Two-Level Heaps: A New Priority Queue Structure with Applications to the Single Source Shortest Path Problem

K. Subramani[1,⋆] and Kamesh Madduri[2,⋆⋆]

[1] LDCSEE, West Virginia University
Morgantown, WV 26506, USA
ksmani@csee.wvu.edu

[2] Computational Research Division, Lawrence Berkeley National Laboratory
Berkeley, CA 94720, USA
KMadduri@lbl.gov

Abstract. The Single Source Shortest Paths problem with positive edge weights (SSSPP) is one of the more widely studied problems in Operations Research and Theoretical Computer Science [1,2] on account of its wide applicability to practical situations. This problem was first solved in polynomial time by Dijkstra [3], who showed that by extracting vertices with the smallest distance from the source and relaxing its outgoing edges, the shortest path to each vertex is obtained. Variations of this general theme have led to a number of algorithms, which work well in practice [4,5,6]. At the heart of a Dijkstra implementation is the technique used to implement a priority queue. It is well known that using Dijkstra's approach requires $\Omega(n \log n)$ steps on a graph having n vertices, since it essentially sorts vertices based on their distances from the source. Accordingly, the fastest implementation of Dijkstra's algorithm on a graph with n vertices and m edges should take $\Omega(m + n \cdot \log n)$ time and consequently the Dijkstra procedure for SSSPP using Fibonacci Heaps is optimal, in the comparison-based model. In this paper, we introduce a new data structure to implement priority queues called Two-Level Heap (TLH) and a new variant of Dijkstra's algorithm called *Phased Dijkstra*. We contrast the performance of Dijkstra's algorithm (both the simple and the phased variants) using a number of data structures to implement the priority queue and empirically establish that Two-Level heaps are far superior to Fibonacci heaps on every graph family considered.

1 Introduction

This paper is concerned with the design of fast empirical strategies for the Single Source Shortest Path problem with positive weights (SSSPP). SSSPP is one of

⋆ This research was supported in part by the Air-Force Office of Scientific Research under contract FA9550-06-1-0050 and in part by the National Science Foundation through Award CCF-0827397.

⋆⋆ This work was supported in part by the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

D.-Z. Du, X. Hu, and P.M. Pardalos (Eds.): COCOA 2009, LNCS 5573, pp. 186–196, 2009.
© Springer-Verlag Berlin Heidelberg 2009

the more widely studied problems within the Operations Research and Theoretical Computer Science communities on account of its wide applicability. This problem was first solved efficiently in [3] using a variant of Breadth-First Search (BFS). Since then, a number of variants of the general relaxation-based theme have been proposed, each claiming success on a select class of inputs. At the heart of the Dijkstra approach is a priority queue structure which implements the EXTRACT-MIN() and DECREASE-KEY() operations efficiently [1]; indeed the technique used for implementing the priority queue, more or less determines the complexity of the Dijkstra implementation. The three common techniques of priority queue implementation are arrays, binary heaps and Fibonacci heaps. A Dijkstra implementation using Fibonacci heaps runs in $O(m + n \cdot \log n)$ time, on a graph with n vertices and m edges and this is the best that one can hope for in a comparison-based model using the Dijkstra approach, since Dijkstra's algorithm sorts the vertices in terms of their actual distances from the source. However, Fibonacci heaps are notoriously difficult to implement and analyze; consequently, there is sufficient interest in investigating whether simpler alternatives providing comparable performance exist. This paper answers that question in the affirmative by detailing a modification of binary heaps called Two-Level Heaps (TLH) that are simple to visualize, analyze and implement and yet outperform Fibonacci heaps on all graph families that we considered. Our results are particularly surprising, since the asymptotic complexity of Dijkstra's algorithm using TLHs is worse than the complexity using Fibonacci heaps.

The principal contributions of this paper are as follows:

(a) The introduction of a new data structure called Two-Level Heaps (TLHs); although this structure was invented primarily for Dijkstra's algorithm, it can be used in any situation where a priority queue is required.
(b) The introduction of the Phased-Dijkstra algorithm; although asymptotically no better than simple Dijkstra, it performs better empirically.

2 Preliminaries

We assume that we are given a graph $\mathbf{G} =< V, E, s >$ with V denoting the vertex set, E denoting the edge set and s denoting the source vertex. Associated with each edge is it The basic Dijkstra approach is described by Algorithm 2.1.

The RELAX() operation is described by Algorithm 2.2.

There are two important operations that determine the complexity of Algorithm 2.1, viz., the EXTRACT-MIN() operation, which is performed once per vertex, and the RELAX() operation, which is performed once per edge. The latter operation is performed through a DECREASE-KEY() operation on the appropriately defined priority queue. Accordingly, the running time of Algorithm 2.1 is $n * T_E + m * T_D$, where T_E denotes the time required for an EXTRACT-MIN() operation and T_D denotes the time required for a DECREASE-KEY() operation. When Algorithm 2.1 completes execution, the shortest path distances are stored in the $d[]$ array. Typical implementations include a storage structure to represent the Shortest Path Tree (SPT), but we will not be concerned with the SPT.

Function SINGLE-SOURCE-SHORTEST-PATH ($\mathbf{G} =< V, E, s >$)

1: $S \leftarrow \phi$; $Q \leftarrow V$.
2: {S contains the vertices whose shortest paths from the source have been determined, while Q is the queue containing the remaining vertices. The actual distance of a node $v \in V$ from the source s is stored in $d[v]$, while its parent in the current Shortest Path Tree is stored in $\pi[v]$. The graph **G** itself is stored in adjacency list form, with the nodes that are adjacent to node u being stored in a linked list $Adj[u]$. If vertex v is on vertex u's adjacency list, then $c(u, v)$ denotes the cost of the edge from u to v. }
3: **for** (each vertex $v \in \mathbf{V}$) **do**
4: $\pi[v] =$ NIL
5: $d[v] = \infty$
6: **end for**
7: $\pi[s] = s$
8: $d[s] = 0$
9: **while** ($Q \neq \phi$) **do**
10: $r \leftarrow$ EXTRACT-MIN(Q)
11: {It is the structure of the priority-queue Q that determines the efficiency of the update operations in Dijkstra's algorithm.}
12: $S \leftarrow S \cup \{r\}$
13: **for** (each vertex $v \in Adj[r]$) **do**
14: RELAX(r, v)
15: **end for**
16: **end while**

Algorithm 2.1. Dijkstra's Algorithm

3 Phased Dijkstra

We build on the ideas of the last section to develop a phase-based implementation of Dijkstra's algorithm. This implementation requires the following data structures:

(i) A heap structure H, and
(ii) A current array A.

We first run a Breadth-First search (BFS) on G from the source s and update the $d[\ ]$ values of vertices using this search. Note that the BFS updates the

Function RELAX (u, v)

1: **if** ($d[v] > d[u] + c(u, v)$) **then**
2: $d[v] = d[u] + c(u, v)$
3: $\pi[v] = u$
4: **end if**

Algorithm 2.2. The RELAX Procedure

distance labels of vertices in the order that vertices are seen. Accordingly, the distance label of a vertex is set exactly once. The $d[\ ]$ values returned by the BFS serve as a first approximation to the true shortest path values. The vertices are then organized as a priority queue H. This is followed by extracting the $\log n$ smallest elements in H and placing them in A. The idea behind the BFS is that in many representative families (especially sparse graphs), we get an approximation which is very close to the actual shortest path distances to many of the vertices in the graph. Thus, the number of queue operations is reduced.

An EXTRACT-MIN() operation is performed by searching through the elements of A; this can be done in $\log n$ time. If the array A is emptied as a consequence of the EXTRACT-MIN(), then an additional $\log n$ elements are moved from H into A.

Let us now study the DECREASE-KEY() operation, which involves the following steps:

(a) If the element whose key is decreased is in A, then decreasing the key is an $O(1)$ operation.
(b) If the element whose key is decreased is in H, then the corresponding element could become the smallest element in H. In this case, the element is compared with the largest element in A and an appropriate exchange is made.

Clearly the time taken for a DECREASE-KEY() operation is proportional to the time required for an EXTRACT-MIN() operation on H. We thus see that from an asymptotic perspective, the phased Dijkstra approach is no better than the regular Dijkstra algorithm described in Section 2.

4 Two-Level Binary Heaps

As described in Section 2, there are n EXTRACT-MIN() calls and m DECREASE-KEY() calls in the basic flow of Dijkstra's algorithm. We now propose the following Two-Level binary heap, which permits implementations of these operations which are more efficient than the traditional Binary Heap.

Assume that the n vertices are divided into $\log^k n$ min-heaps, each containing $\frac{n}{\log^k n}$ elements[1]. The minimum elements of these heaps are then further organized as a linked list D.

Note that there are at most $n/\log^k n$ elements in each of the min-heaps and at most $\log^k n$ elements in D.

To perform an EXTRACT-MIN() operation on this structure, we first determine the smallest element in D; this takes $\log^k n$ time. This element is removed from D as is its mirror in the corresponding min-heap, say H. An element from H is then bubbled to the top and into D. Clearly the total number of steps is $\log^k n + \log \frac{n}{\log^k n}$, which simplifies to $\leq 2 \cdot \log^k n$. Note that for an EXTRACT-MIN() operation, $\log^k n$ comparisons *must* be carried out.

[1] We use $\log^k n$ to represent $(\log n)^k$, as described in [1].

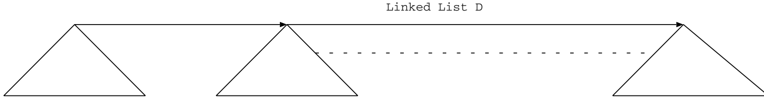

Fig. 1. Two-Level heap

Consider a DECREASE-KEY() operation; in the worst-case, an element in one of the min-heaps could bubble all the way to the top replacing its current representative in D; the time taken for this operation is $\log \frac{n}{\log^k n}$.

Thus the total number of operations taken by Algorithm 2.1 is:

$$f(k) = 2n \cdot \log^k n + m \cdot [\log \frac{n}{\log^k n}] \quad (1)$$

Optimizing for k, we get

$$k = \frac{\log[\frac{m \log \log n}{2n \ln \log n}]}{\log \log n} \quad (2)$$

Note that even at $k = 1$, our data structure beats simple min-heaps, so that for the value of k calculated above, we should do much better. It is very important to note that for each input graph, the value of k has to be recalculated and the data structure has to be built accordingly. Secondly, for very sparse graphs, it is possible that the optimal value of k is negative; in this case, we choose $k = 1$.

5 Performance Analysis

5.1 Experimental Setup

Our test platform for performance results is a 2.8 GHz 32-bit Intel Xeon machine with 4GB memory, 512KB cache and running RedHat Enterprise Linux 4. We build our codes with the Intel C compiler (icc) Version 9.0 and the optimization flag `-O3`.

We report the execution time of a baseline Breadth-First Search implementation for comparison with our shortest path code running times. The BFS running time is a natural lower bound for our SSSPP codes and is a good indicator of how close to optimal the shortest path running times are. It also removes dependencies of low-level implementation details and cache effects. Further, it gives us an estimate of the initialization time in the phased Dijkstra algorithm.

We represent the graph (vertex degree and adjacencies) using cache-friendly adjacency arrays [7], which takes optimal $O(m+n)$ space. Note that the ordering of vertex IDs determines how the graph is traversed. For the synthetic graph instances we experiment with, we randomly permute the vertex IDs so that there is no locality in the input graph. For each test instance, we choose five shortest path source vertices randomly. For each source vertex, we determine the shortest path tree and distances ten times, and compute the average execution time, removing the best and worst cases.

Our two-level binary heap implementation works for both directed and undirected instances. We use eight bytes to represent the edge weight, which can be either an positive integer or a positive real number.

5.2 Problem Families

We study performance on graph instances from several different families. However, we do not include dense graphs where $m = \Omega(n^2)$ in our experiments for two reasons: (a) Dijkstra's algorithm with an array for a priority queue is actually a linear time algorithm in this case (b) dense graphs for the graph sizes that we studied do not fit into the DRAM memory of our computer.

The synthetic graph families we experiment with are listed below. Some of the generators and graph instances are part of the 9th DIMACS Shortest Path Implementation Challenge benchmark package [8].

Random graphs: We generate graphs according to the Erdos-Renyi random graph model, and ensure that the graph is connected. The generator may produce parallel edges as well as self-loops. The ratio of the number of edges to the number of vertices, and the weight distribution, can be varied. We experiment with the following variants of random graphs:

- *iRandom-n*: Integer weight edges, the maximum weight is set to n, m is set to $4n$. n increases by a factor of two for one set of parameter values to the next. 2^{21}. Maximum weight is varied from 2^8 to 2^{20}, in multiples of sixteen.
- *rRandom-n*: Real weighted edges uniformly chosen from [0, 1], m is set to $4n$. n increases by a factor of two for one set of parameter values to the next.
- *iRandom-m*: Integer weight edges, the maximum weight is set to n, m is varied from n to $n\sqrt{n}$.

Mesh graphs: These are synthetic two-dimensional meshes with grid dimensions x and y. We generate *Long* grids where $x = \frac{n}{16}$ and $y = 16$ for this study. The diameter of these graphs is significantly higher than random graphs, and the degree distribution is uniform. We define the graph families (*iLong-n* and *rLong-n* similar to the Random graph family.

Small-world networks: We use the Recursive MATrix (R-MAT) [9] random graph generation algorithm to generate input data sampled from a Kronecker product that are representative of real-world networks with a small-world topology. As in the above random graph classes, we set $m = 4n$, and experiment with both integer (iSW) and real-weighted (rSW) graphs. Small-world networks have a low diameter and an unbalanced degree distribution.

The graph generators we use write the graphs to disk in plain text format, which the shortest path implementations parse and load to memory. The graph generation and loading steps are not timed.

5.3 Shortest Path Implementations

The worst case complexity of Dijkstra's algorithm depends on the priority queue implementation used to store the visited vertices. Cherkassky et al. [10] conducted an extensive experimental evaluation of data structures for solving the

shortest paths problem with non-negative integral edge weights. We use the code from the SPLIB library [11] (that implements the data structures discussed in [10]) for comparison with our two-level binary heap data structure. Note that both our implementation and SPLIB use the same internal representation for the graph (adjacency arrays), are coded in C, compiled with the same compiler (Intel C compiler) and optimization flags, and execution time results are obtained with an identical experimental test setup. We make straight-forward modifications to the Fibonacci Heap and Binary Heap implementations in SPLIB to handle real edge weights.

We compare the performance of Dijkstra's algorithm for graphs with both real and integer weights using the following priority queue representations:

– *Queue*: storing the visited vertices in an array, which results in a worst case complexity of $O(n^2)$.
– *BHeap*: using a binary heap that runs in $O(m \log n)$ time.
– *Fib*: using Fibonacci heaps, that has a worst case $O(m + n \log n)$ complexity.
– *TLHeap*: our two-level heap data structure.

We also evaluate the performance of the Phased Dijkstra algorithm using all the priority queue representations. We denote the implementations *PhBHeap*, *PhFib* and *PhTLHeap*.

5.4 Experimental Results

For each graph family, we present two plots, one corresponding to Dijkstra's algorithm implementations and the second for phased Dijkstra implementations with the same data structures. Figure 2 gives the performance of the various data structures for the *iRandom-n* family, where the number of vertices varied by two orders of magnitude from 2^{18} to 2^{24}. We find that *TLHeap* outperforms the other implementations for problem instances where n is greater than 2^{20}. On an average, the *TLHeap* implementation is 1.6× faster than *BHeap* and 2.64× faster than *Fib*. The performance of the phase-based Dijkstra implementations for the heap and two level-heap implementations are comparable to the regular Dijkstra implementations. However, the phase-based Dijkstra with a Fibonacci heap priority queue implementation performs better than *Fib* (a 20% improvement for the largest problem size). Note that in case of the *iRandom* graph family, the number of vertices in the priority queue is typically large. Our heuristic of maintaining a current array in the phased Dijkstra implementation improves the performance of the Fibonacci heap implementation. As the problem size increases, the performance of the Fibonacci heap implementation does not scale as well as the other data structures. This can be attributed to the pointer-intensive nature of the Fibonacci heap data structure, which leads to poor cache performance in comparison to the other implementations.

In Figure 3, we observe trends identical to integer-weighted case. For large networks, *TLHeap* performs better than the other implementations. Phased Dijkstra results in a minor 5% improvement in performance of *TLHeap* and *BHeap*

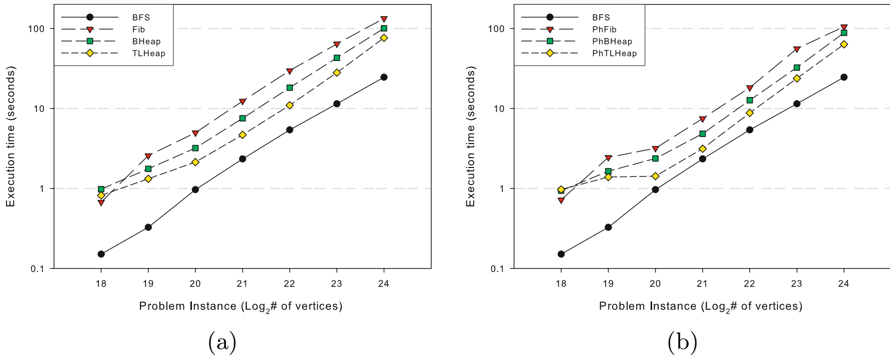

Fig. 2. Execution time for graph instances from the *iRandom-n* family. n is varied from 2^{18} to 2^{24}, and m is set to $4n$.

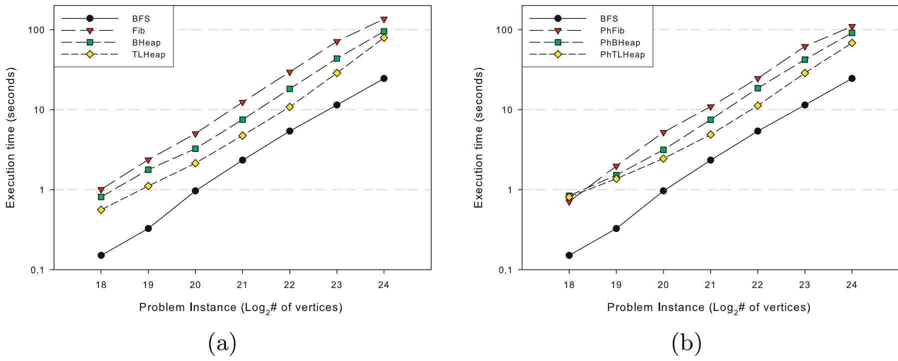

Fig. 3. Execution time for graph instances from the *rRandom-n* family. n is varied from 2^{18} to 2^{24}, and m is set to $4n$.

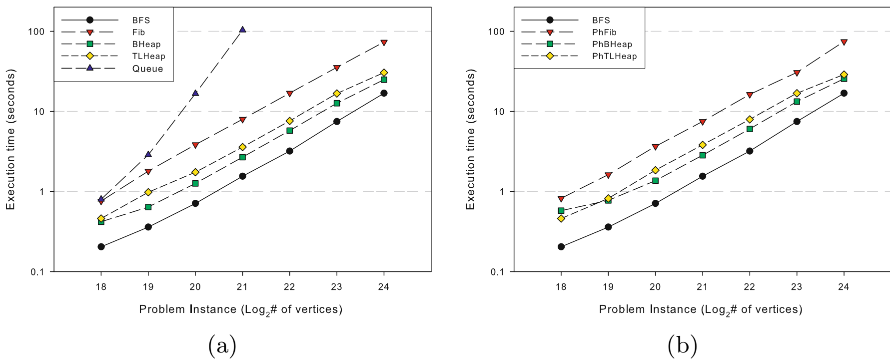

Fig. 4. Execution time for graph instances from the *iLong-n* family. n is varied from 2^{18} to 2^{24}, and m is set to $4n$.

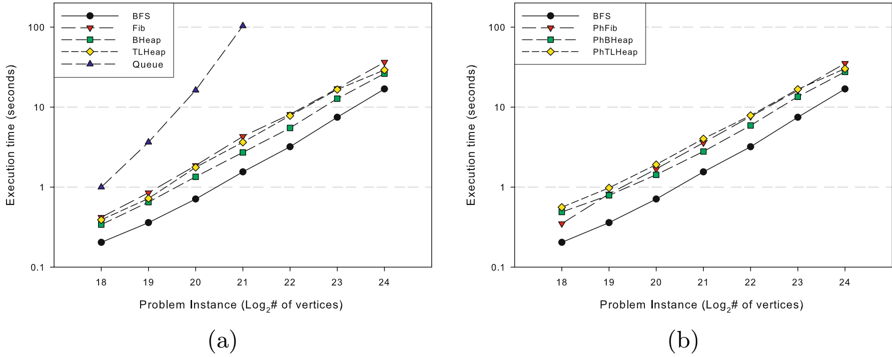

Fig. 5. Execution time for graph instances from the *rLong-n* family. n is varied from 2^{18} to 2^{24}, and m is set to $4n$.

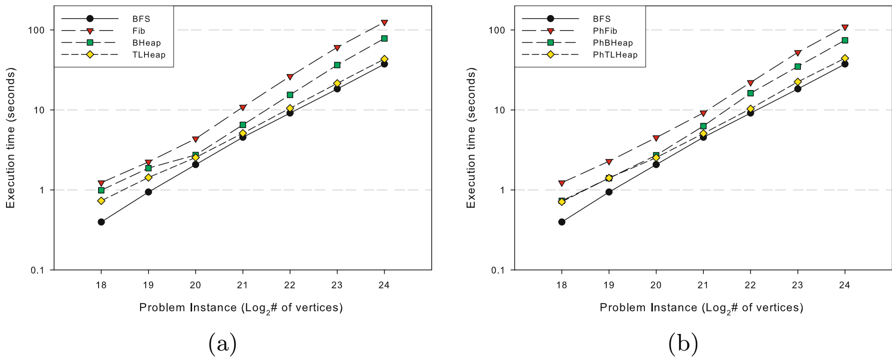

Fig. 6. Execution time for graph instances from the *iSW-n* family. n is varied from 2^{18} to 2^{24}, and m is set to $4n$.

for this family. The results show that our implementations perform equally well on graphs classes with integer and real-weighted edges.

Figure 4 and 5 give the performance of our implementations on integer and real-weighted mesh networks respectively. These are high diameter graphs, and the number of visited vertices in the priority queue is comparatively smaller than the Random-n family of graphs. In this case, the running times of *BHeap* is nearly twice as slow as the reference BFS. *TLHeap* is slightly slower than *BHeap*, as the overhead for having two levels is not justified with a low number of vertices in the priority queue in practice. The performance of *Fib* is comparable to *TLHeap*. For smaller problem sizes, the running time of *Queue* may be comparable to the other implementations. There are also no significant gains in using the phase-based Dijkstra for any of the data structures.

Figure 6 gives the performance of the implementations for synthetic small-world networks with integer-weighted edges. These are typically low diameter

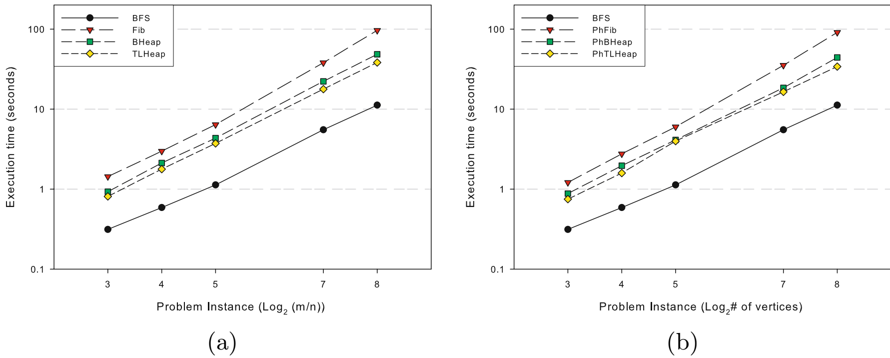

Fig. 7. Execution time for graph instances from the *iRandom-m* family. n is set to 2^{17}, and m is varied from 2^{20} to 2^{25}.

graphs, and we expect the performance to be similar to the *Random-n* family. *TLHeap* and *PhTLHeap* outperform the other Dijkstra-based implementations, and the performance of *PhFib* is significantly better than *Fib*. The running time of *TLHeap* is on an average 1.23× the time taken for BFS, which is the best result across all graph families.

Next, we vary the network sparsity for the random graph family. Figure 7 plots the performance as m is varied across two orders of magnitude, keeping n fixed. We observe the same trends as in the previous case of the *iRandom-n* family. As m increases, the performance of *TLHeap* and *BHeap* is very similar, and they are faster than *Fib*.

6 Conclusion

Our work in this paper was motivated primarily by the need to find a simpler alternative to Fibonacci heaps. Towards this end, we designed the Two-Level Heap structure, which is easy to implement and analyze. When the data structure was designed, we expected that its performance over Dijkstra computations would be comparable (and perhaps somewhat inferior) to the performance of Fibonacci heaps. Our empirical results seem to indicate that not only are Two-Level heaps comparable to Fibonacci heaps, but that they are far superior to the same. This observation is very surprising, considering that Two-Level heaps are in fact asymptotically inferior to Fibonacci heaps over Dijkstra computations. We also note that Two-Level heaps are stand alone structures and could conceivably be used as priority queues in more general applications; consequently, a more detailed study of this structure is merited.

Acknowledgments

The research of the first author was conducted in part at the Discrete Algorithms and Mathematics Department, Sandia National Laboratories, Albuquerque, New

Mexico. Sandia is a multipurpose laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

References

1. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge (2001)
2. Goldberg, A.V.: Scaling algorithms for the shortest paths problem. SIAM Journal on Computing 24(3), 494–504 (1995)
3. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik 1, 269–271 (1959)
4. Raman, R.: Recent results in single-source shortest paths problem. SIGACT news 28, 81–87 (1997)
5. Ahuja, R.K., Mehlhorn, K., Orlin, J.B., Tarjan, R.E.: Faster algorithms for the shortest path problem. Journal of the ACM 37(2), 213–223 (1990)
6. Pallottino, S.: Shortest path methods: Complexity, interrelations and new propositions. NETWORKS: Networks: An International Journal 14, 257–267 (1984)
7. Park, J., Penner, M., Prasanna, V.: Optimizing graph algorithms for improved cache performance. In: Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2002), Fort Lauderdale, FL (April 2002)
8. Demetrescu, C., Goldberg, A., Johnson, D.: 9th DIMACS implementation challenge – Shortest Paths, http://www.dis.uniroma1.it/~challenge9/
9. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: A recursive model for graph mining. In: Proc. 4th SIAM Intl. Conf. on Data Mining, Florida, USA (April 2004)
10. Cherkassky, B., Goldberg, A., Radzik, T.: Shortest paths algorithms: theory and experimental evaluation. Mathematical Programming 73, 129–174 (1996)
11. Goldberg, A.: Network optimization library, http://www.avglab.com/andrew/soft.html