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

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

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

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

Technical report #98-036R
NEC Research Institute, Inc.

March 1998

This TR is a revision of the TR #97-127. The original TR was based on implementations written in different languages, different style, and using somewhat different low-level data structures. After learning that in some cases this results in a non-trivial performance difference, we rewrote some of the codes. The new codes are written in a uniform way. We also provide more sanity checks and better machine-independent measure of performance with our data. Finally, we expanded some parts of the report.

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

Abstract

We conduct a computational study of unit capacity flow and bipartite matching algorithms. Our goal is to determine which variant of the push-relabel method is most efficient in practice and to compare push-relabel algorithms with augmenting path algorithms. We have implemented and compared three push-relabel algorithms, three augmenting path algorithms (one of which is new), and one augment-relabel algorithm. The depth-first search augmenting path algorithm was thought to be a good choice for the bipartite matching problem, but our study shows that it is not robust. For the problems we study, our implementations of the FIFO and lowest-level selection push-relabel algorithms have the most robust asymptotic rate of growth and work best overall. Augmenting path algorithms, although not as robust, on some problem classes are faster by a moderate constant factor. Our study includes several new problem families and input graphs with as many as 5×10^5 vertices.

1 Introduction

Maximum unit capacity flow and bipartite matching are two closely related combinatorial problems that have been extensively studied from the point of view of theoretical analysis of algorithms [21, 22, 23, 13, 3, 6, 14]. The best bound for the former problem is $O(m^{1.5})$ [13, 22] and for the latter it is $O(\sqrt{n}m \log(n^2/m)/\log n)$ [14] or, for dense graphs on a RAM with $\Omega(n)$-bit words, $O(n^{2.5}/\log n)$ [6]. Here n is the number of vertices and m is the number of arcs. In this paper we study computational efficiency of bipartite matching and unit capacity flow algorithms.

We are aware of three previous published computational studies of bipartite matching algorithms [9, 5, 27] and of no computational studies of unit capacity flow algorithms. Our study includes a refinement of a previous study by Setubal [28], which itself refined the results in [27] but has not been published in journal form.

It is well-known that the bipartite matching problem is a special case of the unit capacity flow problem. Hence any implementation that solves the latter can also solve the former. However, we have two sets of codes, one for each problem, and our codes for the bipartite matching problem take advantage of its special structure. The algorithms we study include augmenting path algorithms, push-relabel algorithms, and an augment-relabel algorithm.

The augmenting path algorithms we study find augmenting paths using depth-first search (DFS), breadth-first search (BFS), and a new label-directed depth-first search (LDS). For the bipartite matching problem, Darby-Dowman [9] compared BFS and DFS and found that no implementation performed "consistently better" than the other. Chang and McCormick [5] studied different variants of the depth-first search algorithm. Setubal [27] showed that a push-relabel implementation for bipartite matching was significantly faster than an implementation based on Hopcroft-Karp's algorithm [21] (which is similar to Dinitz's maximum flow algorithm [12] applied to the bipartite matching problem).

For the maximum flow problem with arbitrary capacities, several studies have shown that the push-relabel method is superior to other known methods in practice [4, 8, 10, 11, 26]. Push-relabel algorithms based on the highest-level selection rule appear to be the fastest. For the unit capacity case, Goldberg and Kennedy [16] showed that the lowest-level selection rule, in combination with the global relabeling heuristic, is theoretically efficient. We study the highest-level (HI), lowest-level (LO), and first-in, first-out (FIFO) variants of the push-relabel method. We also study an augment-relabel algorithm (AR) [1] that finds augmenting paths using relabel operations.

The main contributions of this work are new insights into the computational performance of the algorithms and new instance generators for the problems considered. Our study includes more algorithms and more problem families than the previous studies. Our study is the first computational work on the push-relabel method for unit-capacity network, and confirms the theoretical result on the lowest-level selection rule. We show that the push-relabel method is superior to the augmenting path method.

In terms of code performance we arrived at the following conclusions, which need to be taken in the context of our problem families.

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

Out of the augmenting path codes, LDS is the most robust. There are some problem families where BFS and DFS outperform other codes by a small to moderate constant factor, but on other families these codes lose asymptotically. We also found that DFS had very poor performance on some families compared to BFS. (Note that our implementations of breadth-first search algorithms are more efficient than previous implementations due to heuristics we use.) Finally, while the HI and AR codes are competitive on a few problem families, they often perform poorly.

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

2 Definitions

Given a graph $G = (V, E)$, we let $n = |V|$ and $m = |E|$.

2.1 Unit Capacity Flow

The unit capacity maximum flow problem is defined as follows. Let $G = (V, E)$ be a digraph with special vertices s and t. A *pseudoflow* is a function $f : E \rightarrow \{0, 1\}$. For a pseudoflow f and a vertex v, the *excess flow of* v, $e_f(v)$, is defined by $\sum_{(u,v)\in E} f(u, v) - \sum_{(v,w)\in E} f(v, w)$. A *preflow* is a pseudoflow with the property that the excess of every vertex except s is nonnegative.

A *flow* is a pseudoflow f such that, for each vertex $v \in V - \{s, t\}$, $e_f(v) = 0$. The *maximum unit capacity flow problem* is to find a flow maximizing $e_f(t)$.

Next we define the residual graph. Let $\overline{E}$ be the set of all arcs in E and all their reverses, and let $\overline{G} = (V, \overline{E})$. Given a pseudoflow f, the *residual graph* G_f of a pseudoflow f is induced in $\overline{G}$ by the arc set $E_f \subseteq \overline{E}$ containing all arcs of E with zero flow and reversals of all arcs of E with unit flow. An *augmenting path* is a path from s to t in G_f. Note that if P is an augmenting path for a pseudoflow f, if we complement $f(e)$ for every arc $e \in P$, we obtain a pseudoflow f' such that $e_{f'}(s) = e_f(s) - 1, e_{f'}(t) = e_f(t) + 1$, and $e_{f'}(v) = e_f(v)$ for every other vertex v.

A *residual flow value* of a pseudoflow f is the difference between the maximum flow value and the current excess $e_f(t)$ at the sink.

2.2 Bipartite Matching

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

The bipartite matching problem can be transformed into a unit capacity flow problem by the following simple transformation [25]: add a source s, and an arc from s to each vertex of X; add a sink t and an arc from each vertex of Y to t; and replace each arc (u, v), $u \in X, v \in Y$ with an arc from u to v.

3 Implementations

The maximum flow algorithms we consider can be divided into two general classes.

An **augmenting path** algorithm maintains a proper flow f (with zero excess at every vertex except s and t) at all times. The algorithm repeatedly seeks an augmenting path P and uses it to increase $e_f(t)$. The algorithm stops when no such augmenting path exists.

A **push-relabel** algorithm maintains some preflow f, which may have excess flow at vertices other than s and t. The algorithm repeatedly looks for a vertex $u \neq t$ with excess flow, and tries to reduce its excess by complementing $f(e)$ on some arc of G_f out of u, preferably one that may lead to the sink. The algorithm stops when there are no vertices other than s with positive excess.

Our implementations take as input a graph or a bipartite graph in the adjacency list format. We store arcs in an array sorted by indices of tail vertices so that for every vertex, its outgoing

arcs are adjacent in the array. This representation allows very efficient scan of outgoing arcs of a vertex. Space locality yields good cache behavior.

3.1 Initialization

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

For the bipartite matching problem, our augmenting path algorithms use a greedy algorithm to find an initial maximal matching (scanning vertices in their input order, we try to match each vertex in turn). For the push-relabel method we found that using the greedy initial matching never sped up and sometimes slowed down the implementations. All variants of the push-relabel algorithm start with the empty matching.

3.2 Flow Representation

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

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

3.3 Global Relabeling

The global relabeling heuristic improves computational performance of the push-relabel method (see e.g. [4, 15, 26]). This heuristic uses breadth-first search to compute distances to the sink in the residual graph with unit arc lengths; the distance labels are set to these distances. Such a computation takes linear time.

If global relabelings are performed too often, they dominate the running time. If they are performed too rarely, the number of operations performed by the algorithm proper increases. A theoretically good strategy is to amortize the work of global relabelings over other operations of the algorithm. In particular, one can perform a new global relabeling after the algorithm performed $\Theta(m)$ work since the last global relabeling [16]. Our implementations use a simplified version of this strategy, which seems to work well in practice: We perform a new global relabeling after the number of relabel operations since the last global relabeling is $\Theta(n)$.

All our push-relabel implementations use the global relabeling heuristic. Our augment-relabel and label-directed depth-first search use it as well. In all our tests, we performed a global relabeling after n relabels.

3.4 Push-Relabel

We now briefly refine the description of the push-relabel algorithm given above (a full description can be found in [19]). The algorithm maintains for each vertex u a nonnegative integer label, which is a lower bound of the distance from u to the sink in the residual graph. The algorithm works by applying the push and relabel operations to *active* vertices. To be active a vertex must have positive excess flow and a label below n. A push operation applied to a vertex u consists of moving a single unit of flow to one of its neighbors in the residual graph that has a label with a value one less than its own. A relabel operation occurs when no such neighbor exists and consists of increasing the label of u until such a neighbor exists or until the label reaches n. When there are no active vertices, all that remains is to return excess flow to the source. We do this by finding a succession of paths and cycles along which we can remove flow [18]. In the case of bipartite matching this last step is unnecessary, since the only vertices that may have any excess flow are the source's neighbors.

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

The bucket data structure used for HI and LO allows us to implement the gap heuristic [7, 10] with little additional effort. The gap heuristic takes advantage of the fact that if there are no vertices with label d then we can ignore all vertices with label greater than d, since they cannot reach the sink. For FIFO the gap heuristic requires a more expensive data structure which usually outweighs any gain from applying the heuristic, so we do not use it in FIFO.

For the special case of bipartite matching, we exploit the special structure of the problem in order to make the algorithms more efficient. We replace the push operation with a *double-push* operation [2, 17] where a unit of excess flow is pushed first from a vertex in X to a vertex in Y and then is either pushed to the sink or is pushed to a vertex in X. The second portion of the double-push can be performed quite efficiently since all vertices in Y have only a single arc directed out of them in the residual graph. This is true initially and it is easy to see that by using only double-pushes we maintain this property. Additionally since each vertex $v \in Y$

has exactly one arc (v, w) directed out of it the label of v must be the label of w plus one. Therefore we do not need to explicitly represent labels for vertices of Y. See [16, 24] for more details.

3.5 Depth-First Search

This implementation uses depth-first search to find augmenting paths. A search is attempted from each of the source's neighbors. It includes the following heuristics (due to Chang and McCormick [5]). The *clean-up heuristic* deletes all vertices visited during a search if the search fails. The *look-ahead* heuristic examines all neighbours of a vertex in X when this vertex is visited for the first time during a search. If a neighour is unmatched, the search terminates. These heuristics substantially improve the performance on some problem classes.

3.6 Breadth-First Search

This implementation uses breadth-first search to find augmenting paths. A search is attempted from each of the source's neighbors. This differs from the standard shortest augmenting path algorithm where the search is done from the source itself. This difference often leads to a big performance improvement, especially in the bipartite matching case. We also use the clean-up heuristic.

In the bipartite matching case, the difference between our implementation and the standard algorithm is as follows. The standard algorithm finds a shortest augmenting path while our variant finds a shortest augmenting path from a selected unmatched vertex. Augmenting paths for the latter tend to be a little longer, but the regions of the graph examined during the search tend to be much smaller. Our breadth-first search is written so that the time to do a search and to reinitialize for the new search is proportional to the size of the graph visited during the search. Often, this is a small fraction of the graph size.

A simple analysis shows that the DFS and BFS algorithms have complexity $O(nm)$.

3.7 Augment-Relabel

The augment-relabel algorithm [1] uses depth-first search in combination with the relabel operation to find augmenting paths. For the bipartite matching problem, Alt et al. [3] and Cheriyan and Mehlhorn [6] used a variant of the method in combination with certain word operations to obtain improved bounds for dense graphs. Our implementation of this method, AR, does not use such word operations.

The AR algorithm can be thought of as a cross between the simple search and the push-relabel algorithms, in the following sense: augmenting paths are sought, but distance labels on the vertices are used to determine the directions in which paths are extended. These labels are the same as those in the push-relabel algorithm. If a path cannot advance, a relabel operation is applied, and the current path is shrunk. For the bipartite matching case, a queue is used to manage the unmatched vertices in X. Every time the global relabeling routine is invoked, it flushes the queue and fills it with the currently unmatched vertices in X with labels below n, in increasing order of their labels.

A standard argument shows that the AR algorithm takes $O(nm)$ time. (Note that there are $O(n)$ global relabelings, each taking $O(m)$ time.) As observed in [3, 1], one can obtain better theoretical bounds by processing vertices with distance labels bounded by a parameter less than n and using the shortest augmenting path method to complete the computation. However, we were unable to obtain consistent performance gains using this idea, and the implementations we report upon do not switch to the augmenting path algorithm.

3.8 Label-Directed Depth-First Search

This is a new algorithm and it uses a relaxation of the distance labels to select an arc to explore next during a depth-first search. The algorithm maintains vertex labels that do not have the distance label properties (labels may decrease, whereas in push-relabel or augment-relabel algorithms they never do [19]), but are nevertheless estimates on the distances to the sink in the residual graph.

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

This algorithm runs in time $O(nm)$ if we perform a new global relabeling after $\Theta(m)$ work was performed since the last global relabeling. After each global relabeling, one augmenting path can be found in time $O(m)$ (or else there are no augmenting paths and the algorithm is finished). After this, we may spend at most $O(m)$ time searching for another augmenting path without success; then we spend $\Theta(m)$ in the next global relabeling, and repeat. There are at most n augmenting paths, and therefore total running time is $O(nm)$. As mentioned

above, we found that applying a global update after every $\Theta(n)$ relabel operations gives better performance in practice, although the theoretical upper bound just given is no longer valid.

4 Experimental Setup

The experiments were performed on a 200-MHz Pentium Pro-based personal computer running the RedHat 4.0 distribution of Linux with 256 MB of main memory. All our codes are written in C++ and use exactly the same input and graph operation routines, which work on exactly the same graph data structure (described at the beginning of Section 3 above). The codes were compiled with the GNU gcc and g++ compilers respectively with the O4 optimization flag set.

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

4.1 Fewg and manyg

These are random bipartite graphs where the vertices of each partition, X and Y, are divided into k groups of equal size. For each vertex of the j-th group of X the generator chooses y random neighbors from the $(i-1)$-th through $(i+1)$-th groups of Y (with wrap-around), where y is binomially distributed with mean d (thus d = mean vertex degree). The indices i and j are not related because vertices in X are randomly shuffled before neighbors in Y are assigned. The two families we consider are fewg, where there are 32 groups, and manyg, where there are 256 groups; both have $d = 5$.

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

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

4.2 Hi-Lo

The hi-lo family of bipartite matching problems separates high and low vertex selection strategies for the push-relabel method. This generator creates a graph with a unique perfect matching and has been motivated by a generator of Kennedy [24].

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

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

4.3 Hexa

In class hexa, the vertices on each side are divided into n/b blocks of size b. One random bipartite hexagon is added between each block i on one side and each of the blocks $i + k$ on the other side, with $|k| \le K$ for some K. The parameters b and K are chosen by the program in such a way that the average degree is correct (i.e., $3K/b = d$) but few pairs of hexagons have more than one vertex in common. In our tests we set $d = 6$. For this class, the greedy matching paired about 83% of the vertices, and the maximum matchings paired about 95% of the vertices.

4.4 Rope

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

The only maximum matching is a perfect one, consisting of all m-type arcs. The greedy algorithm tends to use r-type arcs, since they are more numerous, and hence tends to fall short of the maximum; in our tests (with $d = 6$), it paired about 90% of the vertices.

4.5 Zipf

Each member of class zipf is a random bipartite graph where the arc between u_i and v_j has nominal probability roughly proportional to $1/(ij)$. Thus the graph is denser near the "core" vertices u_0, v_0, and thins out slowly towards the "periphery". In our experiments we set $d = 6$

and the greedy matching paired between 62 and 69% of the vertices, while the maximum paired about 5% more vertices.

4.6 Karz

The karz family of unit capacity flow problems is inspired by the worst-case problem family developed by Karzanov [22] for a bipartite matching algorithm.

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

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

4.7 rmfuC, rmfuL, rmfuL

The RMFU family is generated by a unit capacity version of the RMF generator due to Goldfarb and Grigoriadis [20]. This generator takes parameters a, b, and c. The graphs that it generates have b "frames" in addition to the source and the sink. Each frame is an $a \times a$ grid, with each vertex of the grid connected to the vertices in all four directions, with wrap-around at the arcs of the grid. The source is connected to c randomly selected vertices of frame 1. For $1 \leq i < b$, we introduce c arcs from frame i to frame $i+1$ by selecting c random vertices at each frame and pairing up the selected vertices. Finally, c randomly selected vertices of frame b are connected to the sink. The three variants we consider are: cube (rmfuC), $a = b$, $c = 0.3a^2$; wide (rmfuW), $b = 10$, $c = 0.3a^2$; and long (rmfuL), $a = 10$, $c = 50$.

4.8 Unimf

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

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

randomly, with a bias towards arcs that create short s-t paths. Note that the maximum flow, which is unique, must *not* use any of these arcs.

In the families blow, saus, and squa, the graph consists of a single block. In family blow the graph has $n/4$ paths of length 4; in family saus it has 3 paths of length $n/3$; and in family squa it has $n^{1/2}$ paths of length $n^{1/2}$. In family wave, the graph consists of 9 blocks connected in series, each block having $n/27$ paths of length 4. In family puff, the graph consists of $\frac{1}{2}n^{1/2}$ blocks in series, and each block has $\frac{1}{2}n^{1/2}$ paths of length 4.

5 Experimental Results

We report the following data for each implementation, problem family, and problem size: running time, standard deviation, operation count, and time per operation. We tabulate the data by problem family and plot the running times as well. The results for the bipartite matching implementations are given in Tables 2 through 8 and Figures 1 through 7. The results for the unit capacity flow implementations are given in Tables 9 through 17 and Figures 8 through 16.

Times are reported in seconds and represent the mean of runs on five instances generated with different seeds. In addition to the mean, we give the standard deviation (as a percentage of the mean). Variance in running time for a given implementation and group of five instances was in general low. We did observe more variance in bipartite matching instances than in flow instances. Such variations do not affect our conclusions, but they do point to the difficulty that exists in obtaining a set of distinct instances for a given family and size and and yet of "uniform hardness" to an implementation. For example, in family hexa, $|V| = 262144$, there was one instance among the five that required significantly more time of most implementations than the other four. Code HI, for instance, solved this problem in 4.78 seconds, while the other four were solved in 1.40 ± 0.07 seconds.

We use operation counts as a machine-independent measure of performance. We count *arc fetches:* each time an algorithm examines one or more fields of an arc, we increment this operation count. In the tables, we give the mean number of arc fetches per input arc. This implicitly assumes that the work of the algorithm can be amortized over the arc fetches. Our results show that the number of arc fetches grows in roughly the same rate as the running time in all tests, thus lending support to a comparison between the algorithms based on running times.

We also give the mean time per arc fetch (which is the mean running time divided by the mean number of arc fetches), in microseconds. It is important to obtain this measure,

because a big variation in it can be evidence that one implementation is better "crafted" than another (e.g. by being more efficient in its use of data structures), thus making any comparison potentially unfair. On the other hand small variations are to be expected, because different codes perform different operations the fetched data and because some codes have better locality properties than others. Our results show that this quantity varies somewhat from one problem to another and from one implementation to another, but never by much. For the largest sizes tested, the largest such variation happened in family rmfuC, where the extreme values are 0.51 (DFS) and 0.21 (AR), hence a factor of 2.4. (It's important to point out here that DFS is always the code with the largest or near-largest value for this measure, an indication of its relative lack of locality.) We believe this measure also gives confidence to the boundaries we used for defining which implementations are good and which aren't, as defined in the next paragraph.

We say that the performance of an implementation on a problem family is *good* if it is within a small constant factor (roughly 4) of the time of the fastest implementation; is *fair* if the performance is within a moderate constant factor (roughly a factor of 16) of the fastest; and *poor* of the implementation is slower by a larger constant factor or asymptotically slower. In Table 1 we present a comparison of all implementations based on these criteria. Next we discuss performance of individual implementations.

Table 1: Summary of results. *Blank* is good, ◦ is fair, and • is poor.

	DFS	BFS	LDS	AR	FIFO	LO	HI
fewg	•	◦					
manyg	•	◦					
hi-lo							•
grid	•						
hexa	•	◦					
rope							◦
zipf					◦		
karz		•	◦	•	•	•	•
rmfuC	•	•	◦	◦			
rmfuL	◦	◦		◦			
rmfuW	•	•	◦	◦			
blow			◦	◦			◦
puff	◦	•		◦			◦
saus					•	•	•
squa							◦
wave	•	•		◦			◦

The depth-first search implementation: This implementation is not robust. For flows, its performance varies from the best (e.g. on karz) to very bad (e.g. on wave). While for flow

problems depth-first search is more robust than breadth-first search, for matching depth-first search is the worst overall. For matchings, the performance is poor except for zipf, rope, and hi-lo (see Tables 8 and 4 and Figures 7 and 3), where it is good.

The breadth-first search implementation: This implementation is far less robust than the best push-relabel implementations. Breadth-first search is one of the worst codes for the flow problems; it performs poorly on half of our flow families. For the matching problems, the performance is good except for the hexa, fewg, and manyg problems, where it is fair (see Tables 2 and 3 and Figures 1 and 2).

The label-directed depth-search implementation: This implementation succeeds in making depth-first search more robust. In fact, it is the most robust augmentation-based implementation, but not as robust as the best push-relabel codes. For flows, performance of the label-directed search is good or fair, but never poor. For matchings, the performance is always good.

The augment-relabel implementation: The augment-relabel implementation is consistently worse than the best push-relabel implementations. For flows, the only exception is saus. Furthermore, this implementation is less robust than label-directed search. For matchings, the performance is good, but never the best.

The first-in, first-out and lowest-level implementations: These implementations perform very similarly and we discuss them together. Their performance is good for most problem families. For flows, the only exceptions are saus, where the performance is fair, and karz, where the performance is poor. Recall that the maximum flow value is 3 for all saus problems and the karz family was designed to be hard for push-relabel algorithms. For matchings, the only exception is zipf (see Table 8 and Figure 7), where the codes perform on the border between good and fair, whith FIFO on the fair side.

The highest-level implementation: This push-relabel implementation doesn't work nearly as well as the other two. For flows, its performance is fair except for the rmfu problem families, where the performance is good, and karz and blow families, where the performance is poor. For the matching problems, the performance is good except for hi-lo (see Table 4 and Figure 3), where the performance is poor, and rope where the performance is fair.

6 Discussion

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

When the number of augmentations is moderate or large, average time to find a path becomes critical. Results of [13] imply that for shortest path augmentation, the total length of augmenting paths is relatively small, especially in the bipartite matching case. If the augmenting paths are always short, breadth-first search tends to examine a small part of the graph for each search and is thus efficient. With a substantial number of long paths, depth-first search is more efficient than breadth-first search. In bipartite matching problems, the augmenting paths tend to be short and breadth-first search outperforms depth-first search. In the flow problems, the paths tend to be longer, and overall depth-first search performs better than breadth-first search.

According to Operations Research folklore, depth-first search is the best way to solve bipartite matching problems; in particular, it is at least as good as breadth-first search. In our tests, depth-first search was the worst algorithm for bipartite matching. Our breadth-first search implementation, however, differs from the standard implementation (see Section 3), and performs better as a result. Furthermore, the folklore has been formed before the push-relabel algorithms were introduced. Thus, our results do not contradict previous work.

Distance labels maintain information from one shortest path computation to another. If the labels are close to actual distances, the relabel-based algorithms work well. Because labels are lower bounds on the actual distances, small labels are always close to actual distances. While residual flow value is large, the lowest-level and the first-in, first-out algorithms work mostly with low-labeled vertices and thus work well. When residual flow value becomes small, the use of global relabeling assures relatively quick termination. (See [16] for a theoretical analysis of the lowest-level algorithm.) The highest-level and the augment-relabel algorithms do a significant fraction of their work on vertices with high distance labels and do not perform so well.

Another distinctive feature of the push-relabel method is as follows. The method can be viewed as searching for several augmenting paths simultaneously, using flow excesses to keep

track of the progress. A selection rule is used to choose where to continue the search next. This gives additional flexibility that is useful; for example, compare LO with AR (the latter algorithm conducts one search at a time).

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

7 Concluding Remarks

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

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

Although pushing is usually better than augmenting, this is not always the case. One clear case when augmenting path algorithms work well is when maximum flow or matching value is very small. In this case breadth-first and depth-first search implementations perform best. The label-directed search implementation also performs well when the above two implementations do, and is more robust. The karz problem family shows that the push-relabel codes can take a long time on problems that can be quickly solved by another code. These problems, however, are designed to be bad for the push-relabel codes. On a wide range of "natural" problems, the push-relabel codes perform well while augmenting path codes often perform poorly.

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

References

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

tions. Prentice-Hall, 1993.

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

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

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

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

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

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

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

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

[10] 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.

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

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

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

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

[15] A. V. Goldberg. *Efficient Graph Algorithms for Sequential and Parallel Computers*. PhD thesis, M.I.T., January 1987. (Also available as Technical Report TR-374, Lab. for Computer Science, M.I.T., 1987).

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

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

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

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

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

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

[22] 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 Networks with Special Structure and Some Applications.

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

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

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

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

[27] J. C. Setubal. New Experimental Results for Bipartite Matching. In *Proceedings of* NETFLOW93, pages 211–216. TR–21/93, Dipartimento di Informatica, Universitá di Pisa, 1993.

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

A Computational Results

This appendix gives the computational results we obtained for implementations for a wide variety of input families. For each input class and graph size we obtained five different random graphs. The times reported in this section are the average over the five instances. Programs were terminated if they did not complete in $1,800$ seconds of CPU time. In each table the number in bold is the mean running time of the implementation and the number beneath this is the standard deviation (sd) of the running times, expressed as a percentage of the running time. In addition, for each implementation we report the mean number of arc fetches per arc and the time (in microseconds) spent per arc fetch.

In the plots both of the axis are log scale.

Table 2: Results for family fewg

$\|V\|, \|A\|$	BIM_DFS	BIM_BFS	BIM_LDS	BIM_AR	BIM_FIFO	BIM_LO	BIM_HI	
32768	**2.43**	**0.53**	**0.24**	**0.23**	**0.25**	**0.17**	**0.16**	time
82333	16%	37%	46%	27%	36%	22%	25%	sd
	50.12	11.17	8.59	8.45	8.13	4.35	4.26	af/arc
	0.60	0.60	0.30	0.30	0.40	0.50	0.50	μsec/af
65536	**11.41**	**2.21**	**0.70**	**0.64**	**0.69**	**0.59**	**0.75**	
164341	9%	24%	28%	19%	28%	38%	35%	
	92.73	19.93	10.74	9.45	9.48	7.05	9.59	
	0.70	0.70	0.40	0.40	0.40	0.50	0.50	
131072	**47.85**	**6.18**	**1.38**	**1.38**	**1.42**	**1.39**	**1.41**	
328397	7%	19%	33%	24%	32%	48%	44%	
	159.06	22.59	9.42	8.73	8.31	7.71	8.05	
	0.90	0.80	0.40	0.50	0.50	0.50	0.50	
262144	**188.29**	**16.82**	**3.57**	**3.47**	**3.70**	**3.83**	**3.78**	
655882	5%	28%	24%	17%	21%	31%	31%	
	277.99	27.20	11.82	10.09	10.59	10.69	10.68	
	1.00	0.90	0.50	0.50	0.50	0.50	0.50	
524288	**722.92**	**53.51**	**7.20**	**7.16**	**7.13**	**7.29**	**6.99**	
1310910	1%	15%	21%	16%	27%	33%	37%	
	498.03	39.87	11.32	10.05	9.45	9.76	9.47	
	1.11	1.02	0.49	0.54	0.58	0.57	0.56	

Table 3: Results for family manyg

$\|V\|, \|A\|$	BIM_DFS	BIM_BFS	BIM_LDS	BIM_AR	BIM_FIFO	BIM_LO	BIM_HI	
32768	**1.47**	**0.47**	**0.25**	**0.30**	**0.26**	**0.25**	**0.22**	time
82370	15%	25%	38%	41%	24%	48%	43%	sd
	33.20	12.35	9.18	12.44	8.95	7.36	6.38	af/arc
	0.50	0.50	0.30	0.30	0.40	0.40	0.40	μsec/af
65536	**8.22**	**1.80**	**0.74**	**0.78**	**0.75**	**0.61**	**0.66**	
164193	6%	12%	17%	15%	9%	28%	29%	
	69.02	20.97	11.96	12.57	10.95	7.72	8.75	
	0.70	0.50	0.40	0.40	0.40	0.50	0.50	
131072	**35.28**	**4.41**	**1.26**	**1.61**	**1.37**	**1.29**	**1.25**	
328326	10%	45%	35%	33%	31%	49%	50%	
	120.75	21.15	9.12	11.28	8.33	7.47	7.43	
	0.90	0.60	0.40	0.40	0.50	0.50	0.50	
262144	**155.59**	**12.82**	**2.69**	**3.63**	**2.71**	**2.37**	**2.49**	
655720	4%	35%	30%	31%	27%	42%	40%	
	232.21	26.36	8.85	11.75	7.36	6.30	6.87	
	1.00	0.70	0.50	0.50	0.60	0.60	0.60	
524288	**639.89**	**44.30**	**6.90**	**9.33**	**7.07**	**6.83**	**6.73**	
1311296	4%	35%	31%	18%	27%	40%	40%	
	439.01	40.54	11.13	13.84	9.46	9.27	9.26	
	1.11	0.83	0.47	0.51	0.57	0.56	0.55	

Table 4: Results for family hi-lo

$\|V\|, \|A\|$	BIM_DFS	BIM_BFS	BIM_LDS	BIM_AR	BIM_FIFO	BIM_LO	BIM_HI	
32768	**0.47**	**0.46**	**0.86**	**1.04**	**0.79**	**0.84**	**5.56**	time
163795	8%	4%	6%	6%	16%	14%	3%	sd
	10.72	9.91	24.05	29.97	22.10	22.90	159.70	af/arc
	0.30	0.30	0.20	0.20	0.20	0.20	0.20	μsec/af
65536	**1.03**	**1.10**	**1.92**	**2.43**	**1.90**	**2.06**	**16.74**	
327635	12%	6%	20%	9%	10%	15%	4%	
	9.84	10.30	23.94	30.75	25.13	27.36	234.10	
	0.30	0.30	0.20	0.20	0.20	0.20	0.20	
131072	**2.26**	**2.36**	**3.74**	**5.95**	**4.42**	**4.70**	**46.71**	
655315	15%	8%	14%	6%	1%	8%	4%	
	9.72	10.51	21.83	33.82	27.64	30.71	320.20	
	0.40	0.30	0.30	0.30	0.20	0.20	0.20	
262144	**5.51**	**5.09**	**9.02**	**13.02**	**9.52**	**14.10**	**139.48**	
1310675	8%	7%	8%	7%	13%	9%	2%	
	11.67	10.91	25.77	35.35	27.93	47.42	464.25	
	0.40	0.40	0.30	0.30	0.30	0.20	0.20	
524288	**11.28**	**10.76**	**18.52**	**29.82**	**24.83**	**29.79**	**403.75**	
2621395	12%	4%	14%	6%	8%	13%	3%	
	11.37	11.22	25.51	38.42	35.16	49.66	649.73	
	0.38	0.37	0.28	0.30	0.27	0.23	0.24	

Table 5: Results for family grid

$\|V\|, \|A\|$	BIM_DFS	BIM_BFS	BIM_LDS	BIM_AR	BIM_FIFO	BIM_LO	BIM_HI	
32768	**0.80**	**0.31**	**0.15**	**0.15**	**0.12**	**0.13**	**0.13**	time
98304	7%	13%	7%	11%	7%	8%	5%	sd
	14.28	5.32	4.60	4.70	2.71	2.75	2.82	af/arc
	0.60	0.60	0.30	0.30	0.40	0.50	0.50	μsec/af
65536	**1.96**	**0.67**	**0.34**	**0.42**	**0.29**	**0.30**	**0.30**	
196608	7%	9%	2%	20%	6%	4%	8%	
	14.45	4.95	4.28	5.19	2.82	2.77	2.69	
	0.70	0.70	0.40	0.40	0.50	0.60	0.60	
131072	**3.33**	**1.49**	**0.73**	**1.01**	**0.62**	**0.63**	**0.60**	
393216	6%	2%	3%	12%	4%	2%	1%	
	12.31	6.13	4.30	5.88	2.69	2.79	2.72	
	0.70	0.60	0.40	0.40	0.60	0.60	0.60	
262144	**16.21**	**4.43**	**1.63**	**2.56**	**1.25**	**1.27**	**1.25**	
786432	4%	12%	3%	1%	2%	2%	4%	
	21.72	7.19	4.35	6.69	2.61	2.55	2.58	
	0.90	0.80	0.50	0.50	0.60	0.60	0.60	
524288	**162.80**	**12.50**	**4.03**	**6.07**	**3.70**	**3.39**	**3.28**	
1572864	2%	7%	2%	9%	2%	2%	3%	
	122.88	10.35	5.54	7.75	3.64	3.59	3.63	
	0.84	0.77	0.46	0.50	0.65	0.60	0.57	

Table 6: Results for family hexa

$\|V\|, \|A\|$	BIM_DFS	BIM_BFS	BIM_LDS	BIM_AR	BIM_FIFO	BIM_LO	BIM_HI	
32768	**2.29**	**0.72**	**0.27**	**0.21**	**0.19**	**0.19**	**0.24**	time
98304	17%	33%	40%	35%	28%	25%	55%	sd
	37.99	10.78	7.74	6.07	5.04	4.58	5.87	af/arc
	0.60	0.70	0.30	0.30	0.40	0.40	0.40	μsec/af
65536	**8.15**	**1.05**	**0.39**	**0.37**	**0.33**	**0.33**	**0.31**	
196608	14%	41%	14%	20%	22%	16%	14%	
	53.99	6.15	4.68	4.41	3.42	2.97	2.88	
	0.80	0.90	0.40	0.40	0.50	0.60	0.50	
131072	**33.85**	**3.19**	**1.09**	**0.83**	**0.76**	**0.76**	**0.75**	
393216	11%	67%	57%	21%	30%	28%	29%	
	95.41	8.59	6.25	4.20	3.32	3.33	3.32	
	0.90	0.90	0.40	0.50	0.60	0.60	0.60	
262144	**134.23**	**7.56**	**1.96**	**1.77**	**2.23**	**1.72**	**2.07**	
786432	10%	77%	28%	23%	64%	38%	73%	
	163.02	9.45	5.23	4.20	5.12	3.60	4.68	
	1.00	1.00	0.50	0.50	0.60	0.60	0.60	
524288	**596.01**	**37.71**	**6.42**	**6.09**	**6.14**	**4.27**	**4.95**	
1572864	5%	49%	45%	46%	56%	26%	56%	
	329.23	21.62	8.51	6.93	7.00	4.60	5.64	
	1.15	1.11	0.48	0.56	0.56	0.59	0.56	

Table 7: Results for family rope

$\|V\|, \|A\|$	BIM_DFS	BIM_BFS	BIM_LDS	BIM_AR	BIM_FIFO	BIM_LO	BIM_HI	
32768	**0.36**	**0.30**	**0.51**	**0.34**	**0.23**	**0.25**	**1.11**	time
98304	9%	7%	13%	4%	2%	4%	25%	sd
	10.37	8.39	18.15	11.75	7.29	7.25	37.07	af/arc
	0.40	0.40	0.30	0.30	0.30	0.40	0.30	μsec/af
65536	**0.92**	**0.77**	**1.22**	**0.92**	**0.53**	**0.56**	**2.64**	
196608	9%	4%	8%	20%	2%	1%	46%	
	12.06	9.46	20.11	13.44	7.17	7.17	41.48	
	0.40	0.40	0.30	0.30	0.40	0.40	0.30	
131072	**2.18**	**1.64**	**2.75**	**2.05**	**1.15**	**1.17**	**7.04**	
393216	12%	8%	32%	22%	2%	1%	24%	
	12.56	9.17	20.84	13.61	7.18	7.16	54.74	
	0.40	0.50	0.30	0.40	0.40	0.40	0.30	
262144	**4.79**	**3.45**	**5.82**	**4.70**	**2.41**	**2.42**	**20.65**	
786432	9%	5%	14%	21%	1%	1%	54%	
	12.63	9.25	21.15	14.32	7.19	7.16	74.91	
	0.50	0.50	0.30	0.40	0.40	0.40	0.40	
524288	**11.02**	**6.88**	**10.70**	**8.07**	**5.07**	**4.89**	**33.53**	
1572864	13%	3%	20%	10%	1%	0%	53%	
	13.76	8.91	18.42	12.34	7.19	7.16	58.73	
	0.51	0.49	0.37	0.42	0.45	0.43	0.36	

Table 8: Results for family zipf

$\|V\|, \|A\|$	BIM_DFS	BIM_BFS	BIM_LDS	BIM_AR	BIM_FIFO	BIM_LO	BIM_HI	
32768	**0.05**	**0.05**	**0.08**	**0.11**	**0.18**	**0.14**	**0.10**	time
98304	0%	0%	5%	5%	37%	0%	6%	sd
	1.65	1.29	2.96	4.14	5.23	2.83	1.99	af/arc
	0.30	0.40	0.30	0.30	0.40	0.50	0.50	μsec/af
65536	**0.12**	**0.11**	**0.19**	**0.28**	**0.45**	**0.32**	**0.20**	
196608	6%	4%	2%	2%	25%	3%	4%	
	1.60	1.26	2.91	4.14	5.52	2.78	1.90	
	0.40	0.50	0.30	0.30	0.40	0.60	0.50	
131072	**0.25**	**0.25**	**0.40**	**0.64**	**1.14**	**0.70**	**0.42**	
393216	0%	6%	2%	2%	20%	1%	2%	
	1.56	1.23	2.87	4.14	7.43	2.73	1.88	
	0.40	0.50	0.40	0.40	0.40	0.60	0.60	
262144	**0.53**	**0.52**	**0.84**	**1.37**	**2.42**	**1.45**	**0.85**	
786432	0%	1%	1%	1%	21%	1%	1%	
	1.52	1.20	2.82	4.12	8.48	2.69	1.83	
	0.40	0.50	0.40	0.40	0.40	0.70	0.60	
524288	**1.09**	**1.06**	**1.70**	**2.86**	**5.37**	**2.99**	**1.69**	
1572864	1%	1%	2%	1%	8%	0%	0%	
	1.48	1.17	2.77	4.12	9.65	2.66	1.76	
	0.47	0.57	0.39	0.44	0.35	0.72	0.61	

Figure 1: fewg

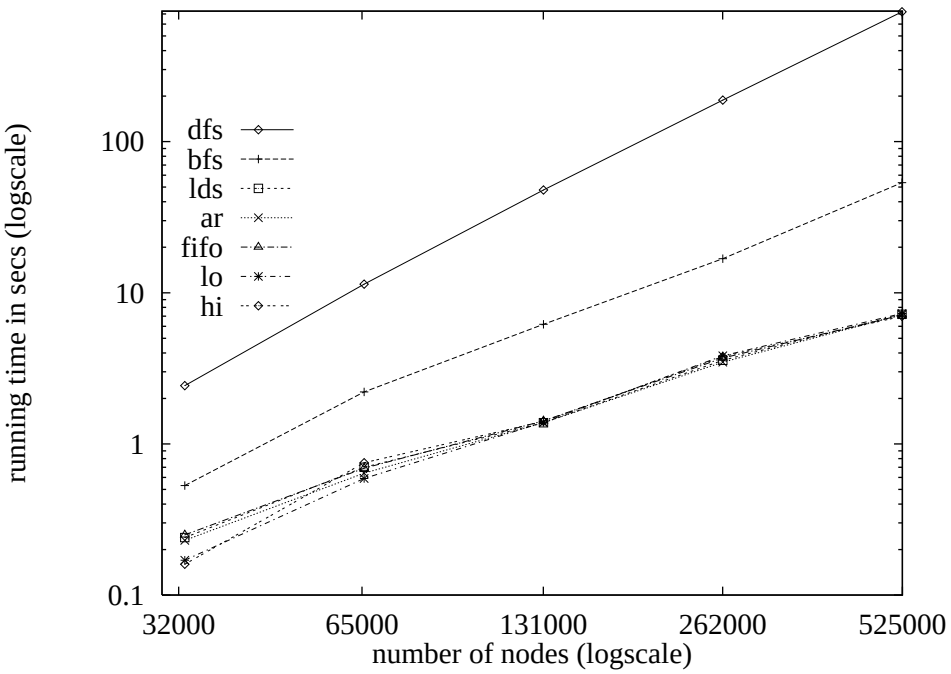

Figure 2: manyg

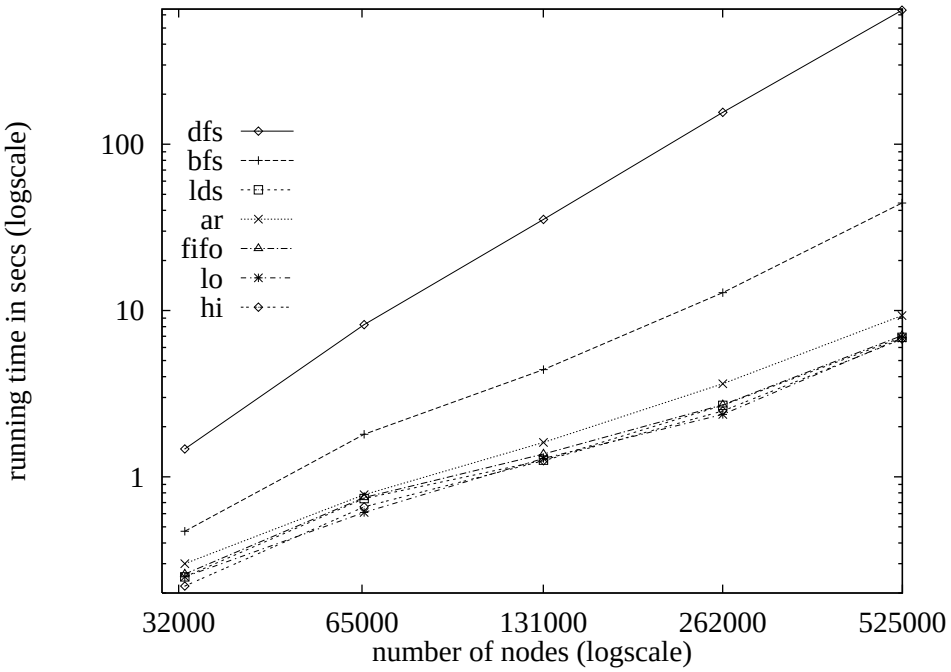

Figure 3: hi-lo

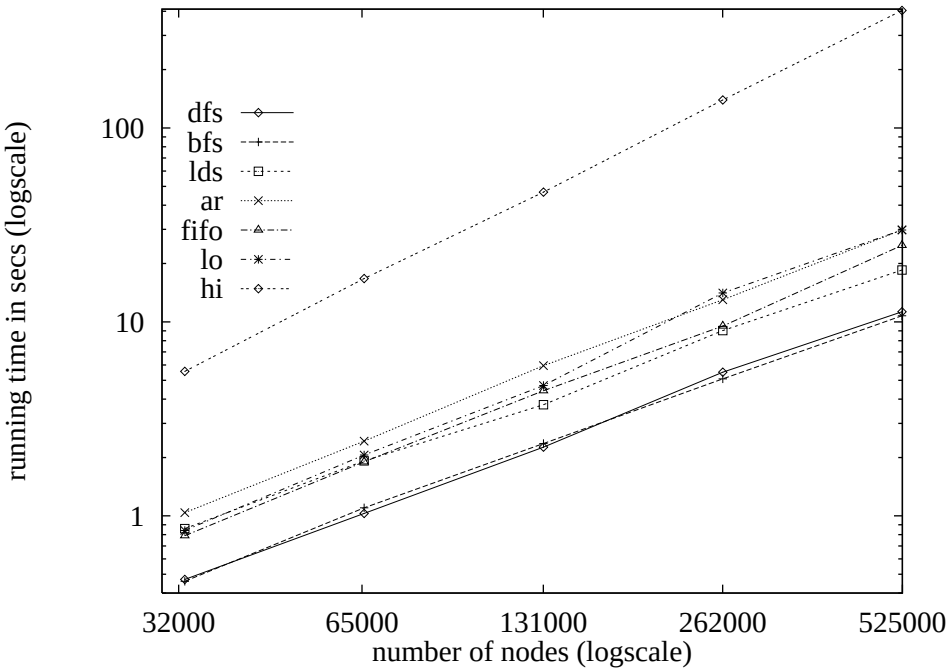

Figure 4: grid

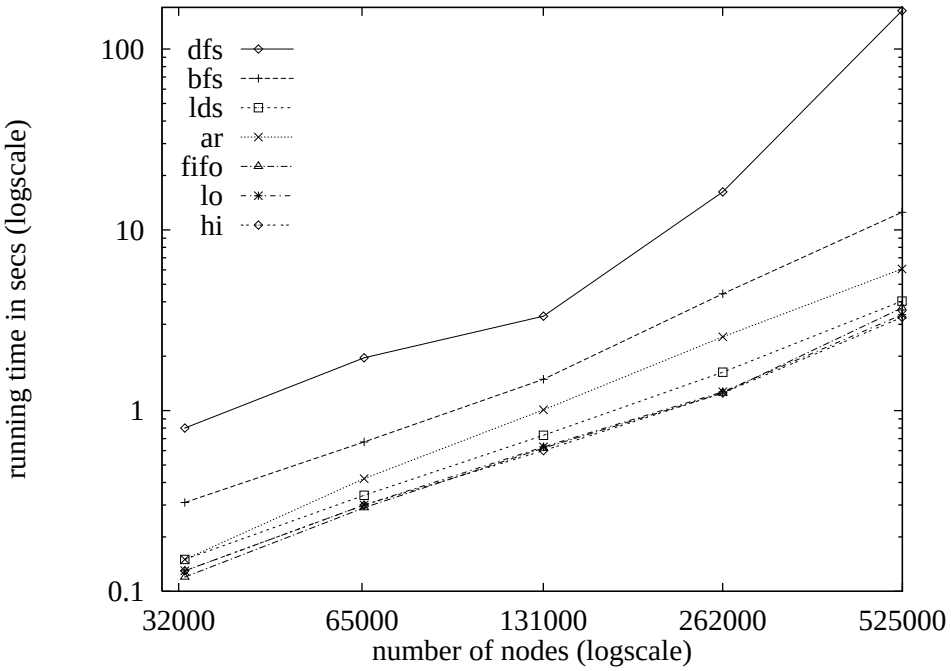

Figure 5: hexa

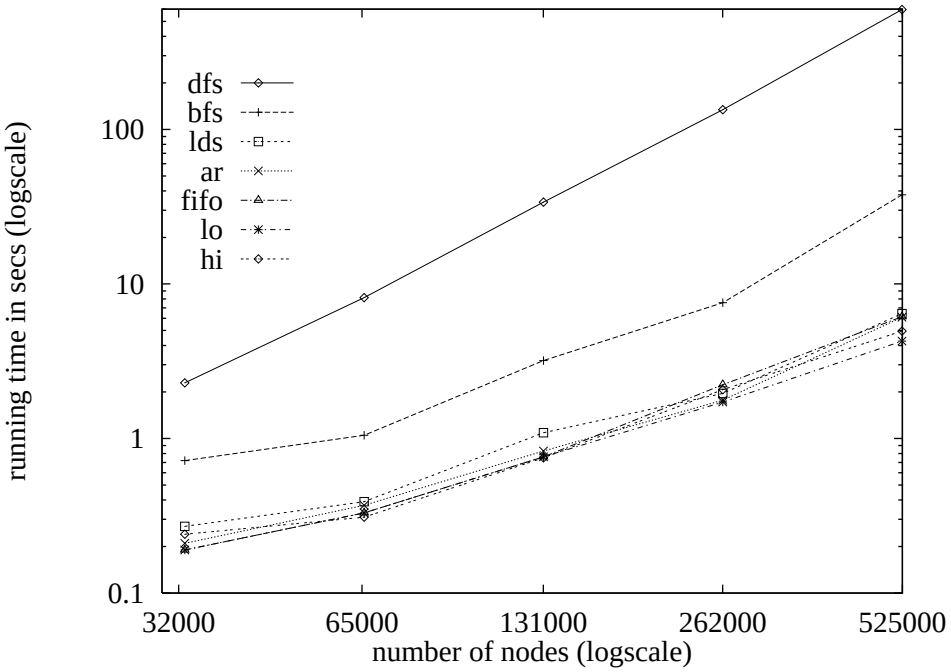

Figure 6: rope

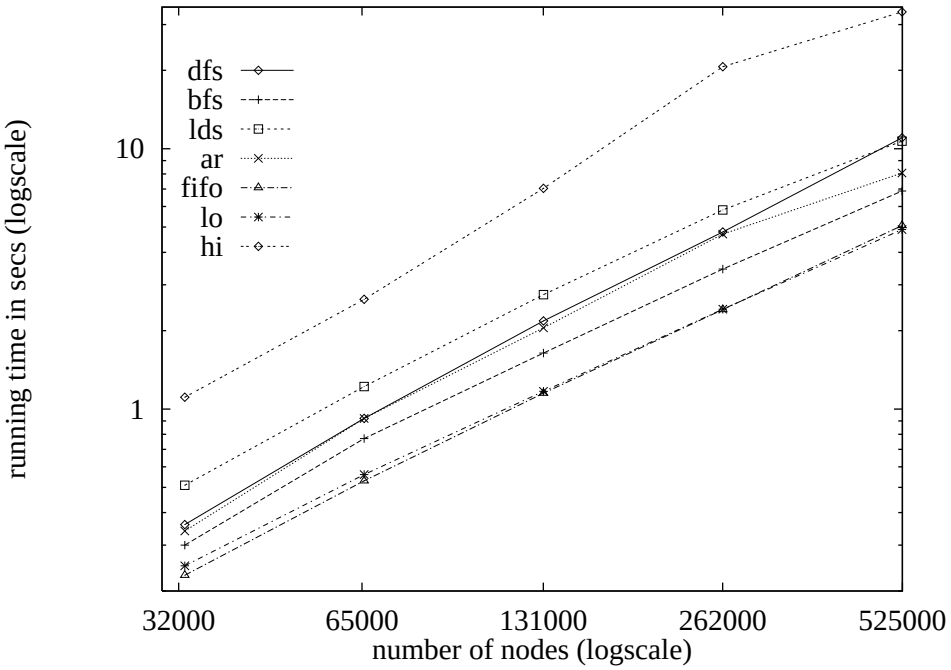

Figure 7: zipf

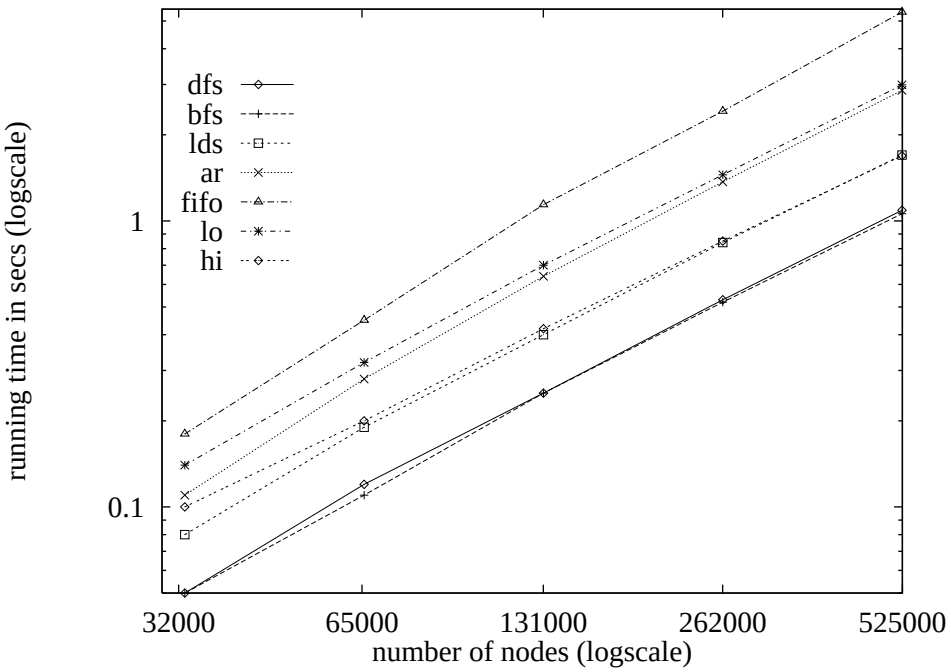

Table 9: Results for family karz

$\|V\|,\|A\|$	UF_DFS	UF_BFS	UF_LDS	UF_AR	UF_FIFO	UF_LO	UF_HI	
7074	**0.02**	**0.65**	**0.08**	**2.85**	**3.31**	**3.52**	**3.21**	time
64416	0%	1%	7%	2%	4%	0%	4%	sd
	1.87	56.14	12.84	349.67	364.38	372.63	340.24	af/arc
	0.20	0.20	0.10	0.10	0.10	0.10	0.10	μsec/af
14042	**0.05**	**2.31**	**0.32**	**10.64**	**10.49**	**13.07**	**11.10**	
166905	9%	0%	2%	1%	5%	2%	8%	
	1.91	79.57	17.43	506.00	446.51	535.39	454.95	
	0.20	0.20	0.10	0.10	0.10	0.10	0.10	
28482	**0.33**	**8.71**	**1.30**	**41.95**	**39.63**	**52.29**	**43.77**	
454464	6%	0%	1%	1%	7%	2%	3%	
	6.59	114.19	24.69	731.67	620.81	773.27	658.11	
	0.10	0.20	0.10	0.10	0.10	0.10	0.10	
56432	**1.15**	**32.50**	**4.72**	**157.06**	**114.22**	**205.65**	**126.23**	
1214730	6%	0%	1%	0%	5%	2%	5%	
	8.86	161.96	34.66	1002.70	666.44	1090.31	661.98	
	0.10	0.20	0.10	0.10	0.10	0.20	0.20	
114306	**0.95**	**128.29**	**19.47**	**668.82**	**502.59**	**878.31**	**606.56**	
3391104	2%	1%	2%	1%	5%	1%	7%	
	2.18	231.49	49.26	1460.11	984.56	1581.39	1049.16	
	0.13	0.16	0.12	0.14	0.15	0.16	0.17	

Table 10: Results for family rmfuC

$\|V\|, \|A\|$	UF_DFS	UF_BFS	UF_LDS	UF_AR	UF_FIFO	UF_LO	UF_HI	
15627	**4.07**	**7.17**	**1.07**	**2.26**	**0.36**	**0.40**	**0.51**	time
67362	1%	0%	11%	5%	4%	4%	5%	sd
	143.41	357.82	69.77	185.18	20.60	20.57	29.54	af/arc
	0.40	0.30	0.20	0.20	0.30	0.30	0.30	μsec/af
32770	**14.94**	**25.21**	**3.48**	**6.99**	**0.92**	**1.01**	**1.40**	
141203	1%	0%	4%	4%	5%	1%	4%	
	227.79	592.52	103.32	260.46	24.09	23.74	37.83	
	0.50	0.30	0.20	0.20	0.30	0.30	0.30	
64002	**47.24**	**78.47**	**9.91**	**20.03**	**2.11**	**2.34**	**3.63**	
275680	1%	1%	7%	4%	2%	1%	3%	
	348.57	930.77	145.49	364.31	26.86	27.25	47.84	
	0.50	0.30	0.20	0.20	0.30	0.30	0.30	
125002	**144.96**	**242.07**	**27.42**	**54.70**	**4.85**	**5.07**	**8.74**	
538250	1%	1%	9%	3%	7%	7%	4%	
	537.03	1459.55	203.51	501.39	31.36	29.73	57.75	
	0.50	0.30	0.30	0.20	0.30	0.30	0.30	
262146	**499.95**	**863.98**	**85.93**	**163.04**	**11.55**	**12.07**	**24.18**	
1128396	1%	1%	5%	2%	7%	5%	5%	
	866.97	2396.22	298.65	697.16	34.30	33.31	73.83	
	0.51	0.32	0.25	0.21	0.30	0.32	0.29	

Table 11: Results for family rmfuL

$\|V\|, \|A\|$	UF_DFS	UF_BFS	UF_LDS	UF_AR	UF_FIFO	UF_LO	UF_HI	
16302	**1.40**	**2.04**	**0.62**	**3.33**	**0.31**	**0.31**	**0.66**	time
73400	4%	3%	20%	5%	11%	8%	9%	sd
	45.79	96.00	37.28	242.80	15.74	13.80	36.04	af/arc
	0.40	0.30	0.20	0.20	0.30	0.30	0.30	μsec/af
32702	**2.98**	**4.14**	**1.48**	**7.47**	**0.67**	**0.62**	**1.52**	
147200	4%	3%	11%	3%	14%	9%	25%	
	45.34	95.55	43.93	268.82	16.90	12.82	41.18	
	0.40	0.30	0.20	0.20	0.30	0.30	0.30	
65502	**6.12**	**8.27**	**2.99**	**15.30**	**1.40**	**1.28**	**3.51**	
294800	4%	3%	21%	3%	14%	11%	32%	
	44.91	94.97	43.37	269.12	16.79	12.87	47.52	
	0.50	0.30	0.20	0.20	0.30	0.30	0.30	
131002	**12.40**	**16.56**	**6.80**	**31.18**	**2.45**	**2.44**	**8.25**	
589550	4%	3%	8%	3%	2%	10%	42%	
	44.54	94.61	49.03	270.25	14.67	12.12	56.29	
	0.50	0.30	0.20	0.20	0.30	0.30	0.20	
262102	**24.15**	**32.38**	**13.18**	**61.89**	**5.03**	**4.86**	**18.73**	
1179500	7%	5%	24%	5%	21%	11%	51%	
	43.24	92.60	46.87	264.16	14.76	11.85	63.85	
	0.47	0.30	0.24	0.20	0.29	0.35	0.25	

Table 12: Results for family rmfuW

$\|V\|, \|A\|$	UF_DFS	UF_BFS	UF_LDS	UF_AR	UF_FIFO	UF_LO	UF_HI	
16002	**8.50**	**17.41**	**1.55**	**1.87**	**0.46**	**0.53**	**0.49**	time
69280	2%	1%	10%	4%	7%	7%	6%	sd
	283.44	825.96	96.73	148.82	25.54	26.73	26.14	af/arc
	0.40	0.30	0.20	0.20	0.30	0.30	0.30	μsec/af
32492	**36.78**	**76.65**	**5.27**	**5.35**	**1.20**	**1.41**	**1.35**	
140674	1%	1%	6%	5%	7%	2%	7%	
	535.95	1647.74	150.03	196.80	29.70	32.49	32.58	
	0.50	0.30	0.20	0.20	0.30	0.30	0.30	
64002	**138.03**	**302.58**	**15.97**	**13.19**	**2.90**	**3.30**	**3.31**	
277120	2%	3%	7%	5%	9%	3%	9%	
	988.11	3173.14	228.67	242.03	35.45	38.13	40.08	
	0.50	0.30	0.30	0.20	0.30	0.30	0.30	
129962	**553.08**	**1308.81**	**52.69**	**34.66**	**7.52**	**7.68**	**8.29**	
562718	2%	2%	20%	24%	20%	15%	19%	
	1886.12	6228.77	355.38	301.58	42.42	42.53	47.06	
	0.50	0.40	0.30	0.20	0.30	0.30	0.30	
259212	N/A	N/A	**170.19**	**86.09**	**18.48**	**18.87**	**20.35**	
1122376	N/A	N/A	9%	9%	10%	6%	8%	
	N/A	N/A	559.52	371.86	50.00	51.96	57.16	
	N/A	N/A	0.27	0.21	0.33	0.32	0.32	

Table 13: Results for family blow

$\|V\|, \|A\|$	UF_DFS	UF_BFS	UF_LDS	UF_AR	UF_FIFO	UF_LO	UF_HI	
16384	**2.56**	**2.79**	**6.00**	**3.68**	**1.10**	**1.05**	**3.22**	time
163840	3%	2%	10%	3%	7%	10%	4%	sd
	58.89	77.95	214.47	167.61	36.40	31.91	111.97	af/arc
	0.30	0.20	0.20	0.10	0.20	0.20	0.20	μsec/af
32768	**6.08**	**6.05**	**14.75**	**9.62**	**2.45**	**2.23**	**9.34**	
327680	7%	7%	10%	6%	2%	8%	3%	
	61.63	79.13	232.45	202.15	36.25	31.56	151.87	
	0.30	0.20	0.20	0.10	0.20	0.20	0.20	
65536	**13.46**	**12.89**	**32.60**	**21.68**	**5.25**	**4.53**	**28.09**	
655360	10%	6%	16%	2%	3%	3%	3%	
	63.26	81.69	235.32	222.04	36.88	30.55	211.69	
	0.30	0.20	0.20	0.10	0.20	0.20	0.20	
131072	**27.33**	**26.64**	**74.36**	**49.16**	**11.67**	**10.33**	**82.58**	
1310720	9%	5%	13%	3%	5%	9%	1%	
	61.68	80.88	251.33	245.48	39.44	34.12	296.67	
	0.30	0.30	0.20	0.20	0.20	0.20	0.20	
262144	**60.15**	**56.32**	**165.02**	**115.61**	**24.20**	**20.57**	**242.21**	
2621440	3%	3%	10%	2%	7%	17%	1%	
	65.67	83.44	263.96	276.66	39.78	33.25	411.23	
	0.35	0.26	0.24	0.16	0.23	0.24	0.22	

Table 14: Results for family puff

$\|V\|, \|A\|$	UF_DFS	UF_BFS	UF_LDS	UF_AR	UF_FIFO	UF_LO	UF_HI	
16321	**3.42**	**5.96**	**0.72**	**3.35**	**0.78**	**0.76**	**2.95**	time
163840	1%	3%	9%	2%	9%	2%	3%	sd
	98.56	166.16	25.67	139.24	26.21	22.09	109.63	af/arc
	0.20	0.20	0.20	0.10	0.20	0.20	0.20	μsec/af
32761	**10.48**	**17.35**	**1.80**	**7.61**	**1.89**	**1.79**	**8.96**	
327600	2%	1%	8%	3%	4%	4%	3%	
	145.10	236.05	29.72	147.91	27.68	24.52	156.76	
	0.20	0.20	0.20	0.20	0.20	0.20	0.20	
65409	**31.71**	**51.25**	**3.88**	**16.13**	**4.36**	**4.00**	**26.26**	
655360	1%	1%	1%	4%	3%	2%	2%	
	213.02	337.86	30.62	149.16	31.91	27.20	221.13	
	0.20	0.20	0.20	0.20	0.20	0.20	0.20	
131041	**93.34**	**148.97**	**8.86**	**35.85**	**9.56**	**8.77**	**78.52**	
1310582	1%	1%	6%	2%	1%	1%	1%	
	307.00	475.84	34.71	159.33	33.53	28.73	312.41	
	0.20	0.20	0.20	0.20	0.20	0.20	0.20	
261889	**279.02**	**439.78**	**18.66**	**83.17**	**20.84**	**18.23**	**233.68**	
2621440	1%	1%	0%	2%	1%	0%	1%	
	446.83	677.72	35.64	178.87	35.53	29.29	437.65	
	0.24	0.25	0.20	0.18	0.22	0.24	0.20	

Table 15: Results for family saus

$\|V\|, \|A\|$	UF_DFS	UF_BFS	UF_LDS	UF_AR	UF_FIFO	UF_LO	UF_HI	
16384	**0.12**	**0.11**	**0.22**	**0.22**	**0.63**	**0.65**	**0.95**	time
163840	21%	22%	26%	0%	8%	6%	47%	sd
	4.23	3.11	7.25	7.37	30.61	30.61	47.34	af/arc
	0.20	0.20	0.20	0.20	0.10	0.10	0.10	μsec/af
32768	**0.31**	**0.28**	**0.70**	**0.48**	**1.76**	**1.80**	**2.77**	
327680	15%	15%	26%	1%	5%	4%	50%	
	4.88	3.84	11.26	7.36	40.66	40.66	65.59	
	0.20	0.20	0.20	0.20	0.10	0.10	0.10	
65536	**0.57**	**0.51**	**1.08**	**0.99**	**4.68**	**4.79**	**6.15**	
655360	21%	18%	36%	0%	3%	3%	50%	
	4.44	3.33	8.45	7.36	53.62	53.62	70.59	
	0.20	0.20	0.20	0.20	0.10	0.10	0.10	
131072	**1.24**	**1.04**	**2.12**	**2.08**	**13.00**	**13.23**	**17.06**	
1310720	8%	16%	28%	1%	2%	2%	49%	
	4.62	3.31	7.78	7.35	74.03	74.03	97.16	
	0.20	0.20	0.20	0.20	0.10	0.10	0.10	
262144	**2.61**	**2.30**	**5.12**	**4.34**	**36.44**	**36.86**	**48.54**	
2621440	19%	18%	40%	1%	1%	1%	54%	
	4.68	3.52	9.45	7.38	102.09	102.09	135.61	
	0.21	0.25	0.21	0.22	0.14	0.14	0.14	

Table 16: Results for family squa

$\|V\|, \|A\|$	UF_DFS	UF_BFS	UF_LDS	UF_AR	UF_FIFO	UF_LO	UF_HI	
16384	**1.23**	**1.08**	**1.39**	**1.41**	**0.78**	**0.87**	**3.66**	time
163840	6%	2%	8%	1%	5%	2%	7%	sd
	41.45	43.35	53.66	76.03	36.58	40.70	199.87	af/arc
	0.20	0.20	0.20	0.10	0.10	0.10	0.10	μsec/af
32768	**3.11**	**2.86**	**3.07**	**4.17**	**1.94**	**2.17**	**10.26**	
327680	8%	6%	22%	18%	4%	2%	16%	
	52.27	56.85	57.50	111.51	45.32	50.83	275.61	
	0.20	0.20	0.20	0.10	0.10	0.10	0.10	
65536	**8.38**	**7.87**	**8.47**	**10.43**	**4.91**	**5.45**	**29.90**	
655360	12%	3%	12%	1%	3%	1%	13%	
	68.78	78.76	77.42	138.64	58.51	66.29	397.56	
	0.20	0.20	0.20	0.10	0.10	0.10	0.10	
131072	**20.75**	**21.43**	**22.80**	**28.70**	**12.84**	**14.12**	**91.02**	
1310720	4%	3%	21%	1%	3%	1%	11%	
	87.34	109.22	101.54	191.24	75.36	86.78	589.23	
	0.20	0.10	0.20	0.10	0.10	0.10	0.10	
262144	**54.52**	**57.73**	**59.18**	**79.12**	**34.53**	**37.99**	**245.15**	
2621440	4%	4%	20%	1%	1%	0%	16%	
	113.84	147.66	126.73	262.95	99.63	116.28	738.17	
	0.18	0.15	0.18	0.11	0.13	0.12	0.13	

Table 17: Results for family wave

$\|V\|, \|A\|$	UF_DFS	UF_BFS	UF_LDS	UF_AR	UF_FIFO	UF_LO	UF_HI	
16381	**24.41**	**38.86**	**1.33**	**3.57**	**1.07**	**1.02**	**3.33**	time
163836	2%	1%	10%	2%	5%	3%	3%	sd
	738.12	1079.08	47.31	149.88	34.52	29.91	119.98	af/arc
	0.20	0.20	0.20	0.10	0.20	0.20	0.20	μsec/af
32761	**104.03**	**164.17**	**3.06**	**8.46**	**2.58**	**2.22**	**9.86**	
327672	2%	2%	29%	4%	2%	4%	4%	
	1500.54	2155.96	51.64	164.88	38.22	31.10	167.27	
	0.20	0.20	0.20	0.20	0.20	0.20	0.20	
65530	**444.29**	**699.96**	**7.30**	**20.13**	**5.89**	**5.16**	**29.66**	
655353	1%	1%	22%	2%	5%	7%	1%	
	3049.70	4306.69	58.76	185.48	41.38	34.98	236.05	
	0.20	0.20	0.20	0.20	0.20	0.20	0.20	
131068	N/A	N/A	**14.83**	**48.71**	**13.66**	**11.96**	**87.97**	
1310715	N/A	N/A	36%	1%	7%	5%	4%	
	N/A	N/A	56.83	212.68	45.30	39.09	324.19	
	N/A	N/A	0.20	0.20	0.20	0.20	0.20	
262144	N/A	N/A	**33.48**	**118.03**	**27.51**	**27.65**	**266.49**	
2621439	N/A	N/A	14%	4%	4%	5%	2%	
	N/A	N/A	61.12	242.33	43.38	43.63	455.11	
	N/A	N/A	0.21	0.19	0.24	0.24	0.22	

Figure 8: karz

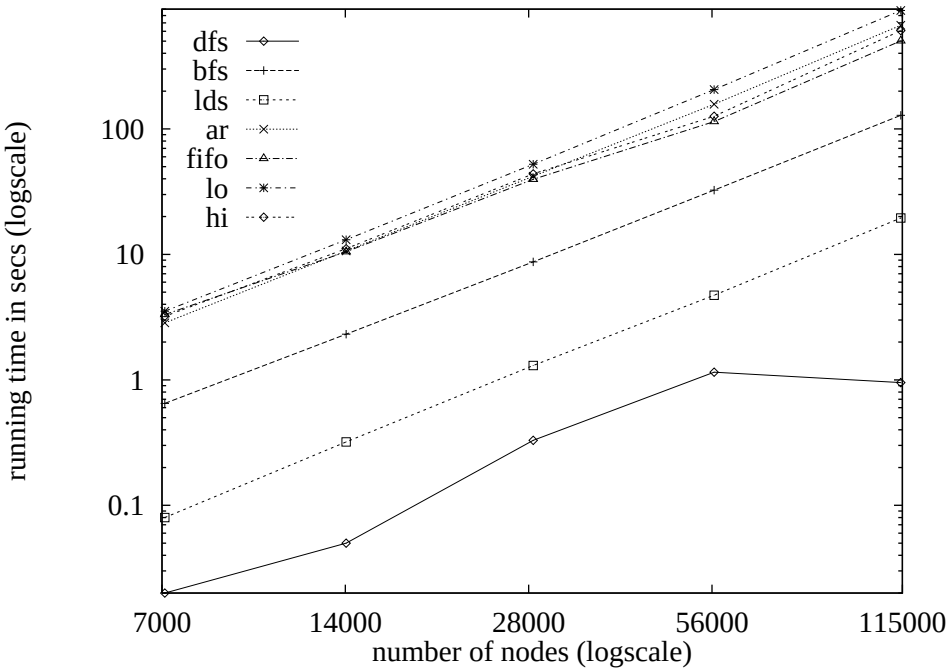

Figure 9: rmfuC

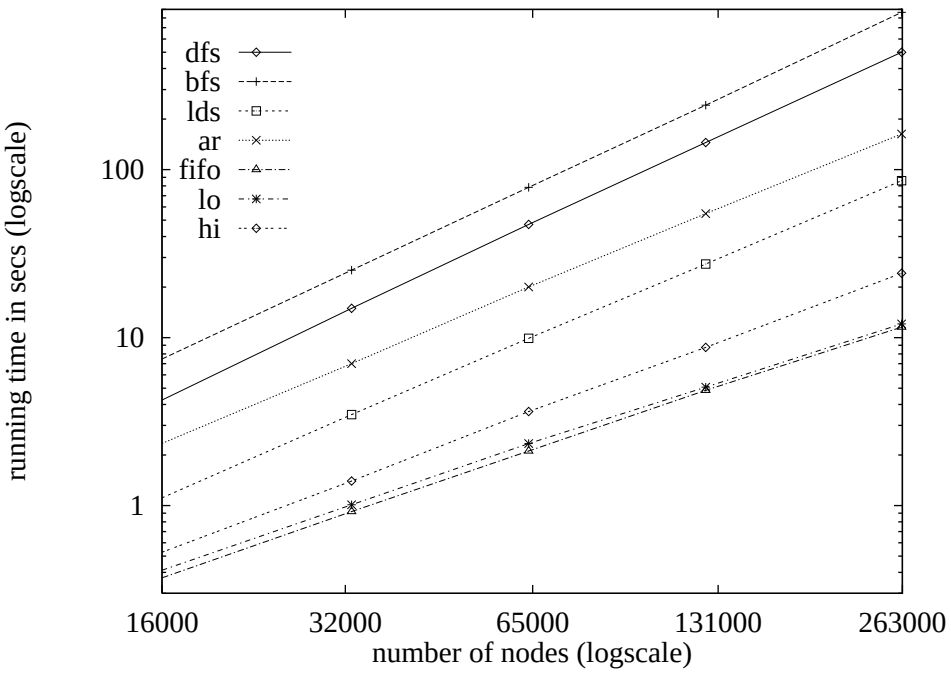

Figure 10: rmfuL

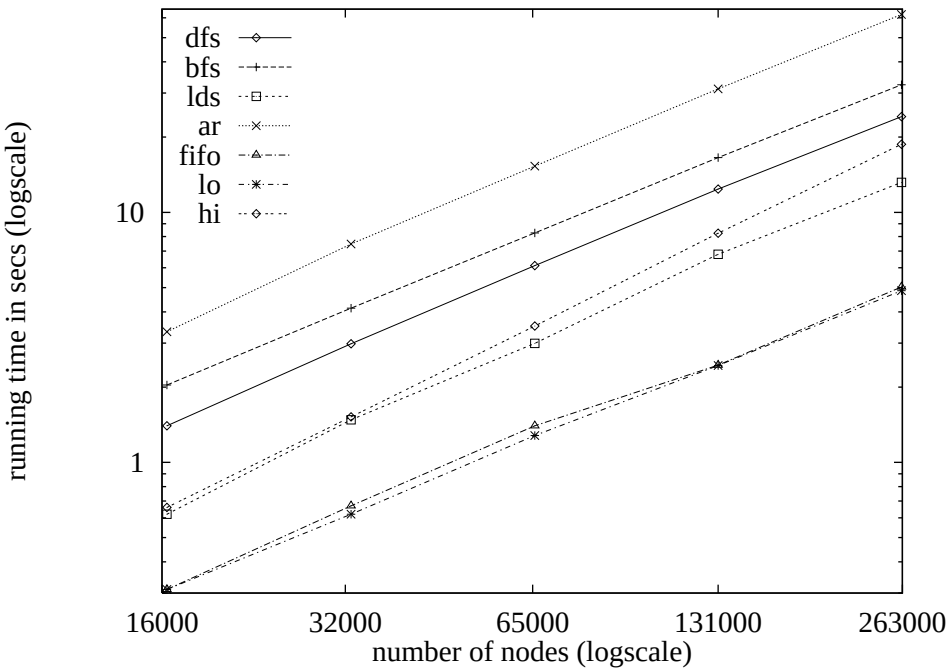

Figure 11: rmfuW

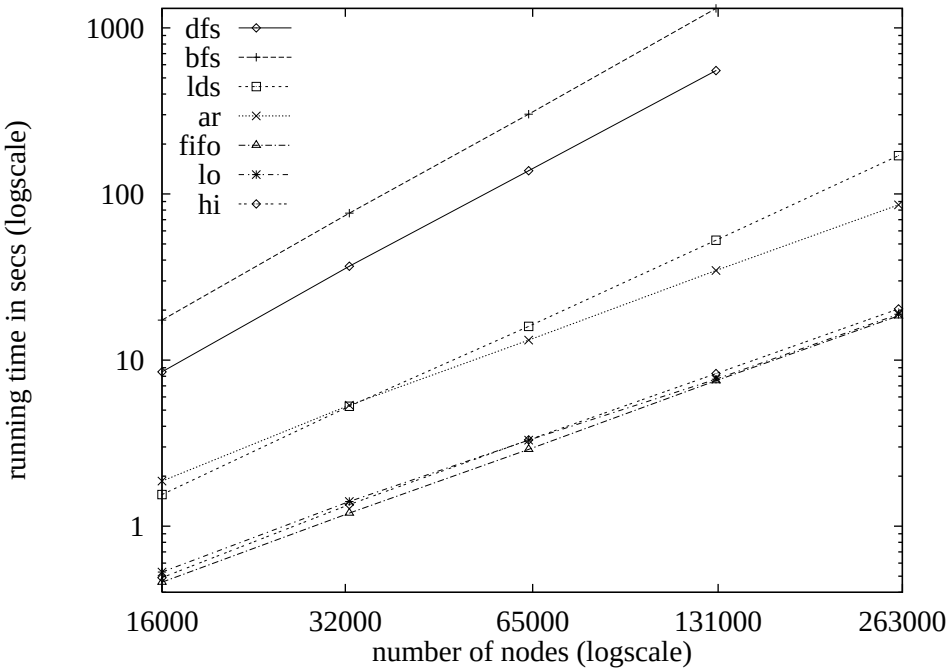

Figure 12: blow

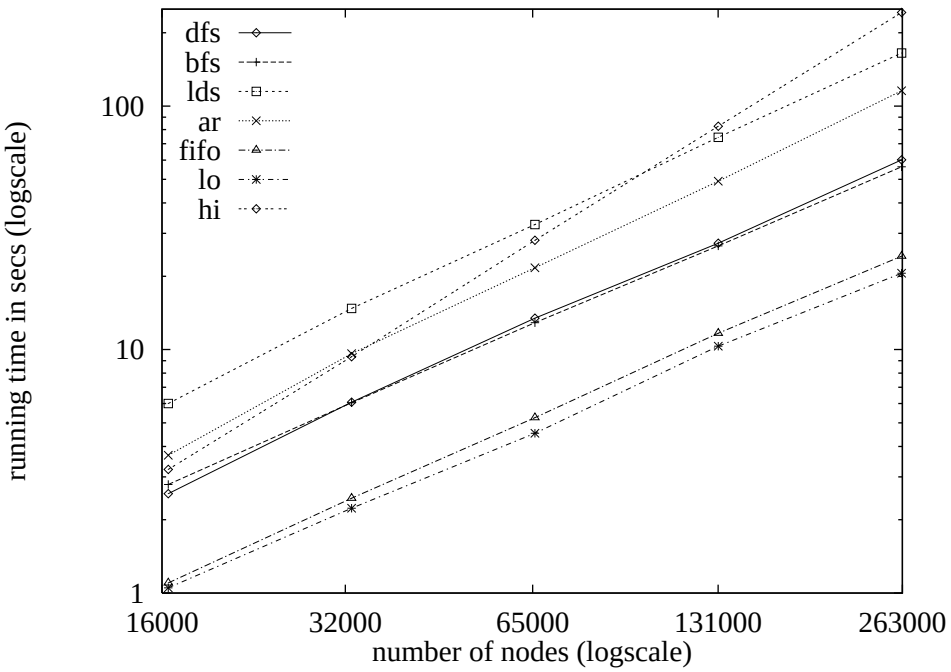

Figure 13: puff

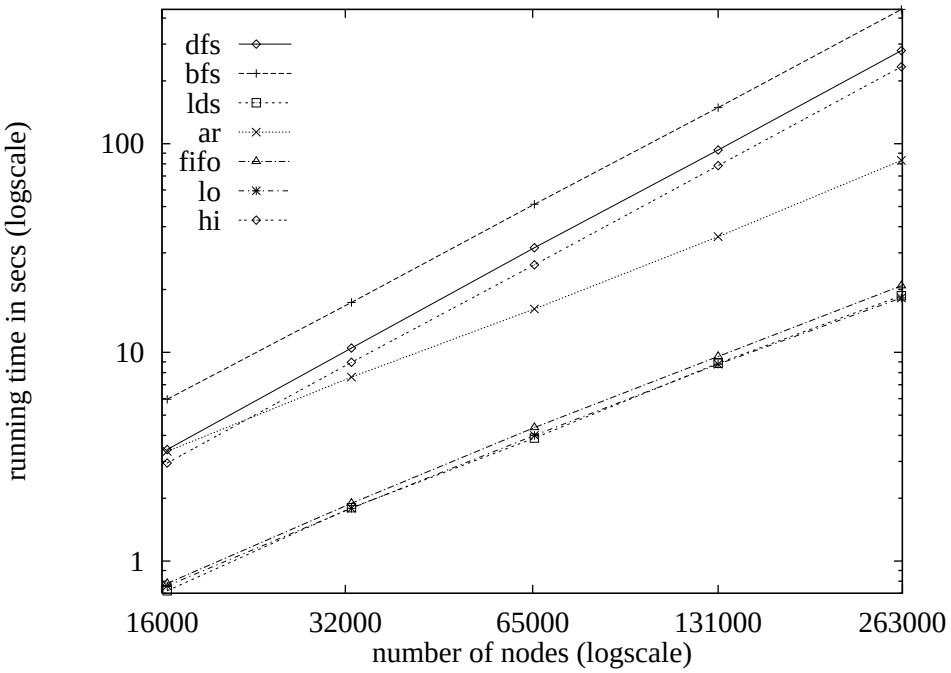

Figure 14: saus

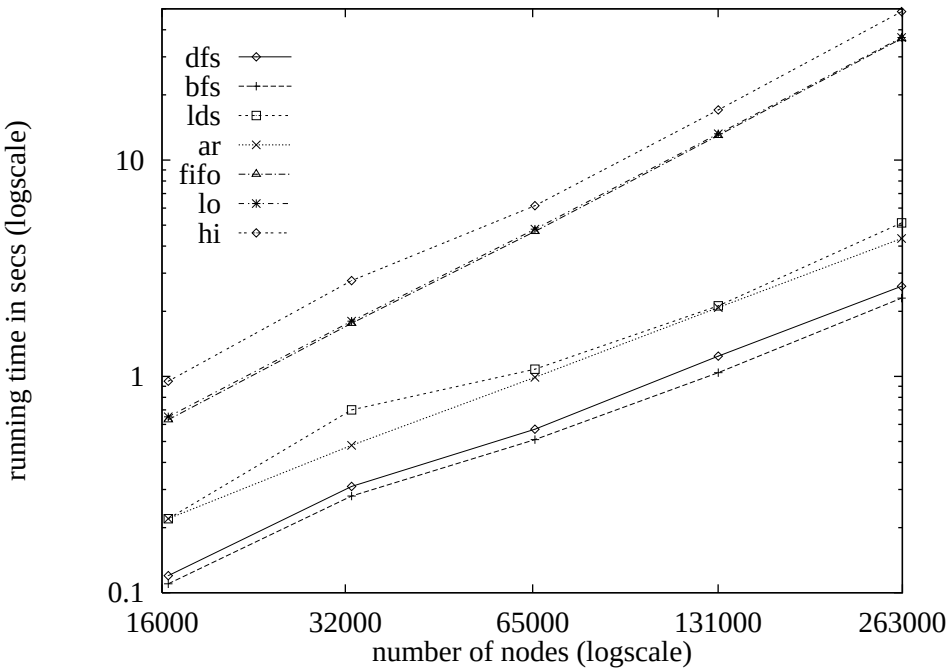

Figure 15: squa

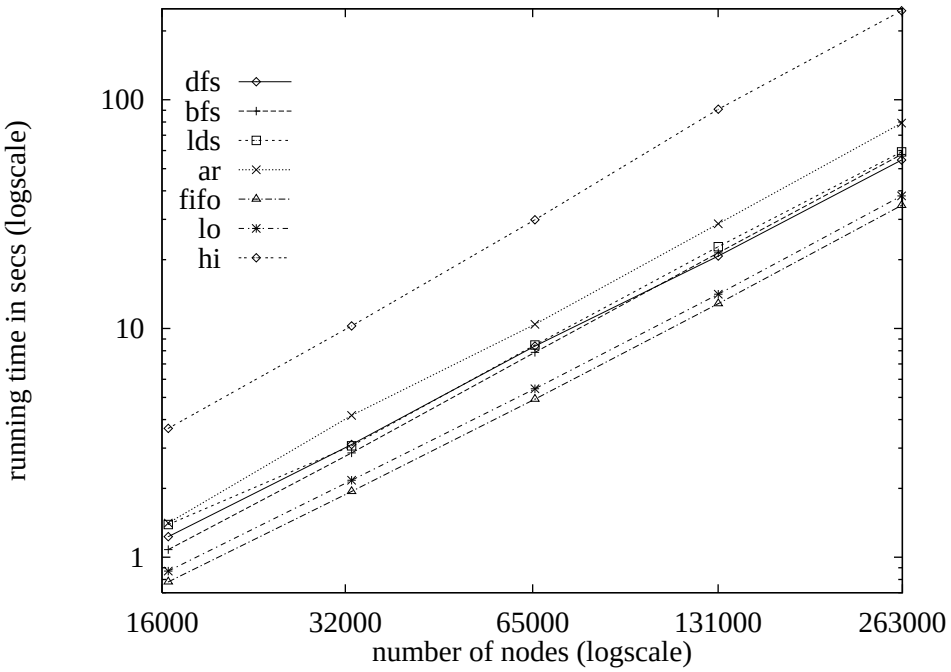
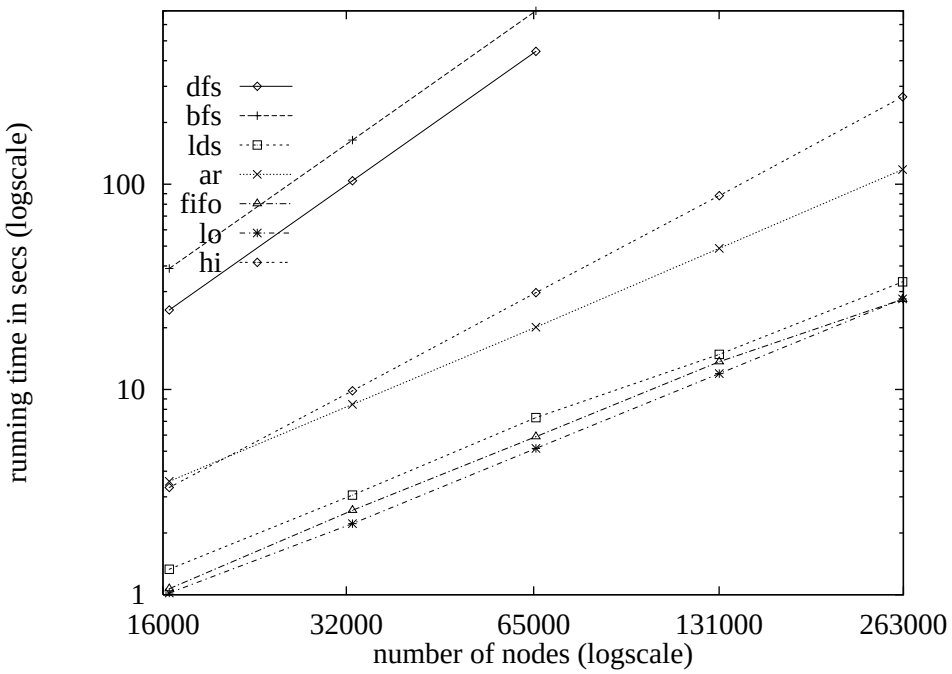
Figure 16: wave