/A	N/A	N/A	2	6
SAP_BF	1181.01	1.39%	3840423488	1.44%	30.01	9.25%	96875493	11.75%	N/A	N/A	N/A	N/A	1151.00	1.46%	3743547995	1.52%	658	2	6
SAP_BS	1529.22	0.48%	4004483842	0.67%	28.97	9.98%	100654398	11.88%	1500.25	0.33%	3903829444	0.42%	N/A	N/A	N/A	N/A	N/A	2	6
SAP_DS	1224.39	0.82%	2624971374	0.69%	25.99	0.91%	88923903	0.25%	1198.41	0.82%	2536047472	0.71%	N/A	N/A	N/A	N/A	N/A	2	5
SAP_LD	191.42	6.51%	716539315	6.86%	26.01	0.44%	90645287	0.25%	165.41	7.51%	625895927	7.86%	N/A	N/A	N/A	N/A	N/A	2	5
SBF_AR	998.68	1.40%	3244336686	1.65%	0.19	27.41%	744117	27.90%	0.00	0.00%	0	0.00%	235.42	0.61%	779695746	0.32%	358	0	0
SBF_BF	1008.12	1.48%	3244336686	1.65%	0.19	31.06%	744117	27.90%	N/A	N/A	N/A	N/A	1007.93	1.47%	3243592568	1.65%	567	0	0
SBF_BS	995.58	1.46%	3244336686	1.65%	0.20	28.66%	744117	27.90%	0.00	0.00%	0	0.00%	995.38	1.46%	3243592568	1.65%	567	0	0
SBF_DS	1013.05	1.37%	3244336686	1.65%	0.19	26.97%	744117	27.90%	0.00	0.00%	0	0.00%	1012.86	1.37%	3243592568	1.65%	567	0	0
SBF_LD	1014.45	1.29%	3244336686	1.65%	0.19	25.34%	744117	27.90%	0.00	0.00%	0	0.00%	1014.26	1.29%	3243592568	1.65%	567	0	0
KR_AR	207.10	4.64%	1157050844	4.71%	1.74	0.26%	4206078	0.00%	205.36	4.68%	1152844767	4.72%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	765.09	1.48%	4160707884	1.64%	1.84	0.00%	4206078	0.00%	N/A	N/A	N/A	N/A	763.25	1.48%	4156501807	1.64%	567	N/A	N/A
KR_BS	1083.24	0.47%	3255359480	0.65%	4.07	0.64%	11825634	0.08%	1079.17	0.48%	3243533847	0.66%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	1020.85	0.82%	2591465692	0.64%	4.19	0.43%	11827837	0.06%	1016.67	0.83%	2579637856	0.64%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	124.95	7.66%	558111512	8.07%	1.73	0.32%	4206078	0.00%	123.22	7.77%	553907334	8.13%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

CUBE_GRID_1180K: $m = 1179520$, $n = 512002$, $v = 1899$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	Avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	0.47	10.92%	2123593	10.75%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	0.52	5.24%	2326121	5.40%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	0.54	31.62%	4299184	106.16%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	0.51	5.07%	2155622	5.75%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	1.83	8.02%	17147964	37.08%	N/A	N/A	N/A	N/A	1.83	8.02%	17147964	37.08%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	22.02	0.49%	93782727	0.13%	N/A	N/A	N/A	N/A	22.02	0.49%	93782727	0.13%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	16.69	0.89%	53726957	1.02%	N/A	N/A	N/A	N/A	16.69	0.89%	53726957	1.02%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	1.68	9.43%	10116108	9.26%	N/A	N/A	N/A	N/A	1.68	9.43%	10116108	9.26%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	9.12	3.93%	55732871	4.08%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	9.12	3.93%	55732871	4.08%	135	N/A	N/A
GR	19.90	4.01%	90737136	4.03%	2.69	4.08%	17614842	4.01%	N/A	N/A	N/A	N/A	17.22	4.01%	73122293	4.03%	134	N/A	N/A
SAP_AR	5.05	4.29%	24552595	4.45%	1.56	9.81%	6424371	14.27%	3.49	4.06%	18128224	3.91%	N/A	N/A	N/A	N/A	N/A	2	5
SAP_BF	18.44	2.78%	65094768	2.71%	1.57	8.61%	6019045	11.54%	N/A	N/A	N/A	N/A	16.87	2.75%	59075723	2.31%	185	2	5
SAP_BS	30.88	0.69%	89048602	1.14%	1.55	9.17%	6544836	14.10%	29.33	0.30%	82503766	0.16%	N/A	N/A	N/A	N/A	N/A	2	5
SAP_DS	21.00	0.89%	52087723	1.70%	1.46	8.47%	6244105	11.30%	19.54	0.41%	45843617	0.50%	N/A	N/A	N/A	N/A	N/A	2	5
SAP_LD	4.72	5.45%	19366289	6.42%	1.51	8.43%	6337046	11.17%	3.20	4.61%	13030084	4.75%	N/A	N/A	N/A	N/A	N/A	2	5
SBF_AR	12.52	4.47%	42857441	4.00%	0.01	136.9%	47257	122.9%	0.00	0.00%	0	0.00%	12.51	4.42%	42810183	3.98%	135	0	0
SBF_BF	12.60	3.87%	42857441	4.00%	0.02	83.85%	47257	122.9%	N/A	N/A	N/A	N/A	12.59	3.84%	42810183	3.98%	135	0	0
SBF_BS	12.37	3.70%	42857441	4.00%	0.01	108.7%	47257	122.9%	0.00	0.00%	0	0.00%	12.35	3.70%	42810183	3.98%	135	0	0
SBF_DS	12.54	3.53%	42857441	4.00%	0.01	141.4%	47257	122.9%	0.00	0.00%	0	0.00%	12.53	3.52%	42810183	3.98%	135	0	0
SBF_LD	12.63	3.60%	42857441	4.00%	0.01	108.3%	47257	122.9%	0.00	0.00%	0	0.00%	12.62	3.59%	42810183	3.98%	135	0	0
KR_AR	2.34	9.71%	17381684	36.58%	0.09	5.83%	233720	0.00%	2.25	9.93%	17147965	37.08%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	9.57	4.04%	55966591	4.06%	0.10	5.27%	233720	0.00%	N/A	N/A	N/A	N/A	9.47	4.11%	55732872	4.08%	135	N/A	N/A
KR_BS	20.87	0.67%	67902941	0.46%	0.21	6.15%	660408	0.40%	20.66	0.64%	67242534	0.46%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	16.95	1.32%	47794703	1.08%	0.25	6.16%	660027	0.20%	16.71	1.27%	47134677	1.10%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	2.12	9.31%	10349828	9.05%	0.10	5.71%	233720	0.00%	2.02	9.74%	10116951	9.25%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

WIDE_GRID_65K: $m = 65453$, $n = 28092$, $v = 842$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	1.69	9.84%	6755499	10.86%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	1.82	11.40%	7460369	12.42%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	1.74	9.03%	7879758	9.57%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	1.84	9.06%	7292404	9.37%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	7.07	10.64%	53394846	11.10%	N/A	N/A	N/A	N/A	7.07	10.64%	53394846	11.10%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	156.51	0.48%	604131853	0.26%	N/A	N/A	N/A	N/A	156.51	0.48%	604131853	0.26%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	114.34	0.87%	336939591	1.08%	N/A	N/A	N/A	N/A	114.34	0.87%	336939591	1.08%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	8.89	12.75%	48975544	13.08%	N/A	N/A	N/A	N/A	8.89	12.75%	48975544	13.08%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	39.22	3.00%	230572713	3.20%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	39.22	3.00%	230572713	3.20%	216	N/A	N/A
GR	84.20	3.16%	375149976	3.12%	11.02	2.73%	72611900	3.09%	N/A	N/A	N/A	N/A	73.18	3.24%	302538076	3.13%	215	N/A	N/A
SAP_AR	17.79	4.12%	89690830	5.21%	4.78	7.84%	25919170	11.93%	13.01	4.94%	63771660	5.59%	N/A	N/A	N/A	N/A	N/A	2	6
SAP_BF	74.98	1.38%	266373814	1.44%	4.98	8.58%	25634106	12.05%	N/A	N/A	N/A	N/A	70.00	1.70%	240739708	1.49%	285	2	6
SAP_BS	218.19	0.51%	557444264	1.07%	4.62	10.23%	24810605	15.26%	213.57	0.34%	532633659	0.46%	N/A	N/A	N/A	N/A	N/A	2	6
SAP_DS	131.86	1.67%	301841315	2.28%	4.16	8.32%	21900859	14.76%	127.70	1.48%	279940456	1.43%	N/A	N/A	N/A	N/A	N/A	2	5
SAP_LD	19.24	3.86%	79057337	3.85%	4.07	1.21%	207793736	0.90%	15.17	4.84%	58280127	5.08%	N/A	N/A	N/A	N/A	N/A	2	5
SBF_AR	52.60	3.02%	177072553	3.18%	0.04	63.74%	154921	61.93%	0.00	0.00%	0	0.00%	52.56	2.99%	176917632	3.15%	216	0	0
SBF_BF	53.05	3.26%	177072553	3.18%	0.04	57.05%	154921	61.93%	N/A	N/A	N/A	N/A	53.01	3.23%	176917632	3.15%	216	0	0
SBF_BS	52.24	3.40%	177072553	3.18%	0.04	57.05%	154921	61.93%	0.00	0.00%	0	0.00%	52.20	3.38%	176917632	3.15%	216	0	0
SBF_DS	53.31	3.31%	177072553	3.18%	0.04	63.74%	154921	61.93%	0.00	0.00%	0	0.00%	53.27	3.29%	176917632	3.15%	216	0	0
SBF_LD	53.48	3.03%	177072553	3.18%	0.04	80.03%	154921	61.93%	0.00	0.00%	0	0.00%	53.44	3.00%	176917362	3.15%	216	0	0
KR_AR	8.77	10.39%	53995986	10.98%	0.25	1.80%	601140	0.00%	8.52	10.69%	53394847	11.10%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	40.52	350.00%	231173853	3.20%	0.26	0.00%	601140	0.00%	N/A	N/A	N/A	N/A	40.26	3.52%	230572714	3.20%	216	N/A	N/A
KR_BS	147.41	0.31%	432485445	0.58%	0.58	1.45%	1697828	0.18%	146.83	0.31%	430787618	0.58%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	112.98	0.58%	297951911	0.71%	0.59	1.95%	1692112	0.23%	112.40	0.58%	296259800	0.72%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	10.79	12.46%	49576684	12.92%	0.25	1.80%	601140	0.00%	10.54	12.73%	48977708		N/A	N/A	N/A	N/A	N/A	N/A	N/A

WIDE_GRID_168K: $m = 168348$, $n = 72252$, $v = 2163$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	6.27	10.81%	22056031	12.26%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	6.68	10.15%	25803255	10.90%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	6.50	17.74%	30906290	35.47%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	6.13	9.09%	22901392	9.67%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	26.83	15.38%	192581248	15.97%	N/A	N/A	N/A	N/A	26.83	15.38%	192581248	15.97%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	1251.88	0.29%	4292477200	0.45%	N/A	N/A	N/A	N/A	1251.88	0.29%	4292477200	0.45%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	839.36	0.69%	2352579576	0.74%	N/A	N/A	N/A	N/A	839.36	0.69%	2352579576	0.74%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	47.37	12.29%	240359405	12.92%	N/A	N/A	N/A	N/A	47.37	12.29%	240359405	12.92%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	174.98	5.97%	978251615	6.54%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	174.98	5.97%	978251615	6.54%	334	N/A	N/A
GR	365.56	6.27%	1586366703	6.49%	46.54	6.51%	304811182	6.42%	N/A	N/A	N/A	N/A	319.02	6.24%	1281555521	6.51%	333	N/A	N/A
SAP_AR	66.07	9.25%	368671172	8.49%	16.49	0.94%	136343490	0.26%	49.58	12.44%	232327683	13.36%	N/A	N/A	N/A	N/A	N/A	2	6
SAP_BF	326.52	4.17%	1164904484	3.52%	16.64	7.50%	127475607	14.12%	N/A	N/A	N/A	N/A	309.88	4.58%	1037428877	4.67%	442	2	6
SAP_BS	1657.82	0.30%	3910673214	0.53%	16.45	0.85%	136940751	0.33%	1641.37	0.30%	3773732463	0.54%	N/A	N/A	N/A	N/A	N/A	2	6
SAP_DS	941.90	0.62%	2049302149	0.74%	12.94	0.84%	95789319	0.36%	928.96	0.63%	1953512829	0.77%	N/A	N/A	N/A	N/A	N/A	2	5
SAP_LD	87.40	10.50%	367597673	9.52%	13.12	2.15%	96845854	0.51%	74.28	12.01%	270757662	12.83%	N/A	N/A	N/A	N/A	N/A	2	5
SBF_AR	227.23	6.12%	748245429	6.52%	0.07	75.00%	296958	78.54%	0.00	0.00%	0	0.00%	227.15	6.11%	747948472	6.51%	334	0	0
SBF_BF	228.91	6.28%	748245429	6.52%	0.08	76.60%	296958	78.54%	N/A	N/A	N/A	N/A	228.83	6.27%	747948472	6.51%	334	0	0
SBF_BS	226.55	6.35%	748245429	6.52%	0.08	76.60%	296958	78.54%	0.00	0.00%	0	0.00%	226.47	6.35%	747948472	6.51%	334	0	0
SBF_DS	230.12	6.21%	748245429	6.52%	0.08	76.60%	296958	78.54%	0.00	0.00%	0	0.00%	230.04	6.21%	747948472	6.51%	334	0	0
SBF_LD	230.04	6.05%	748245429	6.52%	0.07	80.29%	296958	78.54%	0.00	0.00%	0	0.00%	229.96	6.05%	747948472	6.51%	334	0	0
KR_AR	33.36	15.06%	194211966	15.84%	0.69	0.80%	1630718	0.00%	32.68	15.39%	192581249	15.97%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	178.12	5.91%	979882333	6.53%	0.72	0.76%	1630718	0.00%	N/A	N/A	N/A	N/A	177.39	5.93%	978251616	6.54%	334	N/A	N/A
KR_BS	1161.33	0.40%	3010019190	0.89%	1.58	0.69%	4599717	0.22%	1159.75	0.40%	3005419474	0.89%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	825.54	0.42%	2074250222	0.54%	1.63	1.30%	4602731	0.27%	823.91	0.42%	2069647492	0.55%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	56.49	12.21%	241990121	12.83%	0.68	0.80%	1630718	0.00%	55.80	12.36%	240365247	12.92%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

WIDE_GRID_457K: $m = 456680$, $n = 196002$, $v = 5843$

	running time		arc fetches		running time		arc fetches		running time		arc fetches		running time		arc fetches		calls to	calls	calls
	total				sparsification				augmenting paths				blocking flows				blocking	to	to
	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	avg	std dev	flows	SA2	AP
PR_FIFO	23.16	4.57%	73007063	5.44%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_HI	24.28	7.73%	89099684	8.49%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LIFO	282.28	85.31%	3322439259	90.58%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
PR_LO	19.95	9.29%	70622750	10.14%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_AR	93.06	5.73%	629216979	16.98%	N/A	N/A	N/A	N/A	93.06	5.73%	629216979	16.98%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_BS	∞	∞	∞	∞	N/A	N/A	N/A	N/A	∞	∞	∞	∞	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_DS	∞	∞	∞	∞	N/A	N/A	N/A	N/A	∞	∞	∞	∞	N/A	N/A	N/A	N/A	N/A	N/A	N/A
AP_LD	241.10	0.05%	1127548833	10.52%	N/A	N/A	N/A	N/A	241.10	0.05%	1127548833	10.52%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
DINIC	726.52	5.00%	3969947976	5.53%	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	726.52	5.00%	3969947976	5.53%	503	N/A	N/A
GR	1521.40	5.14%	6398696346	5.43%	187.05	5.10%	1216855315	5.52%	N/A	N/A	N/A	N/A	1334.35	5.15%	5181841031	5.41%	502	N/A	N/A
SAP_AR	239.92	6.89%	1575575479	5.55%	68.84	3.07%	792612667	0.11%	171.08	9.86%	782962812	11.10%	N/A	N/A	N/A	N/A	N/A	2	6
SAP_BF	1289.85	3.02%	4769935812	2.94%	78.37	5.80%	790533625	0.12%	N/A	N/A	N/A	N/A	1211.49	3.40%	3979402188	3.52%	625	2	6
SAP_BS	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	N/A	N/A	N/A	N/A	N/A	∞	∞
SAP_DS	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	N/A	N/A	N/A	N/A	N/A	∞	∞
SAP_LD	387.53	6.07%	1722538378	4.94%	51.26	3.40%	536670706	0.21%	336.27	6.79%	1185883152	7.15%	N/A	N/A	N/A	N/A	N/A	2	5
SBF_AR	948.40	5.39%	3016687616	5.46%	0.35	23.82%	1269905	22.42%	0.00	0.00%	0	0.00%	948.05	5.39%	3015417711	5.47%	503	0	0
SBF_BF	957.11	5.28%	3016687616	5.46%	0.34	22.28%	1269905	22.42%	N/A	N/A	N/A	N/A	956.77	5.28%	3015417711	5.47%	503	0	0
SBF_BS	952.04	5.25%	3016687616	5.46%	0.35	22.85%	1269905	22.42%	0.00	0.00%	0	0.00%	951.69	5.26%	3015417711	5.47%	503	0	0
SBF_DS	960.10	5.32%	3016687616	5.46%	0.34	22.21%	1269905	22.42%	0.00	0.00%	0	0.00%	959.75	5.32%	3015417711	5.47%	503	0	0
SBF_LD	958.79	5.25%	3016687616	5.46%	0.34	22.53%	1269905	22.42%	0.00	0.00%	0	0.00%	958.45	5.26%	3015417711	5.47%	503	0	0
KR_AR	112.55	15.53%	633542037	16.86%	1.82	0.25%	4325058	0.00%	110.73	15.78%	629216980	16.98%	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_BF	747.28	5.16%	3974273034	5.35%	1.93	0.00%	4325058	0.00%	N/A	N/A	N/A	N/A	745.35	5.18%	3969947977	5.36%	503	N/A	N/A
KR_BS	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_DS	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KR_LD	278.39	10.17%	1131873887	10.48%	1.81	0.30%	4325058	0.00%	276.58	10.24%	1127564309	10.52%	N/A	N/A	N/A	N/A	N/A	N/A	N/A

WIDE_GRID_1211K: $m = 1211225$, $n = 519842$, $v = 15479$

Bibliography

[1] S. Even and R. E. Tarjan. Network Flow and Testing Graph Connectivity. *SIAM Journal on Computing, 4:507-518, 1975.*

[2] 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 a Network with Special Structure and Some Applications.

[3] D. R. Karger and M. S. Levine. Finding Maximum Flows in Undirected Graphs Seems Easier than Bipartite Matching. In *Proceedings of the 30th ACM Symposium on Theory of Computing*, pages 69-78. ACM, ACM Press, May 1998.

[4] A. Goldberg and S. Rao. Flows in undirected unit capacity networks. In *Proceedings of the 30th Annual Symposium on the Foundations of Computer Science*, pages 2-11. IEEE, IEEE Computer Society Press, Oct. 1997.

[5] D. R. Karger. Random Sampling in cut, flow, and network design problems. In *Proceedings of the 26th ACM Symposium on Theory of Computing*, pages 648-657. ACM, ACM Press, May 1994.

[6] B. Cherkassky, A. Goldberg, P. Martin, J. Setubal, J. Stolfi. Augment or Push? A computational study of Bipartite Matching and Unit Capacity Flow Algorithms. NEC Research Institute, NEC Research Institute TR #98-036R , 1998.

[7] L. R. Ford, Jr. and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399-404, 1956.

[8] E. A. Dinic. Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation. *Soviet Math. Dokl.*, 11:1277-1280, 1970.

[9] D. R. Karger. Using random sampling to find maximum flows in uncapacitated undirected graphs. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 240-249. ACM, ACM Press, May 1997.

[10] D. R. Karger. Better random sampling algorithms for flows in undirected graphs. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 490-499. ACM-SIAM, January 1998.

[11] H. Nagamochi and T. Ibaraki. A Linear-Time Algorithm for Finding a Sparse *k*-Connected Spanning Subgraph of a *k*-Connected Graph. *Algorithmica*, 7:583-596, 1992.

[12] C. Chekuri, A. Goldberg, D. Karger, M. Levine, C. Stein. Experimental Study of Minimum Cut Algorithms. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 324-333, 5-7. ACM-SIAM, January 1997.

[13] 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.

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

[15] H. Alt, N. Blum, K. Melhorn, 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.

[16] J. Cheriyan and K. Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15:521-549, 1996.

[17] M. S. Levine. Experimental Study of Minimum Cut Algorithms. MS thesis, M.I.T., May 1997. (Also available as Technical Report MIT-LCS-TR-719, Lab. for Computer Science, M.I.T., 1997).

[18] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[19] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362-391, June 1983.

[20] Z. Galil and X. Yu. Short length versions of Menger's theorem (extended abstract). In *Proceedings of the 27th ACM Symposium on Theory of Computing*, pages 499-508. ACM, ACM Press, May 1995.

[21] M. R. Henzinger, J. Kleinberg, and S. Rao. Short-length Menger theorems. Technical Report 1997-022, Digital Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, 1997.

[22] A.A Benczúr and D. R. Karger. Approximate *s-t* mincuts in $\tilde{O}(n^2)$ time. In G. Miller, editor, *Proceedings of the 28th ACM Symposium on Theory of Computing*, pages 47-55. ACM, ACM Press, May 1996.

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

[24] A. V. Goldberg and R. Kennedy. Global Price Updates Help. *SIAM J. Disc. Math.*, 10:551-572, 1997.

[25] 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, pages 23-30, 1994.

[26] 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.

[27] D. S. Johnson. A Theoretician's Guide to the Experimental Analysis of Algorithms. Preliminary, partial draft (4/30/96). (Available from http://www.research.att.com/~dsj/papers/exper.ps).

[28] J. N. Hooker. Needed: An empirical science of algorithms. *Operations Research*, 42:201-212, 1994.