Experimental Evaluation of a New Shortest Path Algorithm*

(Extended Abstract)

Seth Pettie, Vijaya Ramachandran, and Srinath Sridhar

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
`seth@cs.utexas.edu, vlr@cs.utexas.edu, srinath@cs.utexas.edu`

Abstract. We evaluate the practical efficiency of a new shortest path algorithm for undirected graphs which was developed by the first two authors. This algorithm works on the fundamental *comparison-addition model.*

Theoretically, this new algorithm out-performs Dijkstra's algorithm on sparse graphs for the all-pairs shortest path problem, and more generally, for the problem of computing single-source shortest paths from $\omega(1)$ different sources. Our extensive experimental analysis demonstrates that this is also the case in practice. We present results which show the new algorithm to run faster than Dijkstra's on a variety of sparse graphs when the number of vertices ranges from a few thousand to a few million, and when computing single-source shortest paths from as few as three different sources.

1 Introduction

The shortest paths problem on graphs is one of the most widely-studied combinatorial optimization problems. Given an edge-weighted graph, a path from a vertex u to a vertex v is a *shortest path* if its total length is minimum among all u-to-v paths. The complexity of finding shortest paths seems to depend upon how the problem is formulated and what kinds of assumptions we place on the graph, its edge-lengths and the machine model. Most shortest path algorithms for graphs can be well-categorized by the following choices.

1. Whether shortest paths are computed from a single *source* vertex to all other vertices (SSSP), or between all pairs of vertices (APSP). One should also consider the intermediate problem of computing shortest paths from *multiple* specified sources (MSSP).
2. Whether the edge lengths are non-negative or arbitrary.

* This work was supported by Texas Advanced Research Program Grant 003658-0029-1999 and NSF Grant CCR-9988160. Seth Pettie was also supported by an MCD Graduate Fellowship.

3. Whether the graph is directed or undirected.
4. Whether shortest paths are computed using just *comparison & addition* operations, or whether they are computed assuming a specific edge-length representation (typically integers in binary) and operations specific to that representation. Comparison-addition based algorithms are necessarily general and they work when edge-lengths are either integers or real numbers.

There is a wealth of literature on variations of the shortest path problem,[1] however despite such intense research, very few of the results beyond the classical algorithms of Dijkstra, Bellman-Ford, Floyd-Warshall, and min-plus matrix multiplication [AHU74,CLR90] work with real-valued edge-lengths using only comparisons and additions.[2]

Previous experimental studies of shortest path algorithms [CGR96,GS97,G01b] focussed on very restricted classes of inputs, where the edge lengths were assumed to be uniformly distributed, relatively small integers. This approach may be preferable for a specific application, however any algorithm implemented for more general use must be *robust*. By robust we mean that it makes no assumptions on the distribution of inputs, and minimal assumptions on the *programming interface* to the input (in the case of shortest path problems this leads naturally to the comparison-addition model); we elaborate on this in Section 2. A fact which many find startling is that Dijkstra's 1959 algorithm is still the best robust SSSP & APSP algorithm for positively-weighted sparse directed graphs.

In this paper we evaluate the performance of the recent *undirected* shortest path algorithm of Pettie & Ramachandran [PR02], henceforth the *PR algorithm*. The PR algorithm is a robust, comparison-addition based algorithm for solving undirected SSSP from multiple specified sources (MSSP). It works by pre-computing a certain structure called the 'component hierarchy', or *CH* (first proposed by Thorup [Tho99], for use with integer edge lengths) in time $O(m + n \log n)$. Once the CH is constructed SSSP is solved from any source in $O(m\alpha(m,n))$ time, where α is the very slow-growing inverse-Ackermann function. Theoretically this algorithm is asymptotically faster than Dijkstra's when the number of sources is $\omega(1)$ and the number of edges is $o(n \log n)$.

The PR algorithm (as well as [Tho99,Hag00]) can also tolerate a dynamic graph in some circumstances. If a component hierarchy is constructed for a graph G, SSSP can be solved in $O(m\alpha(m,n))$ time on any graph G' derived from G by altering each edge weight by up to a constant factor.

As mentioned above, there are only a few shortest path algorithms that work on the comparison-addition model, and there is only one robust algorithm in direct competition with PR, namely Dijkstra's. The Bellman-Ford, Floyd-Warshall, and min-plus matrix multiplication algorithms handle negative edge

[1] For an up-to-date survey of shortest path algorithms, see Zwick [Z01] (an updated version is available on-line).

[2] Some exceptions to this rule are Fredman's min-plus matrix multiplication algorithm [F76] and several algorithms with good *average-case* performance: [MT87,KKP93,KS98,Mey01,G01]

lengths and as a consequence are considerably less efficient than the PR algorithm (quadratic time for SSSP and cubic for APSP). The fastest implementation [Tak92] of Fredman's algorithm [F76] for APSP also takes almost cubic time. The average-case algorithms in [KKP93,McG91,Jak91,MT87] only provide improvements on very dense random graphs.

We evaluate the practical efficiency of the PR algorithm for the MSSP problem on undirected graphs by comparing it with Dijkstra's algorithm. The MSSP problem generalizes the SSSP-APSP extremes, and could be more relevant in some practical scenarios. For instance, a recent algorithm of Thorup [Tho01] for the graphic facility location and k-median problems performs SSSP computations from a polylog number of sources. Our experiments indicate quite convincingly that the Pettie-Ramachandran algorithm outperforms Dijkstra on sparse graphs when computing SSSP from a sufficient number of sources, as few as 3 or 4 in several cases. We obtained this result across all classes of sparse graphs that we considered except for the so-called 'long grids' [CGR96]. We also compare the PR algorithm to breadth first search, a natural lower bound on SSSP and a useful routine to normalize the running times of shortest path algorithms across different architectures. We elaborate on this and other aspects of our results in Section 6. Clearly, our results also apply to the APSP problem, and they show that the PR algorithm outperforms Dijkstra's algorithm for the APSP problem on sparse graphs.

The rest of the paper is organized as follows. In Section 2 we delineate the scope of our study. In Section 3 we give an overview of Dijkstra's algorithm and the PR algorithm. Section 4 describes the design choices we made in implementing the two algorithms. Section 5 describes our experimental set-up, and Section 5.1 the types of graphs we used. Section 6 provides our results. Section 7 ends with a discussion.

2 Scope of this Work

The focus of this paper is *robust* shortest path algorithms, so it is worthwhile to state here exactly what we mean by the term. A robust shortest path algorithm should be robust with respect to:

Input format. The algorithm should work with minimal assumptions on the input format and the programming "hooks" to manipulate the input. The assumption that edge-lengths are subject to comparison and addition operations is minimal since these operations are both necessary and sufficient to solve shortest path problem.

Graph type. The algorithm should work well on *all* graph sizes & topologies. It should not depend on the graph being overly structured (e.g. grids) or overly random (e.g. the $G_{n,m}$ distr.).

Edge-length distribution. The algorithm should not be adversely affected by the range or distribution on edge-lengths, nor should it depend upon the edge-lengths being chosen independently at random.

Some may object to the first criterion because, at some level, edge lengths are represented as `int`s or `double`s; one might as well assume such an input. This is not quite true. For instance, the LEDA platform [MN99] uses different types for rationals, high-precision floating point numbers, and 'real' numbers with provable accuracy guarantees, and Java has similar types BigDecimal and BigInteger. A robust algorithm can be used with all such types with little or no modification, and can be ported to different platforms with minimal modifications.

The bottom line is that robust algorithms are fit for use in a general setting where the format and distribution of inputs is unknown and/or varies. Nothing precludes the use of other specialized shortest path algorithms (indeed, those tailored to small integer weights, e.g. [GS97], will likely be faster), however, depending solely on such an algorithm is clearly unwise.

In our experiments we focus primarily on classes of *sparse graphs*, which we define as having an edge-to-vertex ratio less than $\log n$. Sparse graphs frequently arise naturally; e.g. all planar and grid-like graphs are sparse, and the evidence shows the 'web graph' also to be sparse. Denser graphs are important as well, but as a practical matter the SSSP problem has essentially been solved: Dijkstra's algorithm runs in linear time for densities greater than $\log n$. The "sorting bottleneck" in Dijkstra's algorithm is only apparent for sparse graphs.

3 Overview of the algorithms

Dijkstra's algorithm [Dij59] for SSSP (see [CLR90] or [AHU74]) visits the vertices in order of increasing distance from the source. It maintains a set S of visited vertices whose distance from the source has been established, and a tentative distance $D(v)$ to each unvisited vertex v. $D(v)$ is an upper bound on the actual distance to v, denoted $d(v)$; it is the length of the shortest path from the source to v in the subgraph induced by $S \cup \{v\}$. Dijkstra's algorithm repeatedly finds the unvisited vertex with minimum tentative distance, adds it to the set S and updates D-values appropriately.

Rather than giving a description of the Pettie-Ramachandran [PR02] algorithm (which is somewhat involved), we will instead describe the component hierarchy *approach* put forward by Thorup [Tho99]. Suppose that we are charged with finding all vertices within distance b of the source, that is, all v such that $d(v) \in [0, b)$. One method is to run Dijkstra's algorithm (which visits vertices in order of their d-value) until a vertex with d-value outside $[0, b)$ is visited. Thorup observed that if we choose $t < b$ and find the graph G_t consisting of edges shorter than t, the connected components of G_t, say $\mathcal{G}_t$, can be dealt with separately in the following sense. We can simulate which vertices Dijkstra's algorithm *would* visit for each connected component in $\mathcal{G}_t$, first over the interval $[0, t)$, then $[t, 2t)$, $[2t, 3t)$, up to $[\lfloor \frac{b}{t} \rfloor t, b)$. It is shown in [Tho99] (see also [PR02]) that these separate subproblems do not "interfere" with each other in a technical sense. The subproblems generated by Thorup's approach are solved recursively. The component hierarchy is a rooted tree which represents how the graph is decomposed; it is determined by the underlying graph and choices of t made in the algorithm. The basic procedure in component hierarchy-based algorithms

[Tho99,Hag00,PR02] is `Visit`(x, I), which takes a component hierarchy node x and an interval I, and visits all vertices in the subgraph corresponding to x whose d-values lie in I.

4 Design Choices

4.1 Dijkstra's Algorithm

We use a *pairing heap* [F+86] to implement the priority queue in Dijkstra's algorithm. We made this choice based on the results reported in [MS94] for minimum spanning tree (MST) algorithms. In that experiment the pairing heap was found to be superior to the Fibonacci heap (the choice for the theoretical bound), as well as d-ary heaps, relaxed heaps and splay heaps in implementations of the Prim-Dijkstra MST algorithm.[3] Since the Prim-Dijkstra MST algorithm has the same structure as Dijkstra's SSSP algorithm (Dijkstra presents both of these algorithms together in his classic paper [Dij59]), the pairing heap appears to be the right choice for this algorithm.

The experimental studies by Goldberg [CGR96,GS97,G01b] have used buckets to implement the heap in Dijkstra's algorithm. However, the bucketing strategy they used applies only to integer weights. The bucketing strategies in [Mey01,G01] could apply to arbitrary real edge weights, but they are specifically geared to good performance on edge-weights uniformly distributed in some interval. The method in [G01] can be shown to have bad performance on some natural inputs.[4] In contrast we are evaluating robust, general-purpose algorithms that function in the comparison-addition model.

We experimented with two versions of Dijkstra's algorithm, one which places all vertices on the heap initially with key value ∞ (the traditional method), and the other that keeps on the heap only vertices known to be at finite distance from the source. For sparse graphs one would expect the heap to contain fewer vertices if the second method is used, resulting in a better running time. This is validated by our experimental data. The second method out-performed the first one in all graphs that we tested, so we report results only for the second method.

4.2 Pettie-Ramachandran Algorithm

The primary consideration in [PR02] was asymptotic running time. In our implementation of this algorithm we make several simplifications and adjustments which are more practical but may deteriorate the worst-case asymptotic performance of the algorithm.

1. Finding MST:
   The [PR02] algorithm either assumes the MST is found in $O(m + n \log n)$ time (for the multi-source case) or, for the single source case, in optimal

[3] This algorithm was actually discovered much earlier by Jarník [Jar30].

[4] For instance, where each edge length is chosen independently from one of two uniform distributions with very different ranges.

time using the algorithm of [PR00]. Since, for multiple sources, we both find and sort the MST edges, we chose to use Kruskal's MST algorithm, which runs in $O(m \log n)$ time but does both of these tasks in one pass. Some of our data on larger and denser graphs suggests that it may be better to use the Prim-Dijkstra MST algorithm, which is empirically faster than Kruskal's [MS94], followed by a step to sort only the MST edges.

2. Updating D-values:
   In [PR02] the D-value of an internal CH node is defined to be the minimum D-value over its descendant leaves. As leaf D-values change, the internal D-values must be updated. Rather than use Gabow's near-linear time data structure [G85], which is rather complicated, we use the naïve method. Whenever a leaf's D-value decreases, the new D-value is propagated up the CH until an ancestor is reached with an even lower D-value. The worst-case time for updating a D-value is clearly the height of CH, which is $\log R$, where R is the ratio of the maximum to minimum edge-weight; on the other hand, very few ancestors need to be updated in practice.
3. Using Dijkstra on small subproblems:
   The stream-lined nature of Dijkstra's algorithm makes it the preferred choice for computing shortest paths on small graphs. For this reason we revert to Dijkstra's algorithm when the problem size becomes sufficiently small. If `Visit`(x, I) is called on a CH node x with fewer than ν descendant leaves, we run Dijkstra's algorithm over the interval I rather than calling `Visit` recursively. For *all* the experiments described later, we set $\nu = 50$.
4. Heaps vs. Lazy Bucketing:
   The [PR02] algorithm implements a priority queue with a *comparison-addition based* 'lazy bucketing' structure. This structure provides asymptotic guarantees, but for practical efficiency we decided to use a standard pairing heap to implement the priority queue, augmented with an operation called *threshold* which simulates emptying a bucket. A call to *threshold*(t) returns a list of all heap elements with keys less than t. It is implemented with a simple DFS of the pairing heap. An upper bound on the time for threshold to return k elements is $O(k \log n)$, though in practice it is much faster.
5. Additional Processing of CH:
   In [PR02, Sections 3 & 4] the CH undergoes a round of refinement, which is crucial to the asymptotic running time of the algorithm. We did not implement these refinements, believing their real-world benefits to be negligible. However, our experiments on *hierarchically structured graphs* (which, in effect, have pre-refined CHs) are very encouraging. They suggest that the refinement step could speed up the computation of shortest paths, at the cost of more pre-computation.

4.3 Breadth First Search

We compare the PR algorithm not only with Dijkstra's, but also with breadth first search (BFS), an effective lower bound on the SSSP problem. Our BFS routine is implemented in the usual way, with a FIFO queue [CLR90]. It finds a shortest path (in terms of number of edges) from the source to all other vertices, and computes the lengths of such paths.

5 Experimental Set-up

Our main experimental platform was a SunBlade with a 400 MHz clock and 2GB DRAM and a small cache (.5 MB). The large main memory allowed us to test graphs with millions of vertices. For comparison purposes we also ran our code on selected inputs on the following machines.

1. PC running Debian Linux with a 731 MHz Pentium III processor and 255 MB DRAM.
2. SUN Ultra 60 with a 400 MHz clock, 256 MB DRAM, and a 4 MB cache.
3. HP/UX J282 with 180 MHz clock, 128 MB ECC memory.

5.1 Graph Classes

We ran both algorithms on the following classes of graphs.

$\mathbf{G_{n,m}}$

The distribution $G_{n,m}$ assigns equal probability to all graphs with m edges on n labeled vertices (see [ER61,Bo85] for structural properties of $G_{n,m}$). We assign edge-lengths identically and independently, using either the uniform distribution over $[0,1)$, or the *log-uniform* distribution, where edge lengths are given the value 2^q, q being uniformly distributed over $[0,C)$ for some constant C. We use $C = 100$.

Geometric graphs

Here we generate n random points (the vertices) in the unit square and connect with edges those pairs within some specified distance. Edge-lengths correspond to the distance between points. We present results for distance $1.5/\sqrt{n}$, implying an average degree $\approx 9\pi/4$ which is about 7.

Very sparse graphs

These graphs are generated in two stages: we first generate a random spanning tree, to ensure connectedness, then generate an additional $n/10$ random edges. All edges-lengths are uniformly distributed.

Grid graphs

In many situations the graph topology is not random at all but highly predictable. We examine two classes of grid graphs: $\sqrt{n} \times \sqrt{n}$ square grids and $16 \times n/16$ long grids, both with uniformly distributed edge-lengths [CGR96].

New graph classes. Random graphs can have properties that might actually be improbable in real-world situations. For example, $G_{n,m}$ almost surely produces graphs with low diameter, nice expansion properties, and very few small, dense subgraphs [Bo85]. On the other hand, it may be that graph structure is less crucial to the performance of shortest path algorithms than edge length distribution. In the [PR02] algorithm for instance, all the random graph classes described above look almost identical when viewed through the prism of the component hierarchy. They generally produce short hierarchies, where nodes on the lower levels have just a few children and upper level nodes have vast numbers of children.

We introduce two classes of structured random graphs, *Hierarchical* and *Bullseye*, whose component hierarchies are almost predetermined. Edge lengths will be assigned randomly, though according to different distributions depending on how the edge fits into the overall structure.

Hierarchical graphs

These graphs are organized into a hierarchy of clusters, where the lowest level clusters are composed of vertices, and level i clusters are just composed of level $i-1$ clusters. A hierarchical graph is parameterized by the branching factor b and is constructed so that the CH is almost surely a full b-ary tree of height $\log_b n$. The graph density is also $O(\log_b n)$. We present results for $b = 6$ and $b = 10$.

Bullseye graphs

Bullseye graphs are parameterized by two numbers, the average degree d and the number of *orbits* o. Such a graph is generated by dividing the vertices into o groups of n/o vertices each (the orbits), and assigning $dn/2o$ random edges per orbit, plus a small number of inter-orbit edges to connect the orbits. Edge lengths are assigned depending on the orbits of the endpoints. An intra-orbit edge in orbit i, or an inter-orbit edge where i is the larger orbit is assigned a length uniformly from $[2^i, 2^{i+1})$. The resulting component hierarchy is almost surely a chain of o internal nodes, each with n/o leaf children. We present results for $o = 25$ and $o = 100$ with average degree $d = 3$.

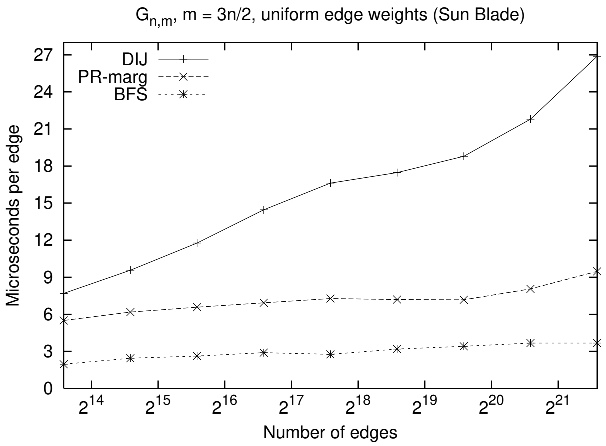
6 Results

The plots in Figures (a)-(j) give the running times of the two algorithms and BFS on the SunBlade for each of the graph classes we considered. Each point in the plots represents the time to compute SSSP/BFS, averaged over thirty trials from randomly chosen sources, on three randomly chosen graphs from the class. The y-axis is a measure of 'microseconds per edge', that is, the time to perform one SSSP/BFS computation divided by the number of edges.

In the plots, DIJ stands for the cost of computing SSSP using Dijkstra's algorithm and PR-marg stands for the *marginal cost* of computing SSSP using the Pettie-Ramachandran algorithm. By marginal cost for the PR algorithm, we mean *the time to compute SSSP after the CH is constructed, and excluding the cost of computing the CH.*

It is unclear how close a *robust* SSSP algorithm can get to the speed of BFS. Our results show that on a variety of graph types the marginal cost of the PR algorithm is *very* competitive with BFS, running between a factor of 1.87 and 2.77 times the BFS speed and usually less than 2.15 times BFS speed — see Table 1, third row. Naturally there is always room for some improvement; the question is, how much slack is there in the PR algorithm?

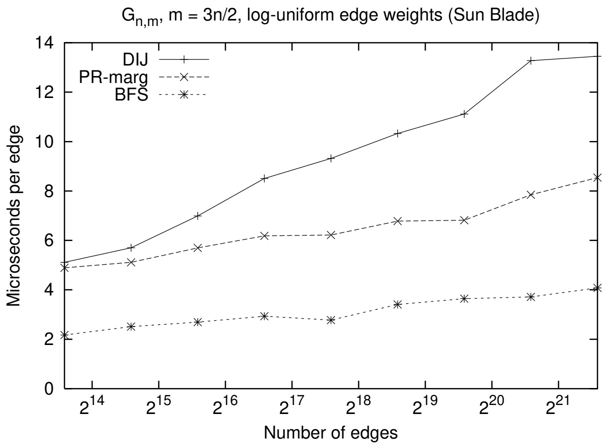
The effect of the CH pre-computation time is described in Table 1 and Figure (k). Table 1 (first row) lists, for each class of graphs, the critical number of



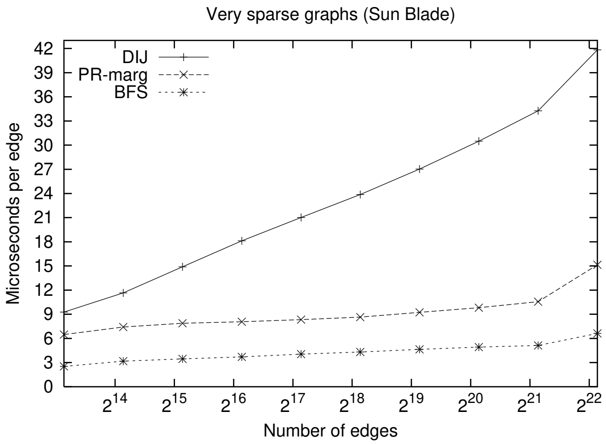
$G_{n,m}$, m = 3n/2, uniform edge weights (Sun Blade)
DIJ
PR-marg
BFS
Microseconds per edge
Number of edges
0
3
6
9
12
15
18
21
24
27
2^{14}
2^{15}
2^{16}
2^{17}
2^{18}
2^{19}
2^{20}
2^{21}

(a)

(b)



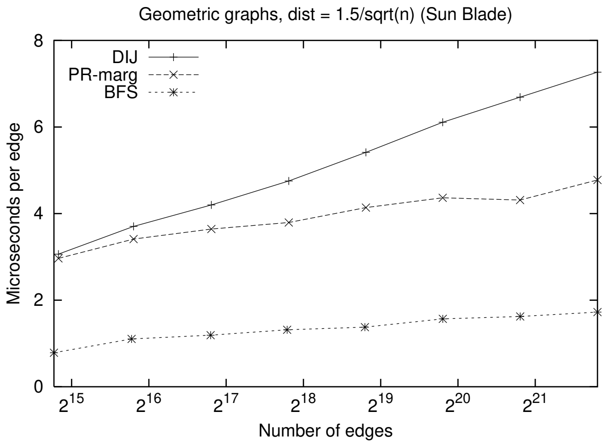
$G_{n,m}$, m = 3n/2, log-uniform edge weights (Sun Blade)
DIJ
PR-marg
BFS
Microseconds per edge
Number of edges
0
2
4
6
8
10
12
14
2^{14}
2^{15}
2^{16}
2^{17}
2^{18}
2^{19}
2^{20}
2^{21}



Very sparse graphs (Sun Blade)
DIJ
PR-marg
BFS
Microseconds per edge
Number of edges
0
3
6
9
12
15
18
21
24
27
30
33
36
39
42
2^{14}
2^{15}
2^{16}
2^{17}
2^{18}
2^{19}
2^{20}
2^{21}
2^{22}

(c)

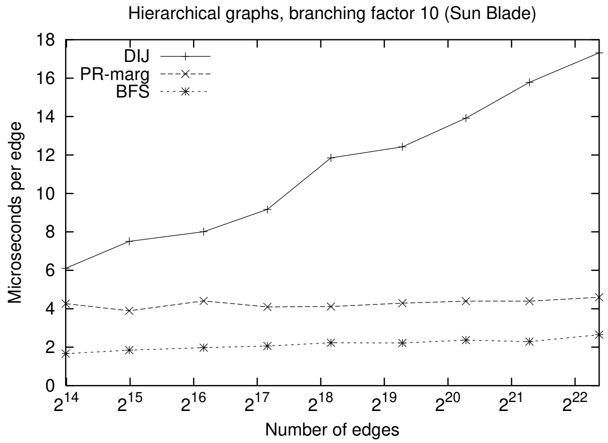
(d)



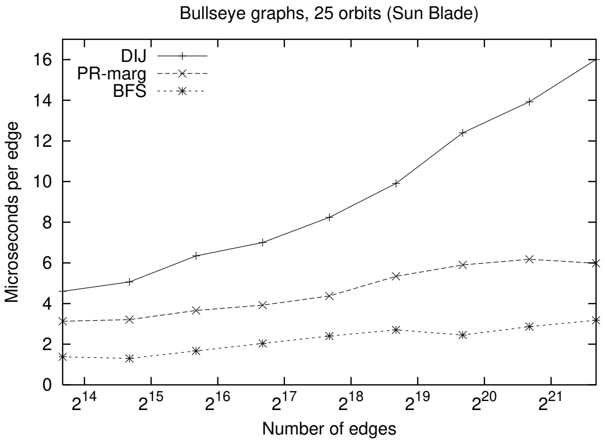
Geometric graphs, dist = 1.5/sqrt(n) (Sun Blade)
DIJ
PR-marg
BFS
Microseconds per edge
Number of edges
0
2
4
6
8
2^{15}
2^{16}
2^{17}
2^{18}
2^{19}
2^{20}
2^{21}


Hierarchical graphs, branching factor 6 (Sun Blade)
DIJ
PR-marg
BFS
Microseconds per edge
Number of edges

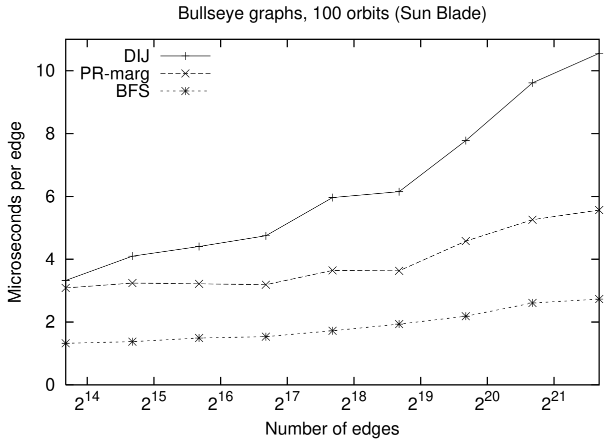
(e)


Hierarchical graphs, branching factor 10 (Sun Blade)
DIJ
PR-marg
BFS
Microseconds per edge
Number of edges

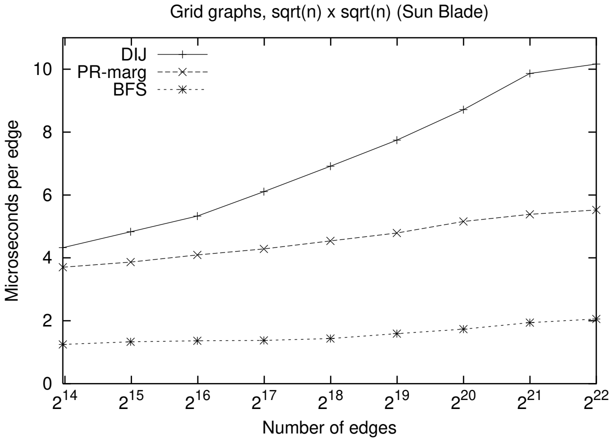
(f)


Bullseye graphs, 25 orbits (Sun Blade)
DIJ
PR-marg
BFS
Microseconds per edge
Number of edges

(g)

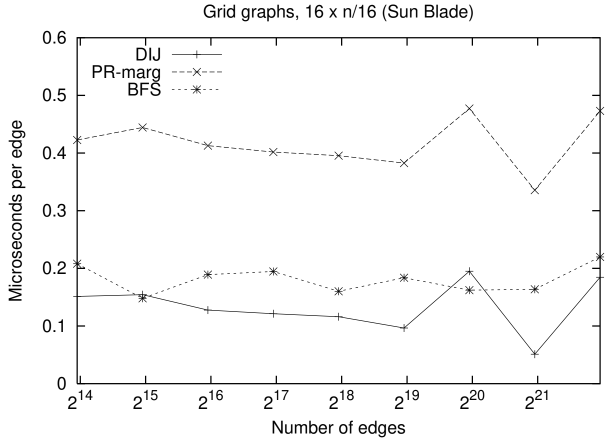

Bullseye graphs, 100 orbits (Sun Blade)
DIJ
PR-marg
BFS
Microseconds per edge
Number of edges

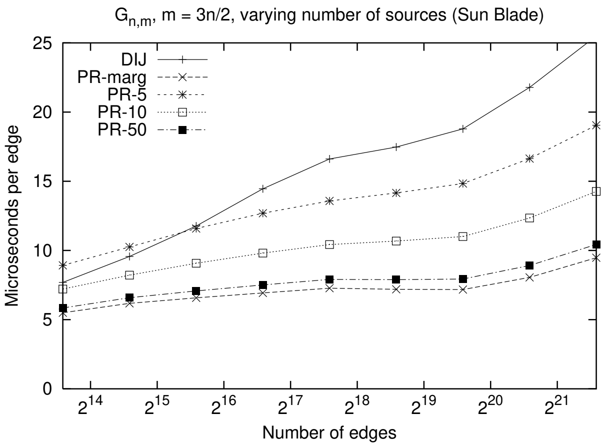
(h)


Grid graphs, sqrt(n) x sqrt(n) (Sun Blade)
DIJ
PR-marg
BFS
Microseconds per edge
Number of edges

(i)

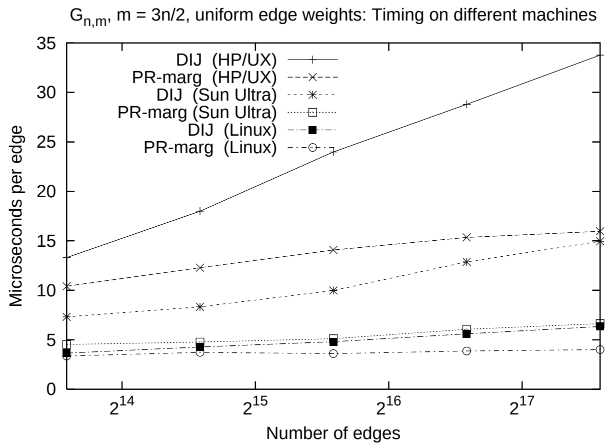
(j)

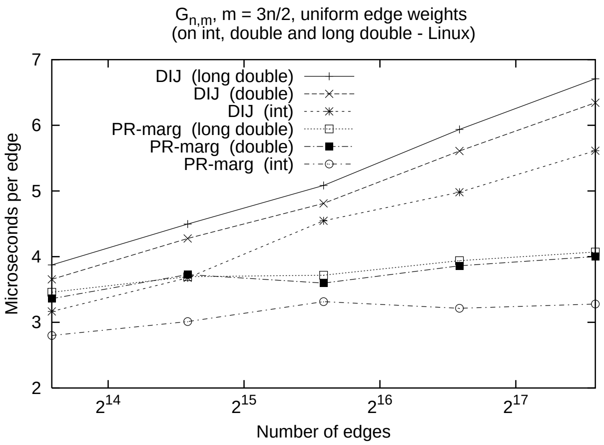

Grid graphs, 16 x n/16 (Sun Blade)
DIJ
PR-marg
BFS
Microseconds per edge
Number of edges


$G_{n,m}$, m = 3n/2, varying number of sources (Sun Blade)
DIJ
PR-marg
PR-5
PR-10
PR-50
Microseconds per edge
Number of edges

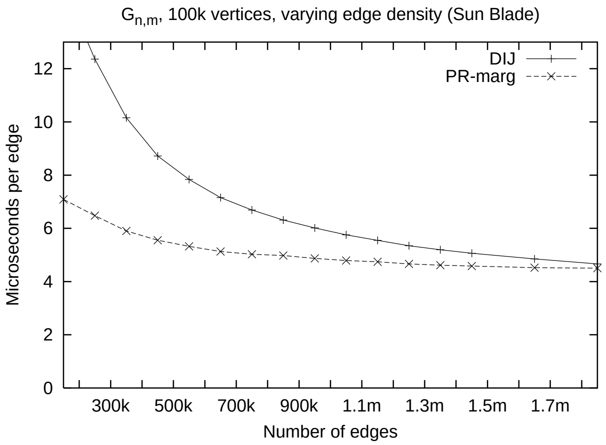
(k)

(l)

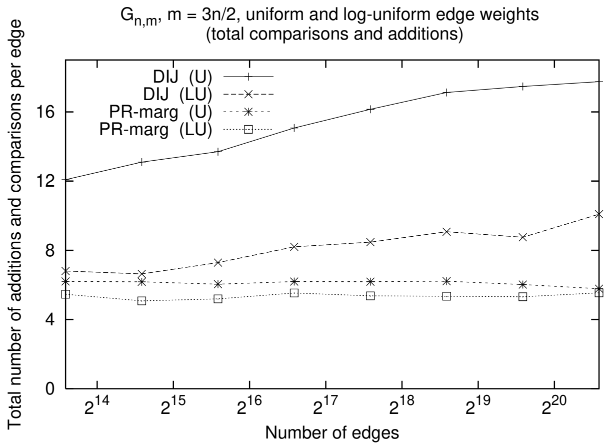

$G_{n,m}$, m = 3n/2, uniform edge weights: Timing on different machines
DIJ (HP/UX)
PR-marg (HP/UX)
DIJ (Sun Ultra)
PR-marg (Sun Ultra)
DIJ (Linux)
PR-marg (Linux)
Microseconds per edge
Number of edges


G_{n,m}, m = 3n/2, uniform edge weights
(on int, double and long double - Linux)
DIJ (long double)
DIJ (double)
DIJ (int)
PR-marg (long double)
PR-marg (double)
PR-marg (int)
Microseconds per edge
Number of edges
2
3
4
5
6
7
2^14
2^15
2^16
2^17

(m)


G_{n,m}, 100k vertices, varying edge density (Sun Blade)
DIJ
PR-marg
Microseconds per edge
Number of edges
0
2
4
6
8
10
12
300k
500k
700k
900k
1.1m
1.3m
1.5m
1.7m

(n)


G_{n,m}, m = 3n/2, uniform and log-uniform edge weights
(total comparisons and additions)
DIJ (U)
DIJ (LU)
PR-marg (U)
PR-marg (LU)
Total number of additions and comparisons per edge
Number of edges
0
4
8
12
16
2^14
2^15
2^16
2^17
2^18
2^19
2^20

(o)

sources s_0 such that PR (including the cost of computing CH) outperforms Dijkstra when SSSP is solved from at least s_0 sources on a graph with 2^{20} vertices. Figure (k) shows the amortized time per SSSP, *including* the cost of computing the CH, for varying numbers of sources in the $G_{n,m}$ graph class. Table 1 indicates that PR overtakes DIJ for a modest number of sources (on all but the long grid graph class), and Figure (k) indicates that the pre-computation time quickly becomes negligible as the number of sources increases. In Figure (k), the line PR-i represents the amortized cost per source (including pre-computation) when computing SSSP from i sources.

	$G_{n,m}$ (uniform)	$G_{n,m}$ (log-uni.)	Sparse	Geom.	Hier. $b=6$	Hier. $b=10$	Bull. $o=25$	Bull. $o=100$	Grid (sq.)	Grid (long)
s_0	3	21	3	17	4	4	4	7	10	n/a
CH/PR	5.05	11.75	4.38	8.48	10.1	9.6	5.33	6.11	7.85	82.66
PR/BFS	2.14	2.11	1.99	2.77	1.87	1.92	2.15	2.01	2.77	2.06

Table 1. *First line: number of SSSP computations (s_0) beyond which PR (including cost of computing CH) outperforms Dijkstra. Second line: ratio of time to construct the CH to the time for PR to perform one marginal SSSP computation. Third line: ratio of time for PR to perform one marginal SSSP computation to time for one BFS computation. These statistics reflect graphs of 2^{20} vertices.*

Figures (a) and (b) show the marginal performance of the PR algorithm to be stable over the uniform and log-uniform distributions. What is somewhat surprising is that Dijkstra's algorithm is dramatically faster under the log-uniform distribution (though still somewhat slower than the marginal performance of the PR algorithm for the same class). We hypothesize that this effect is due to the pairing heap. Recently, Iacono [Iac00] proved that the amortized complexity of `extract-min` in a pairing heap is logarithmic in the number of operations since the extracted item's insertion. Assigning edge lengths from the log-uniform distribution seems to cause Dijkstra's algorithm to visit vertices which were recently inserted into the heap. An interesting experiment would be to use a heap less sensitive to edge-length distribution, such as a binary heap. The plot for very sparse graphs in Figure (c) shows a nice separation between the marginal cost of the PR algorithm and the cost of Dijkstra's algorithm.

Figure (d), on geometric graphs, still shows the marginal cost of PR to be faster than Dijkstra on all graphs tested, though the separation in running times is not as dramatic as in $G_{n,m}$. We believe this is largely due to the density of the graph (the average degree for the graphs tested in Figure (e) is over 7) and the overhead for relaxing edges in PR, which is worse than in Dijkstra's algorithm. Another factor which could be taken into account is the large diameter of the geometric graphs, which is almost always $\Omega(\sqrt{n})$.

We believe the near-linear marginal costs of PR on Hierarchical graphs (Figures (e) and (f)) are a good indication of how well the *full* PR algorithm [PR02] could perform on all the graph classes. This is due to the structure of the component hierarchy. The CH which is derived naturally from a Hierarchical graph is very similar to the CH of *any* graph which is derived using the refinement step in [PR02, Sections 3 & 4]. The results for the Bullseye graphs are similar to those for $G_{n,m}$ for uniform and log-uniform distributions — DIJ performs better when the number of orbits increases.

The only graph class for which Dijkstra's algorithm beat the marginal cost of PR was the 'long grid' [CGR96], shown in Figure (j). This is to be expected. At any given moment in Dijkstra's algorithm the heap probably contains a constant number of elements, hence the worst-case $n \log n$ term in Dijkstra's algorithm never appears. The running times of the algorithms on long grids appear jumpy because of some aberrant delays which affect a small portion of the SSSP/BFS computations. In the case of square grids, Figure (i), Dijkstra's algorithm did exhibit a super-linear running time. The grid topology of the graph did not seem to have any unpredictable effect on either algorithm.

The results on the SUN Ultra 60, the Linux PC and the HP/UX machines are similar (see figure (l) for a comparison of runs on $G_{n,m}$ with uniform distribution of edge lengths), except that the runs are much faster on the Linux PC and much slower on the HP machine. The Linux PC was also much more sensitive to whether edge-lengths are integers, or floating points with double precision, or floating points with quadruple precision (see Figure (m)). In contrast the results on the SUN machines were virtually the same for integers and double-precision floating points. Note that we needed to alter just one line of code to move between ints, doubles, and long doubles. This is one advantage of the comparison-addition model.

Figure (n) shows the change in running time of both algorithms as the number of edges is increased for a fixed number of vertices. This study was performed on $G_{n,m}$ with 100,000 vertices and with uniform distribution of edge lengths. Dijkstra's algorithm seems to converge to a linear running time as the edge density increases. However, the figure shows the marginal cost of the PR algorithm to be slightly superior even for relatively large edge densities.

Finally in Figure (o) we plot the comparison-addition cost of our implementation of Dijkstra's algorithm and the comparison-addition marginal cost of our implementation of the PR algorithm. The plots are for $G_{n,m}$ with uniform and log-uniform edge length distribution. It is interesting to note that this cost appears to be practically linear for both types of graphs for PR while it is super-linear for DIJ. This plot once again shows up the significantly better performance of DIJ on log-uniform distribution over uniform distribution of edge lengths. These results are, of course, specific to the design choices we made for our implementations (in particular, the use of the pairing heap).

7 Discussion

We have implemented a simplified version of the Pettie-Ramachandran shortest path algorithm for undirected graphs and tested it against its chief competitor: Dijkstra's algorithm. The evidence shows that the pre-computation time of the PR algorithm is time well spent if we proceed to compute multiple-source shortest paths from enough sources.

We did not compare our algorithm directly to any integer-based shortest path algorithms, the focus of [CGR96,GS97,G01b], however we do compare it against breadth first search, a practical lower bound on the shortest path problem. In Goldberg's [G01b] recent study, the best algorithms performed (roughly) between 1.6 and 2.5 times the BFS speed,[5] whereas the PR algorithm performed 1.87 to 2.77 times slower than BFS, a remarkable fact considering the algorithms tested in [G01b] were specifically engineered for small integer edge lengths.

One issue we did not directly address is whether the PR algorithm's gain in speed is due to caching effects, or whether it is genuinely performing fewer operations than Dijkstra's algorithm. The data on comparison/addition operations[6] versus running time data suggests that the cache miss-rate is roughly equal in both Dijkstra's algorithm and the PR algorithm. We suspect that plugging in a cache-sensitive heap, such as [S00], will affect the performance of both algorithms similarly.

An open problem is to develop a shortest path algorithm for undirected graphs which beats Dijkstra's when computing *single*-source shortest paths on sparse graphs. We think the component hierarchy approach can lead to such an algorithm (and a qualified success appears in [PR02]). However, the possibility of a *practical* SSSP algorithm based on the component hierarchy is unlikely since it requires computing the MST in advance, and the experimental results in [MS94] suggest that the fastest method (in practice) for computing MST is the Prim-Dijkstra algorithm — which is structured nearly identically to Dijkstra's SSSP algorithm [Dij59].

It would be interesting to see if the performance of PR could be improved by using the hierarchical bucketing structure (using only comparisons and additions) assumed in [PR02] rather than the pairing heap used in our experiments. Very similar bucketing structures were used in two recent theoretical SSSP algorithms [Mey01,G01], both with good *average-case* performance. Both assume uniformly distributed edge lengths. An open question is whether either of these algorithms work well in practice (and if they are competitive with [PR02]), and how sensitive each is to edge-length distribution.

References

[AHU74] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[Bo85] B. Bollobás. *Random Graphs.* Academic Press, London, 1985.

[CGR96] B. V. Cherkassky, A. V. Goldberg, T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. In *Math. Prog.* 73 (1996), 129-174.

[CLR90] T. Cormen, C. Leiserson, R. Rivest. *Intro. to Algorithms.* MIT Press, 1990.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. In *Numer. Math.*, 1 (1959), 269-271.

[ER61] P. Erdös, A. Rényi On the evolution of random graphs. *Bull. Inst. Internat. Statist.* 38, pp. 343–347, 1961.

[5] Goldberg implements his BFS with *the same data structures* used in one of the algorithms, which, if slower than the usual BFS, would bias the timings.

[6] The number of comparisons/additions in PR and DIJ closely correspond to the total number of operations.

[F76] M. Fredman. New bounds on the complexity of the shortest path problem. *SIAM J. Comput.* 5 (1976), no. 1, 83–89.

[F+86] M. L. Fredman, R. Sedgewick, D. D. Sleator, R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. In *Algorithmica* 1 (1986) pp. 111-129.

[FT87] M. L. Fredman, R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *JACM* 34 (1987), 596–615.

[G85] H. N. Gabow. A scaling algorithm for weighted matching on general graphs. In *Proc. FOCS 1985*, 90–99.

[G01] A. Goldberg. A simple shortest path algorithm with linear average time. InterTrust Technical Report STAR-TR-01-03, March 2001.

[G01b] A. Goldberg. Shortest path algorithms: engineering aspects. *ISSAC 2001.*

[GS97] A. Goldberg, C. Silverstein. Implementations of Dijkstra's algorithm based on multi-level buckets. *Network optimization* (1997), Lec. Not. Econ. Math. Syst. 450, 292–327.

[Hag00] T. Hagerup. Improved shortest paths on the word RAM. In *Proc. ICALP 2000*, LNCS volume 1853, 61–72.

[Iac00] J. Iacono. Improved upper bounds for pairing heaps. Algorithm theory—SWAT 2000 (Bergen), LNCS vol. 1851, 32–45,

[Jak91] H. Jakobsson, Mixed-approach algorithms for transitive closure. In *Proc. ACM PODS*, 1991, pp. 199-205.

[Jar30] V. Jarník. O jistém problému minimálním. *Práca Moravské Přírodovědecké Společnosti* 6 (1930), 57–63, in Czech.

[KKP93] D. R. Karger, D. Koller, S. J. Phillips. Finding the hidden path: time bounds for all-pairs shortest paths. *SIAM J. on Comput.* 22 (1993), no. 6, 1199–1217.

[KS98] S. Kolliopoulos, C. Stein. Finding real-valued single-source shortest paths in $o(n^3)$ expected time. *J. Algorithms* 28 (1998), no. 1, 125–141.

[McG91] C. C. McGeoch. A new all-pairs shortest-path algorithm. Tech. Report 91-30 DIMACS, 1991. Also appears in *Algorithmica*, 13(5): 426-461, 1995.

[MN99] K. Mehlhorn, S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing.* Cambridge Univ. Press, 1999.

[Mey01] U. Meyer. Single source shortest paths on arbitrary directed graphs in linear average-case time. In *Proc. SODA 2001*, 797–806.

[MT87] A. Moffat, T. Takaoka. An all pairs shortest path algorithm with expected time $O(n^2 \log n)$. *SIAM J. Comput.* 16 (1987), no. 6, 1023–1031.

[MS94] B. M. E. Moret, H. D. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree. In *DIMACS Series on Discrete Math. and Theor. CS*, 1994.

[PR00] S. Pettie, V. Ramachandran. An optimal minimum spanning tree algorithm. In *Proc. ICALP 2000*, LNCS volume 1853, 49–60. *JACM*, to appear.

[PR02] S. Pettie, V. Ramachandran. Computing shortest paths with comparisons and additions. In *Proc. SODA '02*, January 2002, to appear.

[S00] P. Sanders. Fast priority queues for cached memory. *J. Experimental Algorithms* 5, article 7, 2000.

[Tak92] T. Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Inform. Process. Lett.* 43 (1992), no. 4, 195–199.

[Tho99] M. Thorup. Undirected single source shortest paths with positive integer weights in linear time. *J. Assoc. Comput. Mach.* 46 (1999), no. 3, 362–394.

[Tho01] M. Thorup. Quick k-median, k-center, and facility location for sparse graphs. In *Proc. ICALP 2001*, LNCS Vol. 2076, 249–260.

[Z01] U. Zwick. Exact and approximate distances in graphs – a survey. In *Proc. 9th ESA* (2001), 33–48. Updated copy at `http://www.cs.tau.ac.il/~zwick`