Implementations of Dijkstra's Algorithm Based on Multi-Level Buckets

Andrew V. Goldberg *

NEC Research Institute
4 Independence Way
Princeton, NJ 08540
avg@research.nj.nec.com

Craig Silverstein

Computer Science Department
Stanford University
Stanford, CA 94305
csilvers@theory.stanford.edu

November 1995

Abstract

A 2-level bucket data structure has been shown to perform well in a Dijkstra's algorithm implementation [4, 5]. In this paper we study how the implementation performance depends on the number of bucket levels used. In particular we are interested in the best number of levels to use in practice.

*Part of this work was done while the author was at Computer Science Department, Stanford University, and supported in part by NSF Grant CCR-9307045.

1 Introduction

The shortest paths problem is a fundamental network optimization problem. Algorithms for this problem have been studied for a long time. (See *e.g.* [2, 7, 8, 10, 14, 15, 16].)

An important special case of the problem occurs when no arc length is negative. In this case, implementations of Dijkstra's algorithm [8] achieve the best time bounds. An implementation of [11] runs in $O(m + n \log n)$ time. (Here n and m denote the number of nodes and arcs in the network, respectively.) An improved time bound of $O(m+n \log n / \log \log n)$ [12] can be obtained in a random access machine computation model that allows certain word operations. Under the assumption that arc lengths are integers in the interval $[0, \ldots, C]$, $C \geq 2$, the implementation of [1] runs in $O(m + n\sqrt{\log C})$ time.

In a recent computational study [4, 5], however, a 2-level bucket implementation of Dijkstra's algorithm gave the best overall performance among the codes studied. In particular, the implementation proved to be much more robust than the classical 1-level bucket implementation [7, 9, 18]. In this paper we study relative performance of the multi-level bucket implementations of the algorithm. We conduct computational experiments and explain their results. Our study leads to better understanding of the multi-level implementations and confirms that the 1-level implementation is much less robust than the multi-level implementations. The 1-level implementation should be used only on special problems, such as problems with small arc lengths. On the other hand, implementations using more than one level of buckets are robust, performing consistently over a wide range of inputs and performing poorly only on tests specifically designed to be difficult for a particular implementation.

2 Definitions and Notation

The input to the one-source shortest paths problem is $\langle G, s, \ell \rangle$, where $G = (V, E)$ is a directed graph, $\ell :\rightarrow \mathbf{R}$ is a length function, and $s \in V$ is the source node. In this paper we assume that the length function is nonnegative and that all nodes in G are reachable from s. The goal is to find, for each node $v \in V$, the shortest path from s to v. We denote $|V|$ by n, $|E|$ by m, and the largest arc length by C.

A *shortest paths tree* of G is a spanning tree rooted at s such that for any $v \in V$, the reversal of the v to s path in the tree is a shortest path from s to v.

3 Dijkstra's Algorithm

Dijkstra's algorithm [8] for solving the shortest path problem with nonnegative length function works as follows. (See *e.g.* [6, 13, 17] for more detail.) For every node v, the algorithm maintains a distance label $d(v)$, parent $\pi(v)$, and status $S(v) \in \{unreached, labeled, scanned\}$. These values are initially $d(v) = \infty$, $\pi(v) = \text{nil}$, and $S(v) = unreached$ for each node. The method starts by setting $d(s) = 0$ and $S(s) = labeled$.

At each step, the algorithm selects a labeled node with the smallest distance label and applies the SCAN operation to it. If there are no labeled nodes, the algorithm terminates.

The SCAN operation, applied to a labeled node v, examines arcs (v, w). If $d(v) + \ell(v, w) < d(w)$, then $d(w)$ is set to $d(v) + \ell(v, w)$, $\pi(w)$ is set to v, and $S(w)$ is set to *labeled*. $S(v)$ is then set to *scanned*.

This algorithm terminates, giving both the shortest paths and their lengths:

Theorem 3.1 *If the length function is nonnegative and every node is reachable from s, Dijkstra's algorithm scans each node exactly once and terminates with d giving the shortest path distances and π giving a shortest path tree.*

In addition, the algorithm examines each edge exactly once. The worst-case complexity of Dijkstra's algorithm depends on the method used to find the labeled node with the smallest distance label. The implementation using Fibonacci heaps [11] runs in $O(m + n \log n)$ time. The implementation using R-heaps [1] runs in $O(m + n\sqrt{\log C})$ time.

4 Multi-Level Bucket Implementation

4.1 1-level Bucket Implementation

Another way to implement Dijkstra's algorithm is by using the bucket data structure, proposed independently by Dial [7], Wagner [18], and Dinitz [9]. This implementation maintains an array of buckets, with the i-th bucket containing all nodes v with $d(v) = i$. When a node's distance label changes, the node is removed from the bucket corresponding to its old distance label (if the label was finite) and inserted into the bucket corresponding to the new one.

The implementation maintains an index L. Initially, $L = 0$, and L has the property that all buckets $i < L$ are empty. If L is empty, it is incremented, otherwise the next node to be scanned is removed from bucket L. The following theorem follows easily from the observation that a bucket deletion or insertion takes constant time and at most nC buckets need to be examined by the algorithm.

Theorem 4.1 *If the length function is nonnegative, the bucket-based implementation of Dijkstra's algorithm runs in* $O(m + nC)$ *time.*

Although the algorithm, as stated, needs nC buckets, it can be easily modified to use only $C + 1$. The key observation is that at most $C + 1$ consecutive buckets can be occupied at any given time, and we can "wrap around" when the end of the bucket array is reached.

4.2 2-level Bucket Implementation

A 2-level bucket structure reduces the memory requirement even further and also improves the time bound. The basic 2-level bucket implementation works as follows: there are $\sqrt{C+1}$ top-level buckets, each of which contains $\sqrt{C+1}$ bottom-level buckets. Each bottom-level bucket holds one distance label, as in the 1-level implementation, but each top-level bucket holds a range of $\sqrt{C+1}$ distance labels, corresponding to the labels on the bottom-level buckets contained in that top level bucket. We keep two indices, L_{top} and L_{bottom}, to indicate our current position in the data structure. When moving a node to a new location, we find first the appropriate top-level bucket for that node and then the appropriate bottom-level bucket within that top-level bucket.

The time and space savings come when we modify the basic algorithm to keep only *one* set of bottom-level buckets, the set associated with the current top-level bucket at index L_{top}. When moving a node, we put it into the appropriate top-level bucket. We only move it into a bottom-level bucket if the node is in the top-level bucket at L_{top}. When L_{top} changes (because all the bottom-level buckets become empty), we must *expand* the bucket at the new L_{top}, putting all the nodes in bucket L_{top} into appropriate bottom-level buckets. We can destroy the bottom-level buckets for the bucket at the old L_{top}, since they are now all empty, and reuse the space for the new active bucket.

If there are many empty buckets, the 2-level implementation saves time as well: if one of the top-level buckets is empty, we move to the next without the need to expand, thereby skipping $\sqrt{C+1}$ distance values at once.

It is clear from this description that the total space requirement is $2\sqrt{C+1}$ buckets. Expansion takes constant time per node, and we expand each node at most once. In addition, each node can make us examine at most $\sqrt{C+1}$ bottom-level buckets; we may also have to examine $\sqrt{C+1}$ top level buckets. Thus the time is in $O(m + n(1 + \sqrt{C}))$.

4.3 k-level Bucket Implementation

The scheme for 2-level buckets can easily be extended to allow for more levels. Formally, suppose we have k bucket levels, with $p = \lceil C^{1/k} \rceil$ buckets at each level. The lowest bucket level is 0, and $k-1$ is the highest. In addition, the buckets in each level are numbered from 0 to $p-1$. Consider level i. Associated with this level are *the base distance* B_i and the currently active bucket L_i. Associated with bucket j at level i is the interval $[B_i + jp^i, B_i + (j+1)p^i - 1]$, representing the possible distance labels of nodes in that bucket. The base distances and indices are such that $B_{k-1} = 0 \bmod p^k$ and $B_{i-1} = B_i + L_i p^i$.

The algorithm repeatedly removes a node from the active bucket at the lowest level and updates the distances of all its neighbors. If the distance of a node decreases, we try to replace it at the lowest level. If its distance label does not fit in any interval of the lowest-level buckets, we move up a level and try to fit the node in a higher level bucket, otherwise we put the node in the bucket with the fitting interval.

Once the bottom-level bucket at L_0 becomes empty, we update L_0 by scanning for the next non-empty bucket at the lowest level. If there is none, we go up a level and repeat. Suppose we find a non-empty bucket on level i. We update L_i and expand the non-empty bucket. We set L_{i-1} to be the index of the first non-empty bucket among the expanded buckets. If necessary, we expand L_{i-1} as well, until we have a new, non-empty active bucket at the bottom level. The algorithm then continues.

The space and time bounds on the k-level implementation are generalizations of those for the 2-level case.

Theorem 4.2 [5] *If the length function is nonnegative, the k-level implementation runs in $O(m + n(k + C^{1/k}))$ time and uses $\Theta(kC^{1/k})$ buckets.*

Although the multi-level implementation does not match the best time bounds known for this problem, the time bound is close, and its performance in practice is competitive with other implementations.

4.4 Heuristics

Our implementation uses two heuristics to improve practical performance. These heuristics have low overhead: They never decrease performance by much, and they often give significant time savings.

The first heuristic, which we call *the minimum length heuristic*, is due to Dinitz [9]. Let M be is the smallest nonzero arc cost (we assume that at least one arc length is positive). Then the bucket-based implementations remain correct if the i-th lowest level bucket contains nodes with distance labels in the range $[iM, \ldots, (i+1)M)$. This heuristic reduces the number of buckets used.

The minimum length heuristic allows to use bucket-based algorithms on problems with nonnegative real-valued length functions. This can be achieved by dividing all arc lengths by M. In this case, C is defined as the ratio between the biggest and the smallest positive lengths.

The second heuristic, which we call *the end cutoff heuristic*, is due to Cherkassky [5]. This heuristic keeps track of the first and the last nonempty bucket at each level, which allows the algorithm to skip empty buckets at the ends of the bucket array. The heuristic is more helpful than it may look at first. In particular, consider the 1-level implementation and recall that this implementation uses $C + 1$ buckets and "wraps around" when the end of the bucket array is reached. Suppose the input graph is a path from s, with each arc length equal to C. Without the end cutoff heuristic, the implementation takes $\Theta(nC)$ time. With the heuristic, it takes only $\Theta(n)$ time.

4.5 Bucket Overhead

We study how the implementation performance depends on the number of bucket levels. To interpret our experimental results, it is important to understand the overhead of maintaining and searching the buckets. The major overhead sources are as follows. (We count the work of removing a node from a bucket to be scanned as a part of scanning the node and not as overhead.)

1. **Examining empty buckets:** the overhead is proportional to the total number of empty buckets examined. An *empty bucket operation* consists of examining a bucket which turns out to be empty.

2. **Expanding buckets:** the overhead is proportional to the total number of nodes moved to a lower level during bucket expansions. An *expansion operation* consists of one such node move.

3. **Node moves due to distance label decreases:** the overhead is proportional to the total number of times a node needs to be moved to a different bucket when its distance label decreases. A *move operation* consists of such a node move.

5 Experimental Setup

Our experiments were conducted on a SUN Sparc-10 workstation model 41 with a 40MHZ processor running SUN Unix version 4.1.3. The workstation had 160 Meg. memory and all problem instances fit into the memory. Our code was written in C++ and compiled with the SUN gcc compiler version 2.6.3 using the $-$O2 optimization option.

We made an effort to make our code efficient. In particular, we set the bucket array sizes to be powers of two. This allows us to use word shift operations when computing bucket array indices.

We report experimental results obtained on four types of graphs and on four levels of buckets. Two of the graph types were chosen to exhibit the properties of the algorithm at two extremes: one where the paths from the start node to other nodes tend to be order $\Theta(n)$, and one in which the path lengths are order $\Theta(1)$. The third graph type is random graphs. The fourth type of graphs is meant to be easy or hard for a specific implementation with a specific number of bucket levels. We experimented with several additional problem families. However, these additional results were consistent with those we report here and do not add new insight. The bucket levels ranged from 1 to 4; the distinction between the performance of a 3-level implementation and a 4-level implementation is so slight that any deeper nesting of buckets is unlikely to significantly improve performance.

To put performance of the bucket implementations in perspective, we also give data for a k-ary heap implementation of Dijkstra's algorithm with $k = 4$. (We picked $k = 4$ so we could use word shift operations.) The k-ary heap data is useful, for example, to gauge relative difference in the multi-level bucket implementation performance, or to see if very large costs are as bad for the multi-level bucket implementations as the worst-case analysis suggests. We would like to point out that the experiments described in this paper are designed to compare the multi-level bucket implementations to each other, not to the k-ary heap implementation. A comparison of a 2-level bucket implementation to a k-ary heap implementation appears in [5], and our data is consistent with that of [5].

5.1 The Graph Types

Two types of graphs we explored were grids produced using the GRIDGEN generator [5]. These graphs can be characterized by a length x and width y. The graph is formed by constructing x layers, each of which is a path of length y. We order the layers, as well as the nodes within each

Name	type	description	salient feature
long grid	grid	16 nodes high $n/16$ nodes long	path lengths are $\Theta(n)$
wide grid	grid	$n/16$ nodes high 16 nodes long	path lengths are $\Theta(1)$
random	random	degree 4	path lengths are $\Theta(\log n)$
hard	two paths	$d(S, \text{path } 1) = 0$ $d(S, \text{path } 2) = p - 1$	nodes occupy first and last buckets in bottom level bins
easy	two paths	$d(S, \text{path } 1) = 0$ $d(S, \text{path } 2) = 1$	nodes occupy first and second buckets in bottom level bins

Table 1: The graph types used in our experiments. p is the number of buckets at each level.

layer, and we connect each node to its corresponding node on adjacent layers. All the nodes on the first layer are connected to the source.

The first type of graph we used, the LONG GRID, has a constant width — 16 nodes in our tests. We used graphs of different lengths, ranging from 512 to 32768 nodes. The arcs had lengths chosen independently and uniformly at random in the range from 1 to C. C varied from 1 to $100,000,000$.

The second type of graph we used was the WIDE GRID type. These graphs have length limited to 16 layers, while the width can vary from 512 to 32768 nodes. C was the same as for LONG GRIDS.

The third type of graphs includes random graphs with uniform arc length distribution. A random graph with n nodes has $4n$ arcs.

The fourth type of graphs includes both HARD and EASY graphs. The input to these graphs is the number of nodes, the desired number of levels k and a maximum arc length C. From C it is possible to calculate p, the number of buckets in each level assuming the implementation has k levels. Both graphs consist of two paths connected to the source. The nodes in each path are at distance p from each other. The distance from the source to path 1 is 0; nodes in this path will occupy the first bucket of bottom level bins. The distance from the source to path 2 is $p-1$ for HARD graphs — making these nodes occupy the last bucket in each bottom-level bin — and 1 for EASY graphs —making the nodes occupy the second bucket in each bottom-level bin. In addition, the source is connected to the last node on the first path by an arc of length 1, and to the last node of the second path by an arc of length C.

Graph type	Graph family	Range of values	Other values
long grid	Modifying C	$C = 1$ to $1,000,000$	$x = 8192$
	Modifying x	$x = 512$ to 32768	$C = 16$
			$C = 10,000$
			$C = 100,000,000$
	Modifying C and x	$x = 512$ to 32768	$C = x$
			$C = x/10$
wide grid	Modifying C	$C = 1$ to $1,000,000$	$y = 8192$
	Modifying y	$y = 512$ to 32768	$C = 16$
			$C = 10,000$
			$C = 100,000,000$
	Modifying C and y	$y = 512$ to 32768	$C = y$
			$C = y/10$
random graph	Modifying C	$C = 1$ to $1,000,000$	$n = 131072$
	Modifying n	$n = 8,192$ to $524,288$	$C = 16$
			$C = 10,000$
			$C = 100,000,000$
	Modifying C and n	$n = 8,192$ to $524,288$	$C = n$
			$C = n/10$
easy, hard	Modifying C	$C = 100$ to $10,000,000$	$n = 131072, p = 2$
			$n = 131072, p = 3$

Table 2: The problem families used in our experiments. C is the maximum arc length; x and y the length and width, respectively, of grid graphs; and p the number of levels for which easy and hard graphs are meant to be easy or hard.

A summary of our graph types appears in Table 1.

5.2 Problem Families

For each graph type we examined how the relative performance of the implementations changed as we increased various parameters. Each type of modification constitutes a *problem family*. The families are summarized in Table 2. In general, each family is constructed by varying one parameter while holding the others constant. Different families can vary the same parameter, using different constant values. For instance, one problem family modifies x as $C = 16$, another modifies x as $C = 10,000$, and a third modifies x as $C = 100,000,000$.

6 Data Interpretation

We use the overhead operation counts, from Section 4.5, to explain the data. The work performed actually scanning nodes is the same for all implementations; variations in overall cost come from differing amounts of overhead. Since each node is scanned exactly once, it is often helpful to look at the number of overhead operations per node.

Relative cost of the overhead operations is important. The work involved in an empty bucket operation is much less than the work involved in an expansion or a move operation. A move is about twice as expensive as an expansion, since expansion merely involves insertion, while moving involves deletion as well. Scanning a node involves removing it from an appropriate bucket, examining its outgoing arcs, and potentially changing the distance labels and parent pointers of its neighbors. Even though all networks we study have small degree, scanning a node takes more time than an expansion or a move operation and much more time than an empty bucket operation.

The cost of insertion and deletion, although bounded by a constant, is not uniform. Inserting into an empty bucket is about half as expensive as inserting into a non-empty bucket, due to the cost of updating the doubly-linked list. Likewise, deleting the last node from a bucket is cheaper than deleting a penultimate, or earlier, node. Usually it is not necessary to distinguish between the two types of insertions and deletions — we do not do so — but we will refer to this fact when it is needed to explain the data.

The number of overhead operations has a significant effect on the running time only if there is significantly more than one overhead operation per node.

Often, the relative implementation performance is determined by the number of empty bucket operations. The advantage of multiple bucket levels is that after examining an empty bucket we may increase L by a large amount. This is a game of diminishing returns, however, since the rate of decrease of empty bucket operations is less than the rate of increase of expansion operations.

Several key statistics relate to the distribution of path lengths. We define the *depth* D of a network to be the highest distance from the source to a node reachable from the source. Network depth is an important parameter in understanding performance of our implementations. Without the minimum length and end cutoff heuristics, the one level implementation examines exactly $D+1$ buckets until there are no labeled nodes. Even with the heuristics and multiple levels, the number of empty operations usually grows as D grows. Depth can often be used to explain performance.

The variance of the shortest path lengths is also an important statistic. If the distribution of shortest path lengths is highly non-uniform, there will be large stretches of empty buckets which multi-level implementations can quickly skip over.

Equally crucial is the density of the distribution: If there are few empty buckets, the overhead of bucket expansion may well be higher than the overhead of examining empty buckets, favoring small numbers of bucket levels. Distributions for grids are fairly uniform, and vary in density as C varies. D/n gives a fairly good estimate of distribution density for shortest path lengths.

7 Experimental Results

In this section we present our experimental results. In all the tables, k denotes the number of bucket levels.

As we have mentioned above, the k-ary heap data is given mostly for calibration purposes. This data has a succinct interpretation, however, which we give in Section 7.5.

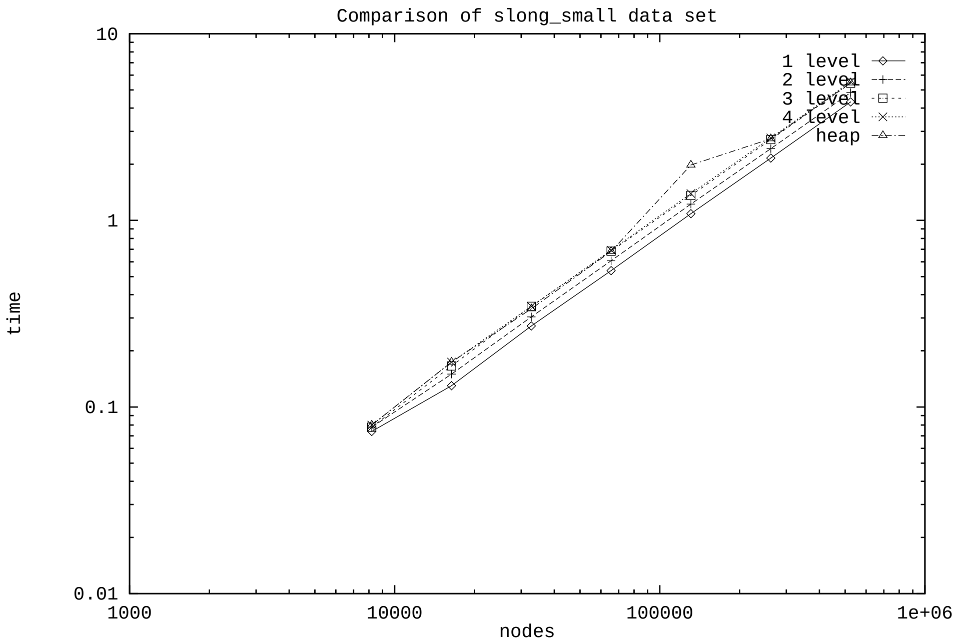
7.1 Varying Grid Size

Tables 3, 4, and 5 show the relative performance of our implementations on long grids as the size of the grid changes. The first table concerns LONG-SMALL networks with $C = 16$, the second LONG-MEDIUM networks with $C = 10,000$, and the third LONG-LARGE networks with $C = 100,000,000$.

For LONG-SMALL networks, D is comparable to n. The number of empty bucket operations is small and multiple bucket levels do not help. On these networks, performance of all four bucket implementations is very similar. The 1-level implementation the fastest by a small margin. The 3- and 4-level implementations perform almost identically and are the slowest by a small margin.

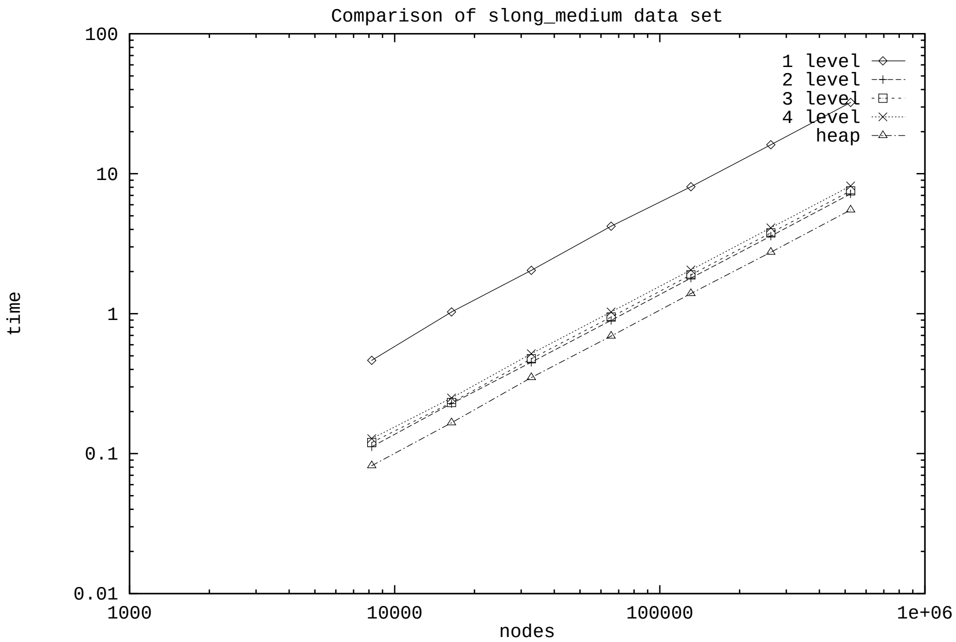
The relative performance is consistent with the operation counts. The number of empty bucket operations and the number of move operations is similar for all implementations. While the 1-level implementation does no expansion operations, the other bucket implementations do less than one expansion operation per node, and the relative running time differences are small.

For LONG-MEDIUM networks, D is much greater than n. The 1-level implementation is slower than the other bucket implementations because it performs many more empty bucket operations — about two hundred per node. The running time of the 1-level implementation is dominated by the time spend examining empty buckets. The number of move operations for all



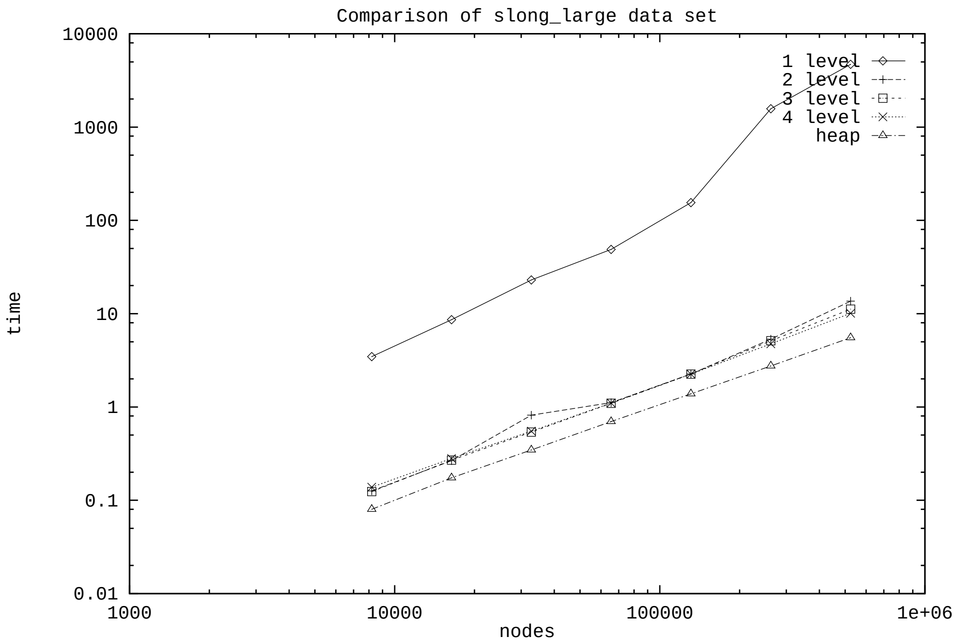
k	nodes	8193	16385	32769	65537	131073	262145	524289
1	**time**	**0.07** s	**0.13** s	**0.27** s	**0.54** s	**1.09** s	**2.16** s	**4.31** s
	empty	175	324	659	1296	2566	5086	10308
	expanded	0	0	0	0	0	0	0
	moved	10346	20728	41475	82950	165726	331441	663107
2	**time**	**0.08** s	**0.15** s	**0.30** s	**0.61** s	**1.22** s	**2.43** s	**4.84** s
	empty	135	258	522	1033	2024	4041	8181
	expanded	5626	11259	22544	45151	90331	180596	361254
	moved	9607	19213	38433	76835	153577	307080	614622
3	**time**	**0.08** s	**0.17** s	**0.35** s	**0.68** s	**1.36** s	**2.71** s	**5.44** s
	empty	88	163	335	650	1271	2521	5155
	expanded	10646	21329	42772	85533	171125	342183	684440
	moved	9475	18973	37937	75885	151681	303295	606800
4	**time**	**0.08** s	**0.17** s	**0.35** s	**0.69** s	**1.39** s	**2.77** s	**5.53** s
	empty	88	163	335	650	1271	2521	5155
	expanded	11582	23198	46515	93040	186126	372184	744542
	moved	9475	18973	37937	75885	151681	303295	606800
h	**time**	**0.08** s	**0.17** s	**0.34** s	**0.68** s	**1.98** s	**2.74** s	**5.46** s
	moved	10342	20726	41476	82936	165712	331496	663082

Table 3: The performance on long grids as the grid length increases, for $C = 16$.



k	nodes	8193	16385	32769	65537	131073	262145	524289
1	**time**	**0.46** s	**1.03** s	**2.04** s	**4.21** s	**8.07** s	**16.13** s	**32.29** s
	empty	1469907	3251249	6519227	12993783	25974606	51938208	104134091
	expanded	0	0	0	0	0	0	0
	moved	10670	21367	42782	85485	170925	341972	683785
2	**time**	**0.11** s	**0.23** s	**0.45** s	**0.90** s	**1.80** s	**3.59** s	**7.19** s
	empty	92143	171430	344659	684436	1377759	2747626	5506539
	expanded	8068	16185	32374	64753	129491	259030	517975
	moved	10611	21274	42606	85117	170201	340490	680859
3	**time**	**0.12** s	**0.23** s	**0.48** s	**0.95** s	**1.89** s	**3.78** s	**7.55** s
	empty	30448	63751	126799	253677	505943	1013106	2025712
	expanded	15449	31205	62428	124910	249828	499598	999259
	moved	10302	20700	41504	82864	165671	331377	662686
4	**time**	**0.13** s	**0.25** s	**0.52** s	**1.03** s	**2.05** s	**4.10** s	**8.18** s
	empty	15211	29968	59331	118878	237499	475558	950782
	expanded	21472	43698	87489	174962	349963	699931	1399914
	moved	9829	19660	39354	78627	157385	314680	629336
h	**time**	**0.08** s	**0.17** s	**0.35** s	**0.69** s	**1.39** s	**2.75** s	**5.51** s
	moved	10669	21367	42782	85484	170922	341972	683787

Table 4: The performance on long grids as the grid length increases, for $C = 10,000$.



k	nodes	8193	16385	32769	65537	131073	262145	524289
1	**time**	**3.46** s	**8.63** s	**22.96** s	**48.80** s	**154.85** s	**1575.18** s	**4711.51** s
	empty	11954646	30067065	82573330	177903397	557755127	2913256282	3391076670
	expanded	0	0	0	0	0	0	0
	moved	10670	21368	42783	85490	170931	341988	684013
2	**time**	**0.13** s	**0.27** s	**0.82** s	**1.11** s	**2.26** s	**5.33** s	**13.63** s
	empty	119833	302413	594727	1321875	2641065	7073796	21891135
	expanded	8137	16292	32639	65271	130678	261726	523819
	moved	10646	21328	42724	85370	170754	341792	683595
3	**time**	**0.12** s	**0.27** s	**0.54** s	**1.10** s	**2.25** s	**5.13** s	**11.18** s
	empty	51620	119716	256973	516915	1213987	4117328	10407425
	expanded	15892	32059	64261	129015	258528	518070	1040303
	moved	10462	21043	42221	84565	169274	339129	679966
4	**time**	**0.14** s	**0.28** s	**0.55** s	**1.10** s	**2.25** s	**4.76** s	**10.11** s
	empty	19402	42993	95630	203827	445451	1261218	4153649
	expanded	22817	46097	92270	182036	372381	766285	1533172
	moved	10062	20246	40572	80193	162887	333387	666659
h	**time**	**0.08** s	**0.17** s	**0.35** s	**0.70** s	**1.38** s	**2.75** s	**5.53** s
	moved	10670	21368	42783	85490	170931	341989	683818

Table 5: The performance on long grids as the grid length increases, for $C = 100,000,000$.

implementations is a little over one per node and has little effect on the relative performance.

For 2-, 3-, and 4-level implementations, empty buckets do not provide the dominant cost. While the 1-level implementation examines 50-100 times as many empty buckets as the 2-level implementation, the 2-level implementation examines only 3-5 times as many empty buckets as the 4-level implementation. The cost of expansion becomes dominant, so the 2-level implementation is the fastest, followed by the 3- and 4-level implementations.

For LONG-LARGE networks, D is huge compared to n. We would thus expect the same behavior as for LONG-MEDIUM networks: 1-level implementations suffer due to the huge number of empty buckets, while multi-level implementations can skip over the huge swaths of empty buckets at an increase in expansion operations. And indeed, the 1-level implementation performs poorly. Implementations with several bucket levels perform similarly to each other for small n. For large n, the 4-level implementation is somewhat better.

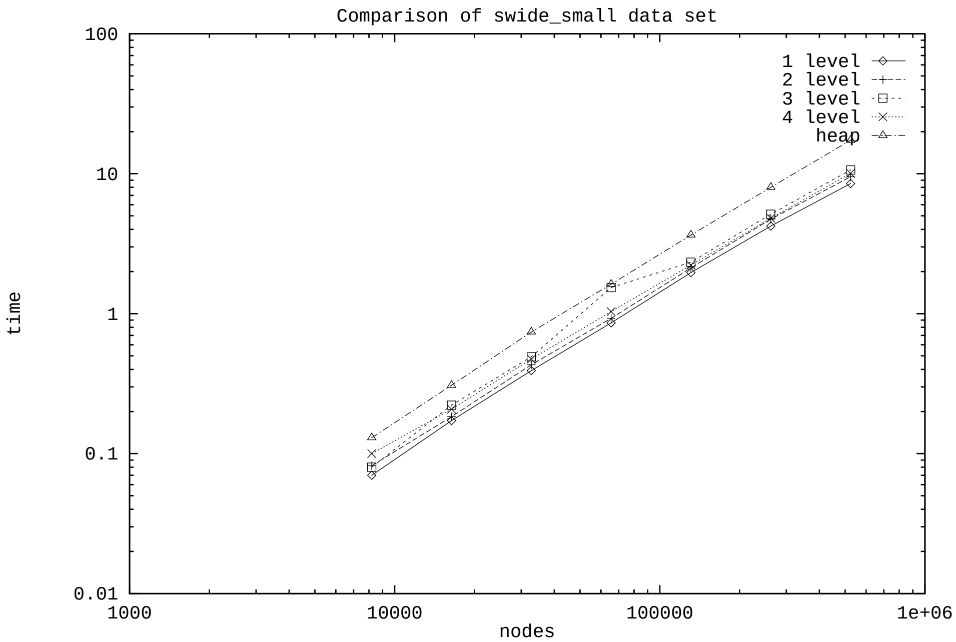
Tables 6, 7, and 8 show the relative performance of the implementations on the wide grid families WIDE-SMALL, WIDE-MEDIUM, and WIDE-LARGE. Once again, for these families $C = 16$, $C = 10,000$, and $C = 100,000,000$, respectively.

For WIDE-SMALL family, D is bounded by 256. On this family, the number of empty bucket operations is very small for all implementations and does not grow much as the problem size grows. The number of move operations is very similar for all bucket implementations. The number of expansion operations grows with the number of levels and accounts for the difference in performance. However, all implementations do less than one expansion operation per node, and the performance difference is relatively small.

For WIDE-MEDIUM networks, D is bounded by $160,000$. The number of empty bucket operations is well below the number of nodes and grows slower. The number of move operations is similar for all implementations. The number of expansion operations grows with the number of bucket levels and explains the worse performance of implementations with more bucket levels. However, even for the 4-level implementation, the number of these operations is only about two per node, and the performance difference is small.

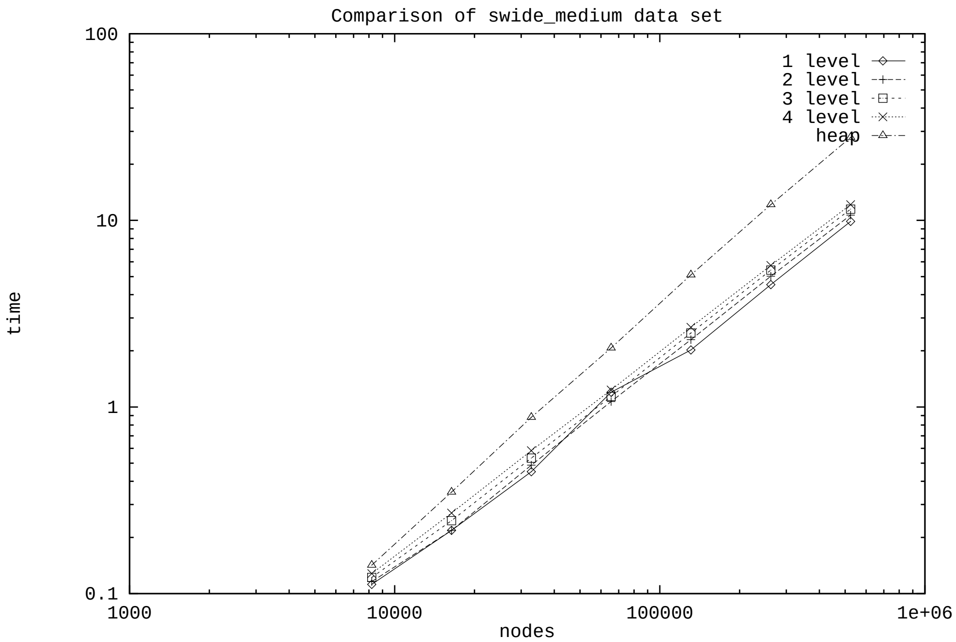
For WIDE-LARGE networks, D is bounded by $160,000,000$. For small values of n, when D is large compared to n, multi-bucket implementations with more bucket levels perform better. As n grows, so does the advantage of the 4-level implementation.

The erratic performance curve of the 1-level implementation is due to the end cutoff heuristic. The number of empty buckets seen in the 1-level case increases fitfully. For $n = 524289$, the heuristic is so successful that the 1-level implementation has less empty buckets than the 2- and



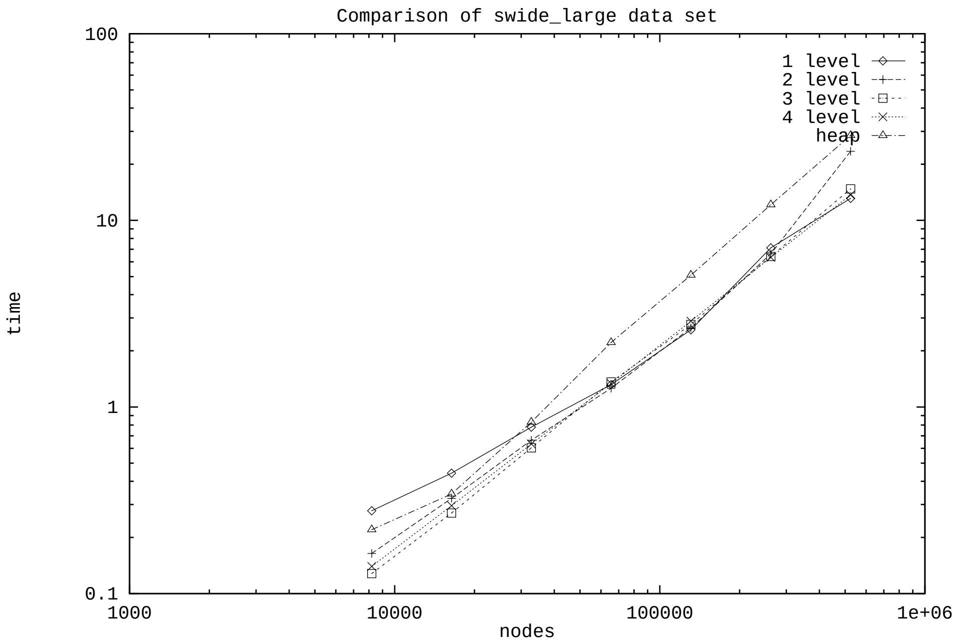
k	nodes	8193	16385	32769	65537	131073	262145	524289
1	**time**	**0.07** s	**0.17** s	**0.39** s	**0.86** s	**1.97** s	**4.22** s	**8.50** s
	empty	4	5	3	4	4	4	4
	expanded	0	0	0	0	0	0	0
	moved	10403	20774	41487	83002	166159	330764	661544
2	**time**	**0.08** s	**0.18** s	**0.43** s	**0.93** s	**2.12** s	**4.74** s	**9.53** s
	empty	3	3	2	1	2	1	1
	expanded	5512	11029	22097	44160	88293	177263	354599
	moved	9643	19206	38431	76896	153908	306714	613425
3	**time**	**0.08** s	**0.22** s	**0.49** s	**1.54** s	**2.33** s	**5.14** s	**10.65** s
	empty	2	1	1	1	0	0	1
	expanded	10341	20725	41583	83097	166210	333259	666781
	moved	9545	18996	38010	76032	152208	303497	606884
4	**time**	**0.10** s	**0.21** s	**0.47** s	**1.04** s	**2.22** s	**4.81** s	**10.05** s
	empty	2	1	1	1	0	0	1
	expanded	10975	21985	44105	88136	176287	353472	707384
	moved	9545	18996	38010	76032	152208	303497	606884
h	**time**	**0.13** s	**0.31** s	**0.74** s	**1.62** s	**3.65** s	**8.01** s	**17.37** s
	moved	10410	20779	41503	83028	166165	330765	661446

Table 6: The performance on wide grids as the grid width increases, for $C = 16$.



k	nodes	8193	16385	32769	65537	131073	262145	524289
1	**time**	**0.11** s	**0.22** s	**0.45** s	**1.20** s	**2.02** s	**4.52** s	**9.88** s
	empty	57518	61990	49721	35902	23613	16827	14476
	expanded	0	0	0	0	0	0	0
	moved	10744	21489	42888	85789	171687	341477	682841
2	**time**	**0.12** s	**0.22** s	**0.49** s	**1.07** s	**2.30** s	**5.02** s	**10.66** s
	empty	42525	45971	39080	26990	14938	9145	6749
	expanded	8065	16174	32350	64734	129455	258885	517777
	moved	10691	21399	42698	85406	170927	339971	679917
3	**time**	**0.12** s	**0.25** s	**0.53** s	**1.14** s	**2.49** s	**5.42** s	**11.50** s
	empty	25049	33588	31450	22276	11683	6427	4360
	expanded	15410	31128	62317	124658	249285	498739	997542
	moved	10373	20814	41537	83116	166301	330927	661807
4	**time**	**0.13** s	**0.27** s	**0.58** s	**1.23** s	**2.67** s	**5.73** s	**12.14** s
	empty	13299	22504	24743	18499	9580	4801	3170
	expanded	21297	43414	86899	173850	347671	696046	1391877
	moved	9907	19757	39423	78923	157931	314516	629090
h	**time**	**0.14** s	**0.35** s	**0.88** s	**2.07** s	**5.11** s	**12.14** s	**27.84** s
	moved	10744	21488	42887	85789	171689	341476	682841

Table 7: The performance on wide grids as the grid width increases, for $C = 10,000$.



k	nodes	8193	16385	32769	65537	131073	262145	524289
1	**time**	**0.28** s	**0.44** s	**0.78** s	**1.32** s	**2.59** s	**7.13** s	**13.13** s
	empty	524049	697832	919232	979496	1539567	8031971	11915214
	expanded	0	0	0	0	0	0	0
	moved	10744	21488	42889	85794	171696	341491	682753
2	**time**	**0.16** s	**0.32** s	**0.66** s	**1.26** s	**2.65** s	**6.62** s	**23.47** s
	empty	254271	439313	667675	769532	1241494	6213300	37317527
	expanded	8135	16285	32628	65255	130675	261716	523803
	moved	10722	21446	42827	85672	171505	341282	682651
3	**time**	**0.13** s	**0.27** s	**0.60** s	**1.36** s	**2.77** s	**6.41** s	**14.76** s
	empty	43746	110234	243215	423863	835803	3507279	12187754
	expanded	15858	32019	64217	128902	258380	517853	1040000
	moved	10533	21168	42306	84834	169970	338579	679055
4	**time**	**0.14** s	**0.30** s	**0.63** s	**1.34** s	**2.89** s	**6.34** s	**13.70** s
	empty	18755	41631	103074	239710	446362	1046830	3341629
	expanded	22715	45971	92010	181403	371373	765540	1531673
	moved	10114	20346	40569	80405	163378	332856	665754
h	**time**	**0.22** s	**0.34** s	**0.83** s	**2.22** s	**5.09** s	**12.13** s	**28.56** s
	moved	10744	21488	42889	85795	171697	341492	682885

Table 8: The performance on wide grids as the grid width increases, for $C = 100,000,000$.

3-level implementations. This is quite unusual.

Tables 9, 10, and 11 show the relative performance of different bucket level implementations on random graphs. The first table concerns RANDOM-SMALL networks with $C = 16$, the second RANDOM-MEDIUM networks with $C = 10,000$, and the third RANDOM-LARGE networks with $C = 100,000,000$.

For these networks, the expected value of D is proportional to $C \log n$, and the path length distribution is fairly uniform. $\log n$ is small enough that random grids perform similarly to wide grids, in which D is proportional to C.

A useful insight can be gained comparing Tables 3 and 6 for large problem sizes. The number of empty bucket operations is much higher for the long grids than the wide grids, and the numbers are similar for the other overhead operations. Yet, except for the 1-level case, the running times for long grids are better. The reason for this is that for long grids, buckets almost always contain at most one element, while wide grids usually have many elements in one bucket. As we observed in Section 6, linked list operations are faster if the former case. The list operations are used by scanning, expansion, and move operations, which on this family are much more frequent than the empty bucket operations. This explains the data.

Similar phenomena occurs in Tables 4 and 7.

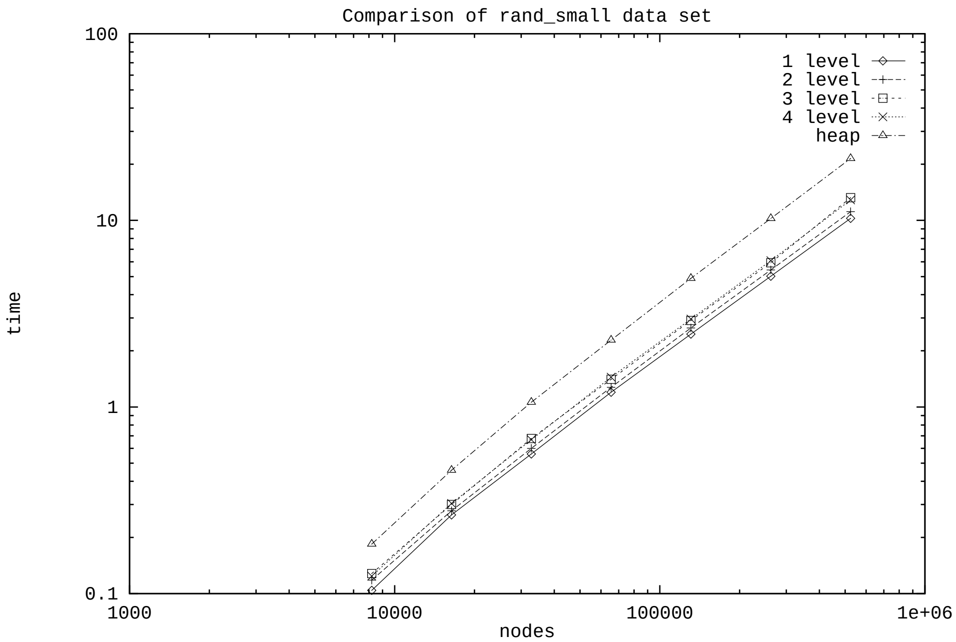
7.2 Varying the Maximum Arc Length

Tables 12, 13, and 14 show the relative performance of the implementations as the maximum arc length C changes. This is important since theoretical bounds depend on C. The tables show results for grids with $131,073$ nodes. The value of C grows starting from 1 and increasing by a factor of 10 at each step. Again, the wide grid and random graph families give similar results.

The 1-level implementation performs the best for small C, but its performance degrades quickly as C increases. This is because, as C grows, the cost of empty operations because dominant. For the LONG-LEN family, there is a clear crossover. For the WIDE-LEN and RANDOM-LEN families, the data suggests crossovers for larger values of C, and additional experiments confirm this.

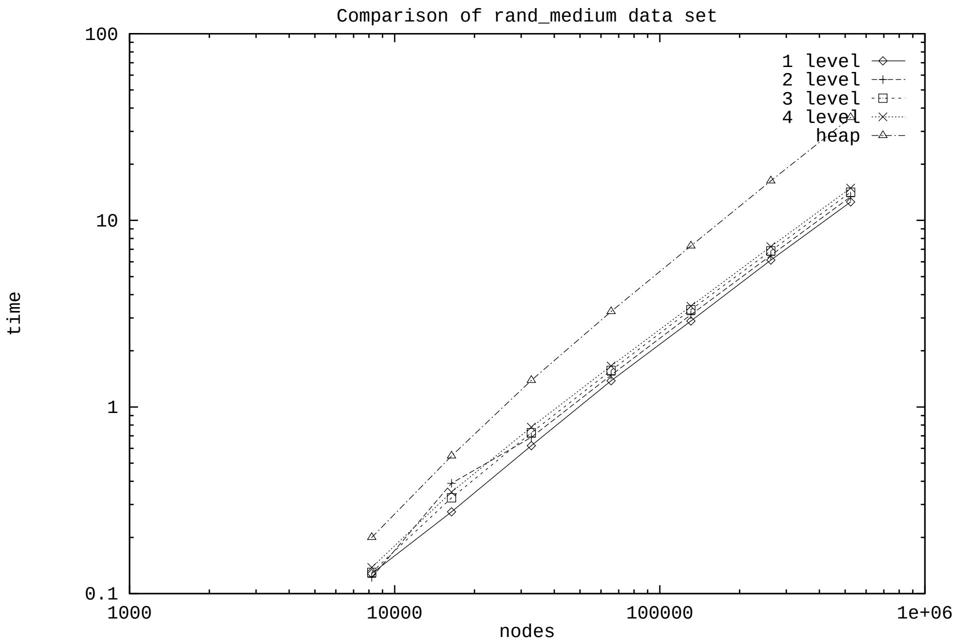
Consider the LONG-LEN family. When the number of empty bucket operations is small compared to the number of nodes, the 1-level implementation is a little faster than the multi-level implementations. For large C, the number of empty bucket operations increases, and the multi-bucket implementations are faster.

For the WIDE-LEN family, D is not much bigger than n unless C is very large. The number



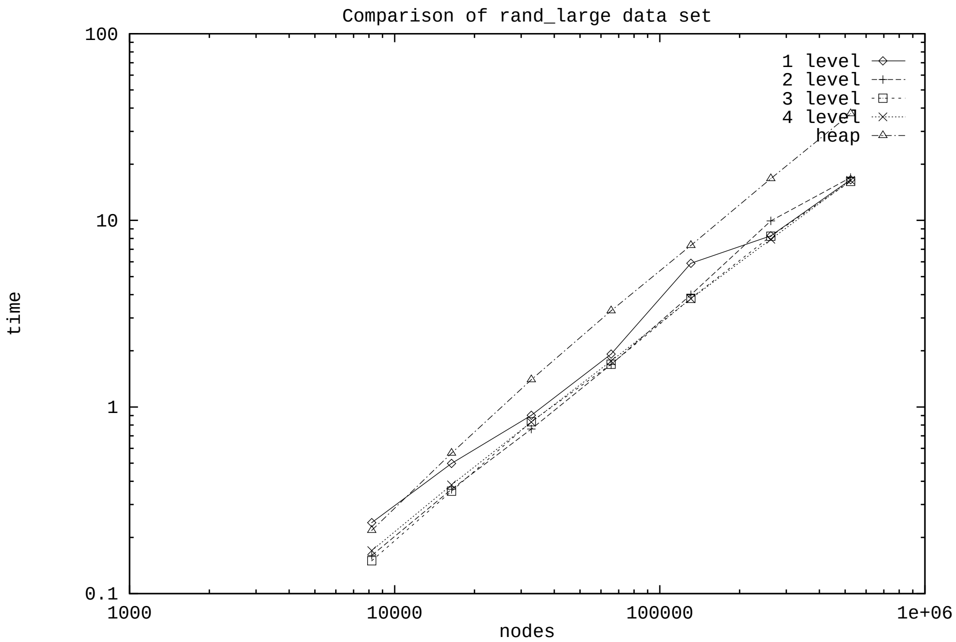
k	nodes	8192	16384	32768	65536	131072	262144	524288
1	**time**	**0.10 s**	**0.26 s**	**0.56 s**	**1.20 s**	**2.46 s**	**5.02 s**	**10.25 s**
	empty	14	11	11	9	8	8	12
	expanded	0	0	0	0	0	0	0
	moved	10510	20967	42050	84111	168108	336448	673077
2	**time**	**0.12 s**	**0.28 s**	**0.60 s**	**1.28 s**	**2.65 s**	**5.42 s**	**11.14 s**
	empty	7	8	7	6	5	5	6
	expanded	5591	11208	22430	44820	89613	179217	358153
	moved	9711	19364	38839	77658	155252	310848	621393
3	**time**	**0.13 s**	**0.30 s**	**0.68 s**	**1.41 s**	**2.91 s**	**5.93 s**	**13.24 s**
	empty	2	3	2	4	3	1	4
	expanded	10634	21457	43642	84654	170499	329260	685554
	moved	9606	19069	38141	76958	153417	308969	612510
4	**time**	**0.12 s**	**0.30 s**	**0.67 s**	**1.45 s**	**2.96 s**	**6.09 s**	**12.87 s**
	empty	2	3	2	3	3	1	4
	expanded	10736	21827	44707	90755	184001	371878	876093
	moved	9606	19069	38141	76958	153417	308969	612510
h	**time**	**0.18 s**	**0.46 s**	**1.06 s**	**2.28 s**	**4.89 s**	**10.25 s**	**21.46 s**
	moved	10513	20960	42057	84088	168077	336425	673051

Table 9: The performance on random graphs as n increases, for $C = 16$.



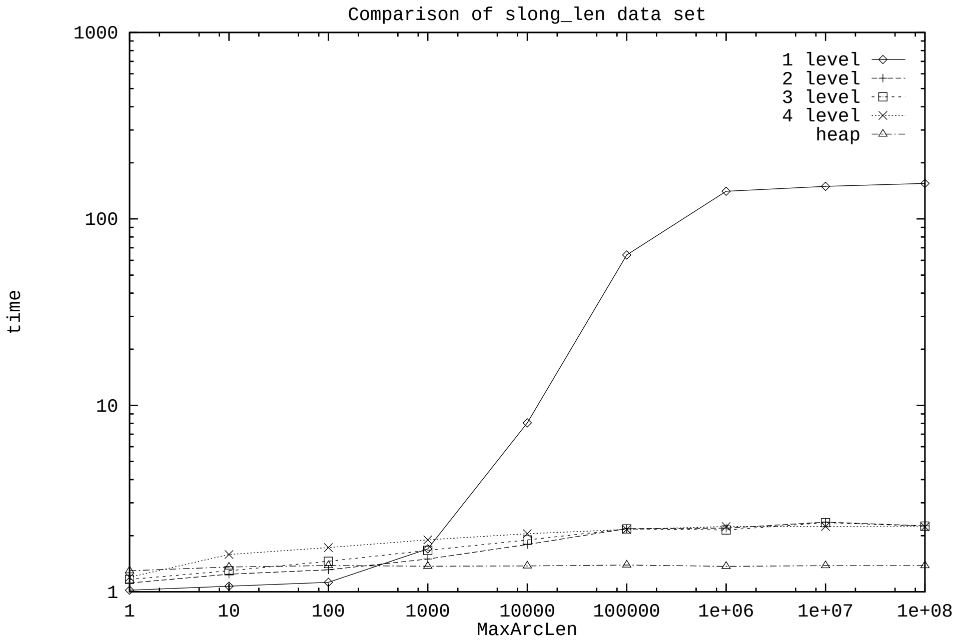
k	nodes	8192	16384	32768	65536	131072	262144	524288
1	**time**	**0.13** s	**0.27** s	**0.62** s	**1.38** s	**2.89** s	**6.12** s	**12.56** s
	empty	40163	37693	36205	35860	33829	33747	38580
	expanded	0	0	0	0	0	0	0
	moved	10941	21879	43880	87646	175335	350745	701720
2	**time**	**0.12** s	**0.39** s	**0.69** s	**1.48** s	**3.12** s	**6.50** s	**13.39** s
	empty	16447	15860	15143	14469	14127	13937	14252
	expanded	8084	16172	32332	64671	129356	258761	517449
	moved	10883	21765	43654	87213	174469	348974	698169
3	**time**	**0.13** s	**0.33** s	**0.73** s	**1.57** s	**3.32** s	**6.86** s	**14.15** s
	empty	9877	9630	8931	8361	8187	7985	8062
	expanded	15552	31129	62223	124465	248896	497852	995570
	moved	10568	21095	42363	84560	169185	338459	676980
4	**time**	**0.14** s	**0.35** s	**0.78** s	**1.66** s	**3.47** s	**7.24** s	**14.90** s
	empty	6754	6780	6254	5671	5536	5453	5385
	expanded	21751	43595	87115	174153	348326	696759	1392883
	moved	10013	19985	40084	80069	160259	320395	641105
h	**time**	**0.20** s	**0.55** s	**1.39** s	**3.25** s	**7.30** s	**16.29** s	**35.60** s
	moved	10941	21878	43881	87647	175336	350747	701717

Table 10: The performance on random graphs as n increases, for $C = 10,000$.



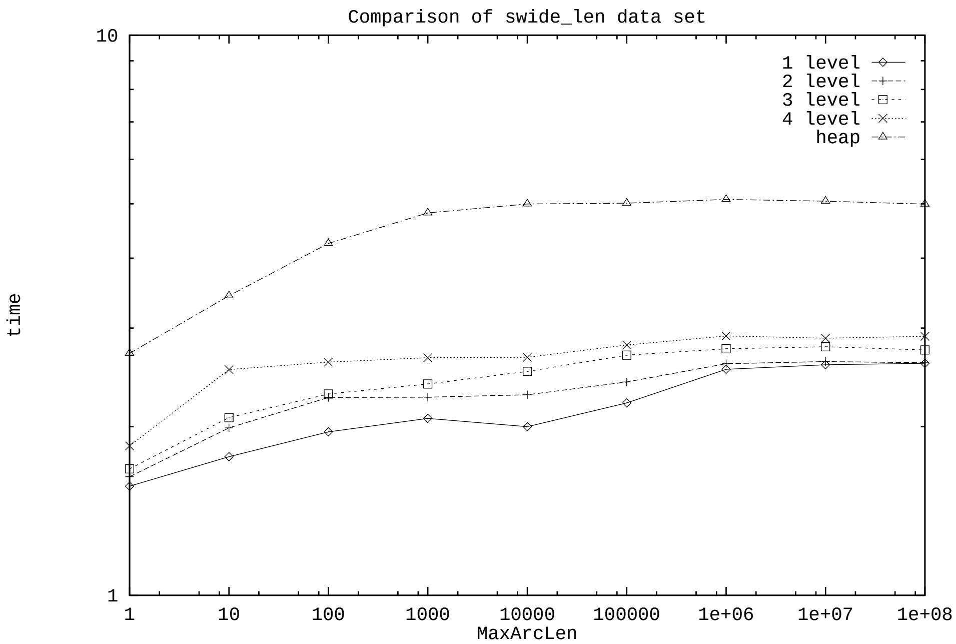
k	nodes	8192	16384	32768	65536	131072	262144	524288
1	**time**	**0.24** s	**0.50** s	**0.90** s	**1.92** s	**5.89** s	**8.27** s	**16.52** s
	empty	324376	576850	784427	1720162	8848367	7743432	15031125
	expanded	0	0	0	0	0	0	0
	moved	10941	21882	43882	87657	175355	350664	701551
2	**time**	**0.16** s	**0.36** s	**0.76** s	**1.69** s	**3.99** s	**9.95** s	**16.98** s
	empty	111362	227866	320028	721023	3528171	13295674	13223005
	expanded	8138	16309	32638	65331	130888	261886	523892
	moved	10914	21848	43819	87554	175267	350632	701572
3	**time**	**0.15** s	**0.35** s	**0.83** s	**1.69** s	**3.82** s	**8.22** s	**16.19** s
	empty	43605	95448	153918	371270	1785388	4788428	6513717
	expanded	15874	32079	64543	129116	258766	519199	1040105
	moved	10700	21539	43391	86664	173649	348194	697449
4	**time**	**0.17** s	**0.38** s	**0.83** s	**1.76** s	**3.80** s	**7.92** s	**16.38** s
	empty	16138	51614	98956	176341	641185	1430850	2874672
	expanded	23617	45975	91525	187079	384445	770991	1544200
	moved	10526	20627	41155	83678	171155	343289	687877
h	**time**	**0.22** s	**0.57** s	**1.40** s	**3.28** s	**7.34** s	**16.75** s	**37.34** s
	moved	10941	21882	43883	87658	175355	350770	701767

Table 11: The performance on random graphs as n increases, for $C = 100,000,000$.



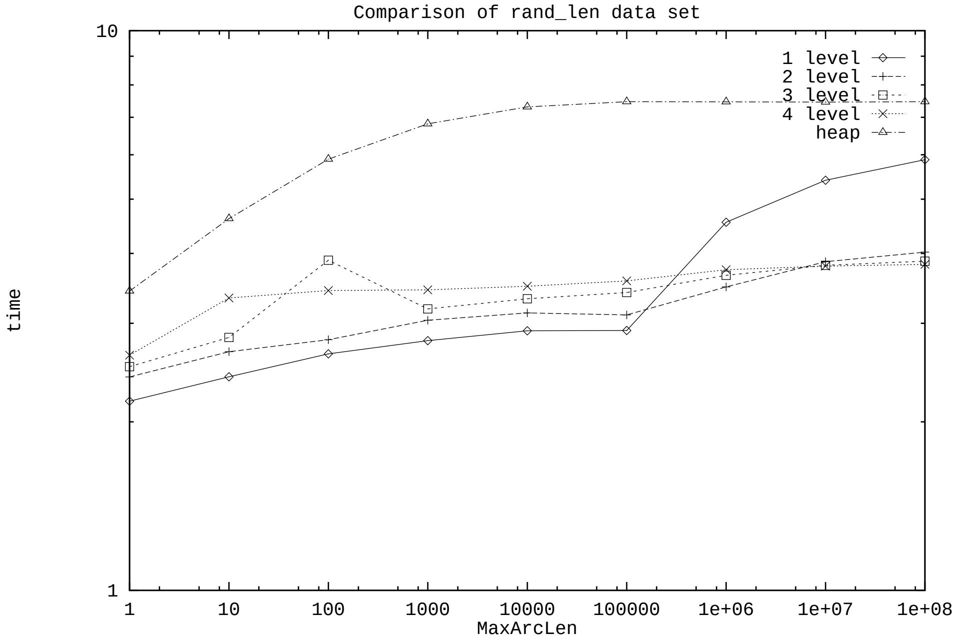
k	MaxArcLen	1	10	100	1000	10000	100000	1000000	9999994	99999937
1	**time**	**1.02 s**	**1.07 s**	**1.12 s**	**1.70 s**	**8.06 s**	**64.10 s**	**140.62 s**	**149.37 s**	**154.92 s**
	empty	0	344	162140	2459807	25974606	233725061	516226971	530194171	557755127
	expanded	0	0	0	0	0	0	0	0	0
	moved	131072	162913	170043	170841	170925	170931	170931	170930	170931
2	**time**	**1.11 s**	**1.24 s**	**1.31 s**	**1.50 s**	**1.79 s**	**2.18 s**	**2.20 s**	**2.37 s**	**2.26 s**
	empty	0	186	113303	488015	1377759	2637060	2351276	3214423	2641065
	expanded	65536	101610	114198	127298	129491	130433	130711	130584	130678
	moved	131072	153029	162976	169090	170201	170639	170769	170708	170754
3	**time**	**1.17 s**	**1.30 s**	**1.46 s**	**1.67 s**	**1.90 s**	**2.17 s**	**2.15 s**	**2.35 s**	**2.25 s**
	empty	0	186	67042	210032	505943	1109200	912364	1548074	1213987
	expanded	98304	140900	199299	233169	249828	256194	259315	257048	258528
	moved	131072	150271	155013	160228	165671	168227	169631	168612	169274
4	**time**	**1.20 s**	**1.58 s**	**1.73 s**	**1.90 s**	**2.05 s**	**2.16 s**	**2.24 s**	**2.24 s**	**2.23 s**
	empty	0	0	24133	111558	237499	366030	439919	446718	445451
	expanded	114688	295963	318137	337675	349963	363434	372394	370211	372381
	moved	131072	147261	153579	155821	157385	160343	162777	162417	162887
h	**time**	**1.30 s**	**1.36 s**	**1.38 s**	**1.37 s**	**1.38 s**	**1.39 s**	**1.37 s**	**1.38 s**	**1.38 s**
	moved	131072	162877	170055	170842	170922	170930	170931	170931	170931

Table 12: The performance on long grids as the maximum arc length increases. $n = 131072$.



k	MaxArcLen	1	10	100	1000	10000	100000	1000000	9999994	99999937
1	**time**	**1.57 s**	**1.77 s**	**1.96 s**	**2.07 s**	**2.00 s**	**2.21 s**	**2.53 s**	**2.58 s**	**2.60 s**
	empty	0	2	44	974	23613	581673	1424789	1451039	1539567
	expanded	0	0	0	0	0	0	0	0	0
	moved	131072	163173	170785	171605	171687	171696	171697	171697	171696
2	**time**	**1.63 s**	**1.99 s**	**2.26 s**	**2.26 s**	**2.28 s**	**2.40 s**	**2.59 s**	**2.61 s**	**2.60 s**
	empty	0	0	19	375	14938	479411	1133298	1186518	1241494
	expanded	65536	100158	113393	127149	129455	130404	130704	130562	130675
	moved	131072	153220	163569	169800	170927	171392	171514	171457	171505
3	**time**	**1.68 s**	**2.08 s**	**2.29 s**	**2.38 s**	**2.51 s**	**2.68 s**	**2.76 s**	**2.78 s**	**2.74 s**
	empty	0	0	11	248	11683	373940	719023	853078	835803
	expanded	98304	135691	195716	231747	249285	255843	259176	256798	258380
	moved	131072	150718	155661	160711	166301	168958	170357	169320	169970
4	**time**	**1.85 s**	**2.53 s**	**2.61 s**	**2.66 s**	**2.66 s**	**2.80 s**	**2.90 s**	**2.88 s**	**2.90 s**
	empty	0	0	5	154	9580	287454	450945	427626	446362
	expanded	114688	290817	313946	334568	347671	361765	371385	368992	371373
	moved	131072	147703	154192	156160	157931	160768	163348	162963	163378
h	**time**	**2.70 s**	**3.43 s**	**4.25 s**	**4.82 s**	**5.00 s**	**5.01 s**	**5.09 s**	**5.05 s**	**4.99 s**
	moved	131072	163181	170796	171600	171689	171696	171697	171697	171697

Table 13: The performance on wide grids as the maximum arc length increases. $n = 131072$.



k	MaxArcLen	1	10	100	1000	10000	100000	1000000	9999998	99999973
1	**time**	**2.18 s**	**2.41 s**	**2.65 s**	**2.79 s**	**2.91 s**	**2.91 s**	**4.55 s**	**5.40 s**	**5.88 s**
	empty	0	3	115	2186	33829	459628	3440649	7113429	8848367
	expanded	0	0	0	0	0	0	0	0	0
	moved	131071	164293	174122	175240	175335	175354	175355	175355	175355
2	**time**	**2.41 s**	**2.67 s**	**2.80 s**	**3.04 s**	**3.13 s**	**3.11 s**	**3.48 s**	**3.87 s**	**4.02 s**
	empty	0	2	52	812	14127	224907	1496248	2948066	3528171
	expanded	63492	100962	113140	126996	129356	130357	130844	130836	130888
	moved	131071	153960	165961	173164	174469	175016	175244	175242	175267
3	**time**	**2.51 s**	**2.83 s**	**3.89 s**	**3.18 s**	**3.32 s**	**3.41 s**	**3.66 s**	**3.81 s**	**3.87 s**
	empty	0	2	32	543	8187	129347	828194	1569370	1785388
	expanded	99713	146210	197705	231714	248896	256684	259070	257949	258766
	moved	131071	151261	157817	163191	169185	172669	173811	173252	173649
4	**time**	**2.63 s**	**3.33 s**	**3.43 s**	**3.44 s**	**3.49 s**	**3.57 s**	**3.74 s**	**3.80 s**	**3.82 s**
	empty	0	0	14	317	5536	98881	383948	624711	641185
	expanded	135537	294636	315855	335732	348326	356897	384596	382361	384445
	moved	131071	148090	156053	158628	160259	161839	171309	170225	171155
h	**time**	**3.43 s**	**4.61 s**	**5.89 s**	**6.81 s**	**7.31 s**	**7.47 s**	**7.46 s**	**7.45 s**	**7.46 s**
	moved	131071	164270	174121	175242	175336	175354	175355	175355	175355

Table 14: The performance on random graphs as the maximum arc length increases. $n = 131072$.

of empty bucket operations grows with C but remains below the number of nodes for $C \leq 1,000$. Although the number of empty bucket operations decreases with the number of bucket levels, this dependence is much less than for the LONG-LEN family.

The RAND-LEN family is similar to the WIDE-LEN family.

Comparison to the heap implementation shows that, except for the 1-level implementation, the bucket implementations are not much more sensitive to C.

7.3 Varying Grid Size and Maximum Arc Length

A natural experiment is to make C proportional to n. Tables 15 and 16 show data for LONG-L and LONG-L10 families of long grids. For the former family, $C = x$ (the length of the grid). In the second table, $C = x/10$. Tables 17 and 18 show equivalent results for WIDE-L and WIDE-10L families, and tables 19 and 20 show results for the RAND-L and RAND-10L families.

For long grids, the expected value of D grows as n^2. The 1-level bucket implementation is the worst by a large margin. The 2-, 3-, and 4-level implementations perform similarly. For the 1-level implementation, the growth in the number of empty bucket operations is close to quadratic, as the theory suggests. For large values of n, these operations dominate the running time. For multi-level implementations, the growth rate is much slower (although superlinear). As usual, more levels decrease the number of empty bucket operations but increase the number of expansion operations. However, the performance differences among multi-level implementations are relatively small.

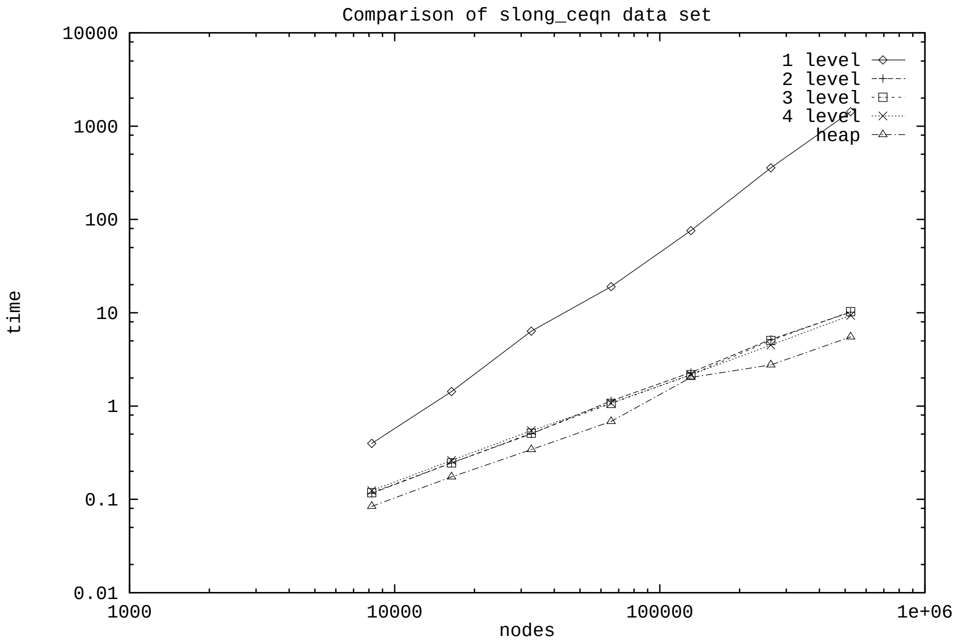
For wide grids, the expected value of D is linear in n. All bucket implementations perform similarly, although the implementations with fewer levels are a little faster. The number of overhead operations is comparable to the number of nodes.

For random graphs, the expected value of D is linear in $n \log n$, and the results are similar to those for wide grids.

7.4 Hard and Easy Problems

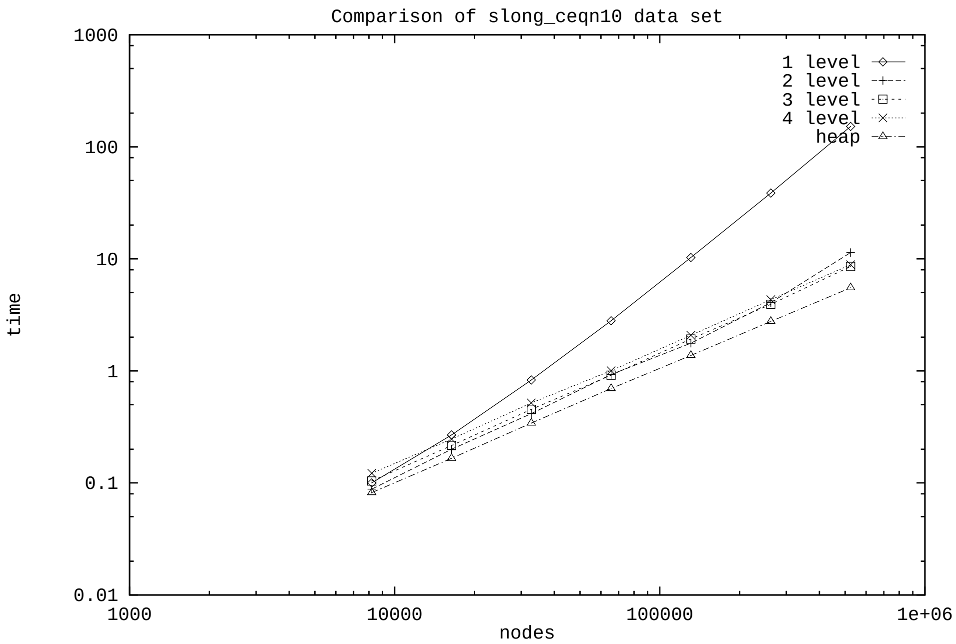
Bucket implementations with more than one level perform similarly on problems discussed in previous sections. Next we study problems designed to be hard or easy for an implementation with a specific number of bucket levels.

Recall that the graphs we use for these problems consist of the source connected to two paths with an equal number of arcs. Suppose that the problem is designed for an implementation that uses k bucket levels. The path arcs have the same length equal to the number p of buckets at the



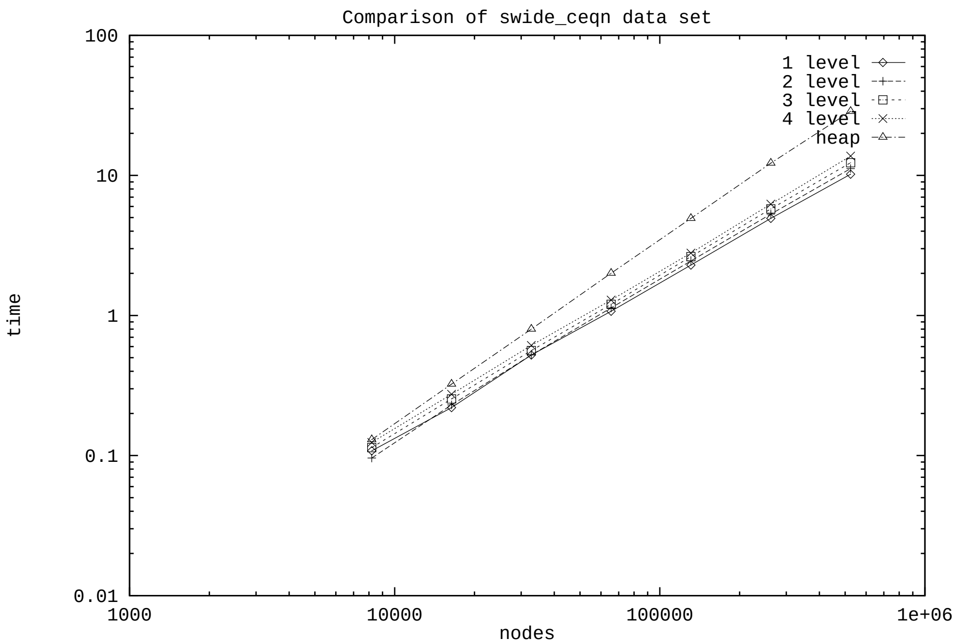
k	nodes	8193	16385	32769	65537	131073	262145	524289
1	**time**	**0.40** s	**1.43** s	**6.36** s	**19.07** s	**76.01** s	**357.92** s	**1432.18** s
	empty	1202520	4781164	21511738	68626877	274515736	1236191488	1517531420
	expanded	0	0	0	0	0	0	0
	moved	10669	21367	42783	85489	170931	341989	683817
2	**time**	**0.12** s	**0.25** s	**0.50** s	**1.13** s	**2.29** s	**5.17** s	**10.13** s
	empty	101272	241520	502656	1538746	3015827	8116176	14061592
	expanded	8046	16184	32524	65042	130387	261201	523123
	moved	10603	21277	42672	85263	170614	341543	683278
3	**time**	**0.12** s	**0.25** s	**0.51** s	**1.07** s	**2.17** s	**5.05** s	**10.31** s
	empty	28832	67496	235616	530607	1080725	4102953	8562016
	expanded	15281	31611	61862	125867	256803	510723	1030301
	moved	10228	20860	41247	83222	168511	335874	675430
4	**time**	**0.12** s	**0.26** s	**0.55** s	**1.08** s	**2.17** s	**4.51** s	**9.35** s
	empty	15550	28375	60158	173225	406687	923431	2168228
	expanded	20962	44972	94433	178985	357281	751631	1536325
	moved	9818	19953	41237	80015	159067	327811	667911
h	**time**	**0.08** s	**0.17** s	**0.34** s	**0.68** s	**2.03** s	**2.76** s	**5.52** s
	moved	10669	21367	42783	85488	170930	341989	683817

Table 15: The performance on long grids as the grid length and the maximum arc length grow together. $C = x$.



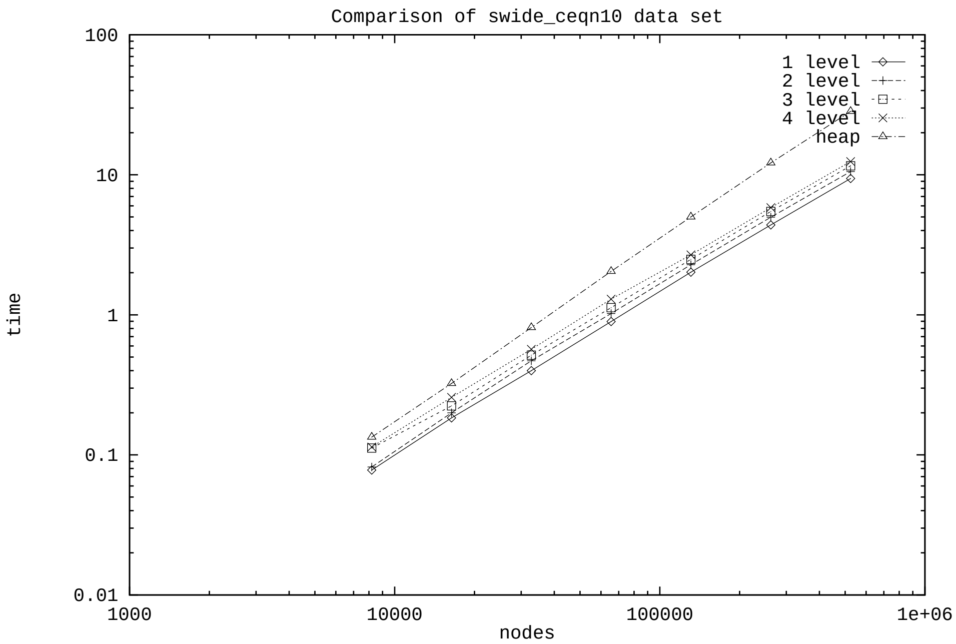
k	nodes	8193	16385	32769	65537	131073	262145	524289
1	**time**	**0.10** s	**0.27** s	**0.83** s	**2.80** s	**10.28** s	**38.70** s	**151.84** s
	empty	126164	517213	2104235	8440398	33868207	135703564	544664734
	expanded	0	0	0	0	0	0	0
	moved	10663	21363	42775	85483	170926	341985	683815
2	**time**	**0.09** s	**0.20** s	**0.42** s	**0.93** s	**1.77** s	**4.07** s	**11.38** s
	empty	31384	132799	213999	852984	1207835	4541666	6412881
	expanded	7890	15793	32180	64354	129860	259733	521876
	moved	10527	21086	42506	84928	170376	340865	682683
3	**time**	**0.10** s	**0.22** s	**0.45** s	**0.92** s	**1.93** s	**3.93** s	**8.55** s
	empty	13026	27711	63947	218912	540698	1080983	4409066
	expanded	14219	30420	63076	121990	252659	514540	1011132
	moved	9928	20432	41764	81752	166783	337574	667191
4	**time**	**0.12** s	**0.25** s	**0.52** s	**1.01** s	**2.08** s	**4.32** s	**8.88** s
	empty	6842	13853	27467	128023	227402	459851	1043327
	expanded	20470	44465	93237	166814	358514	748654	1533453
	moved	9702	19827	40910	78129	158962	326977	667128
h	**time**	**0.08** s	**0.17** s	**0.34** s	**0.70** s	**1.37** s	**2.76** s	**5.52** s
	moved	10663	21362	42775	85482	170927	341986	683813

Table 16: The performance on long grids as the grid length and the maximum arc length grow together. $C = x/10$.



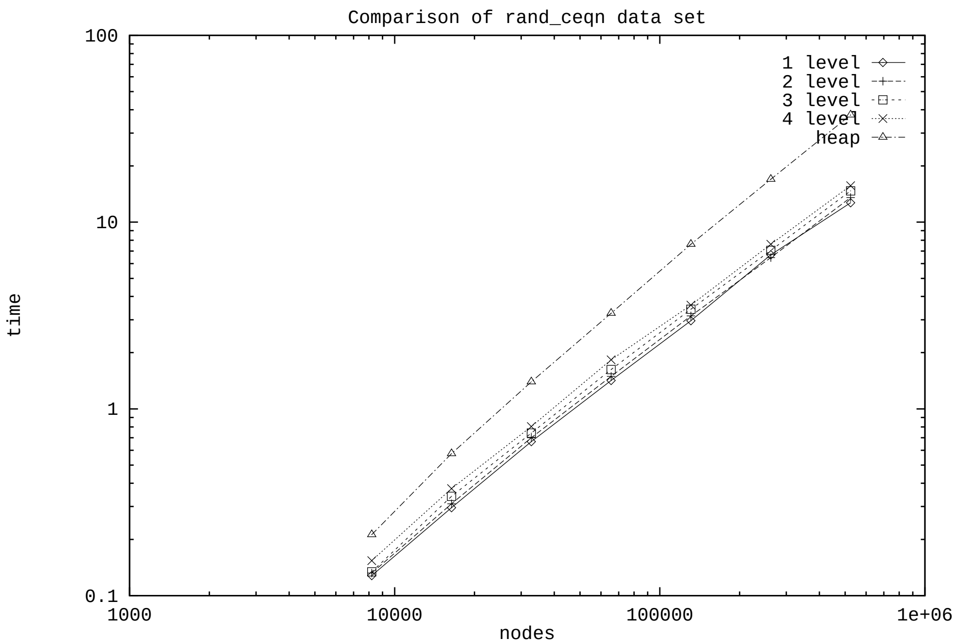
k	nodes	8193	16385	32769	65537	131073	262145	524289
1	**time**	**0.11** s	**0.22** s	**0.52** s	**1.07** s	**2.29** s	**4.94** s	**10.22** s
	empty	45690	97796	215440	340159	695868	1618779	3262258
	expanded	0	0	0	0	0	0	0
	moved	10743	21489	42889	85794	171696	341492	682884
2	**time**	**0.10** s	**0.23** s	**0.52** s	**1.13** s	**2.45** s	**5.30** s	**11.20** s
	empty	35036	73094	171475	281937	572086	1366719	2741423
	expanded	8033	16179	32519	65039	130364	261133	523078
	moved	10675	21398	42770	85561	171358	341029	682350
3	**time**	**0.11** s	**0.25** s	**0.56** s	**1.20** s	**2.63** s	**5.77** s	**12.32** s
	empty	22372	44881	133782	214406	429484	1118349	2264334
	expanded	15215	31558	61725	125622	256586	510218	1029673
	moved	10294	20984	41304	83467	169226	335315	674513
4	**time**	**0.12** s	**0.27** s	**0.61** s	**1.29** s	**2.79** s	**6.24** s	**13.75** s
	empty	12798	25572	54125	149542	325912	715430	1432781
	expanded	20746	44784	94291	178066	355332	750138	1534999
	moved	9860	20035	41293	80178	159545	327257	667021
h	**time**	**0.13** s	**0.32** s	**0.80** s	**2.01** s	**4.93** s	**12.24** s	**28.69** s
	moved	10744	21489	42889	85794	171697	341492	682884

Table 17: The performance on wide grids as the grid width and the maximum arc length grow together. $C = y$.



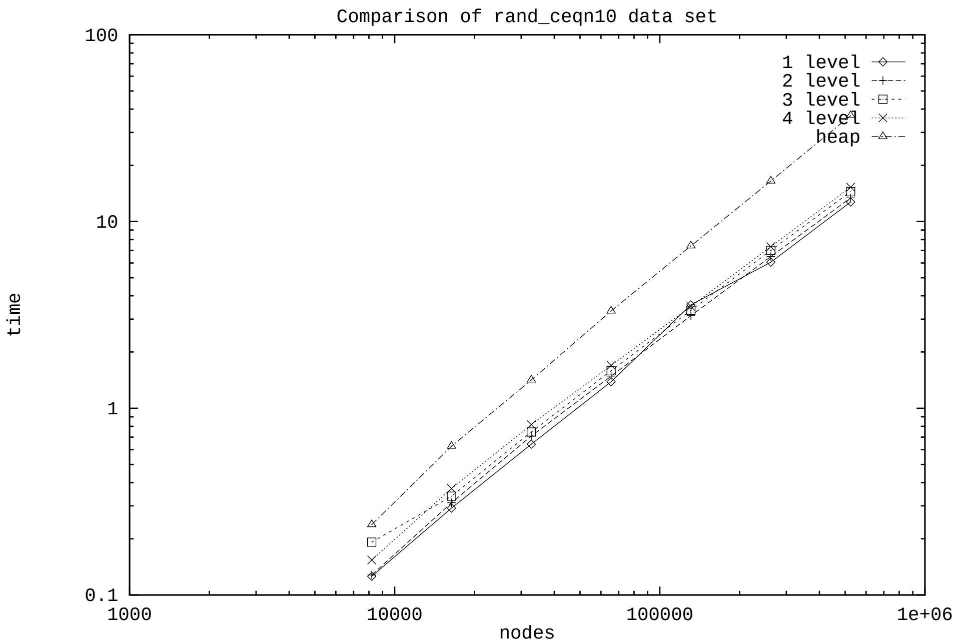
k	nodes	8193	16385	32769	65537	131073	262145	524289
1	**time**	**0.08** s	**0.18** s	**0.40** s	**0.90** s	**2.02** s	**4.39** s	**9.41** s
	empty	1954	4208	8418	17375	36746	78160	163252
	expanded	0	0	0	0	0	0	0
	moved	10738	21481	42883	85788	171689	341487	682879
2	**time**	**0.08** s	**0.20** s	**0.47** s	**1.02** s	**2.28** s	**4.98** s	**10.56** s
	empty	1283	2818	5673	12607	25433	57081	113240
	expanded	7875	15751	32143	64308	129828	259649	521777
	moved	10612	21212	42608	85221	171099	340339	681757
3	**time**	**0.11** s	**0.22** s	**0.52** s	**1.13** s	**2.49** s	**5.48** s	**11.59** s
	empty	1055	2072	4151	10149	20455	42051	95572
	expanded	14103	30273	62983	121562	252227	514202	1009685
	moved	9972	20525	41818	81990	167479	337047	666319
4	**time**	**0.11** s	**0.26** s	**0.57** s	**1.30** s	**2.69** s	**5.84** s	**12.43** s
	empty	785	1516	3057	8432	16963	34765	69667
	expanded	20249	44193	92997	165155	356820	747187	1531935
	moved	9766	19894	40967	78465	159408	326432	666258
h	**time**	**0.13** s	**0.32** s	**0.81** s	**2.05** s	**5.01** s	**12.18** s	**28.40** s
	moved	10737	21483	42884	85788	171689	341487	682879

Table 18: The performance on wide grids as the grid width and the maximum arc length grow together. $C = y/10$.



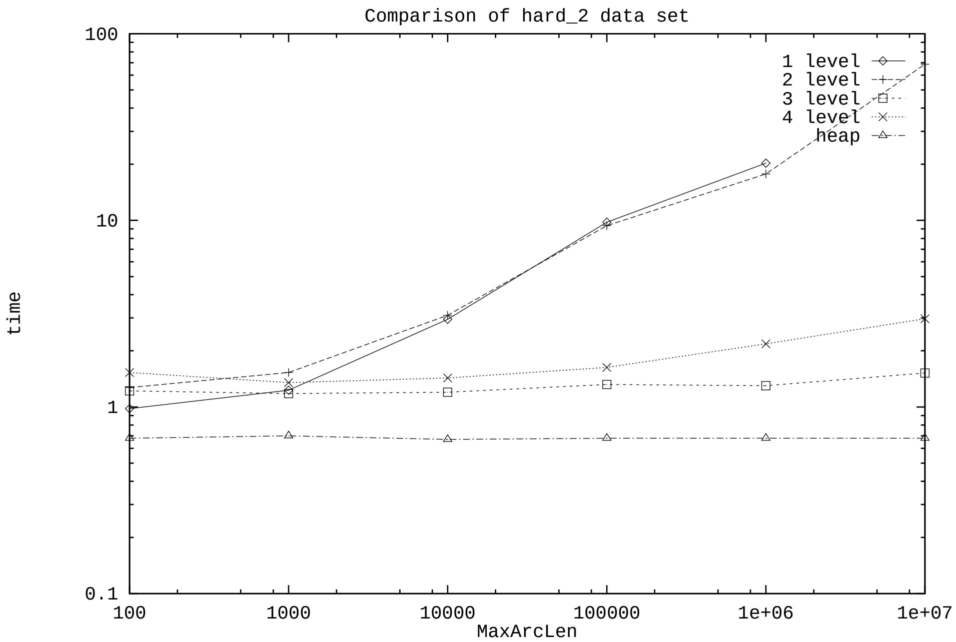
k	nodes	8192	16384	32768	65536	131072	262144	524288
1	**time**	**0.13** s	**0.30** s	**0.67** s	**1.42** s	**2.97** s	**6.68** s	**12.70** s
	empty	31772	68466	146869	310302	638538	1343418	3155034
	expanded	0	0	0	0	0	0	0
	moved	10940	21882	43883	87656	175354	350771	701767
2	**time**	**0.13** s	**0.31** s	**0.70** s	**1.49** s	**3.15** s	**6.46** s	**13.55** s
	empty	13452	31866	64619	150381	299076	682778	1366776
	expanded	8065	16124	32496	65001	130534	261073	523236
	moved	10872	21745	43748	87372	175083	350237	701221
3	**time**	**0.13** s	**0.34** s	**0.74** s	**1.62** s	**3.42** s	**7.05** s	**14.70** s
	empty	8180	16319	43217	86128	171769	426854	854375
	expanded	15400	31739	61633	126959	257958	508030	1031856
	moved	10498	21372	42096	85655	173253	342807	693324
4	**time**	**0.15** s	**0.37** s	**0.80** s	**1.83** s	**3.59** s	**7.62** s	**15.67** s
	empty	5670	11252	22556	65347	130828	260415	521462
	expanded	21303	45372	94179	171629	364060	754591	1539446
	moved	9973	20396	42080	79929	163556	336303	685667
h	**time**	**0.21** s	**0.58** s	**1.40** s	**3.25** s	**7.61** s	**16.97** s	**37.48** s
	moved	10940	21881	43883	87656	175354	350771	701766

Table 19: The performance on random graphs as the grid width and the maximum arc length grow together. $C = y$.



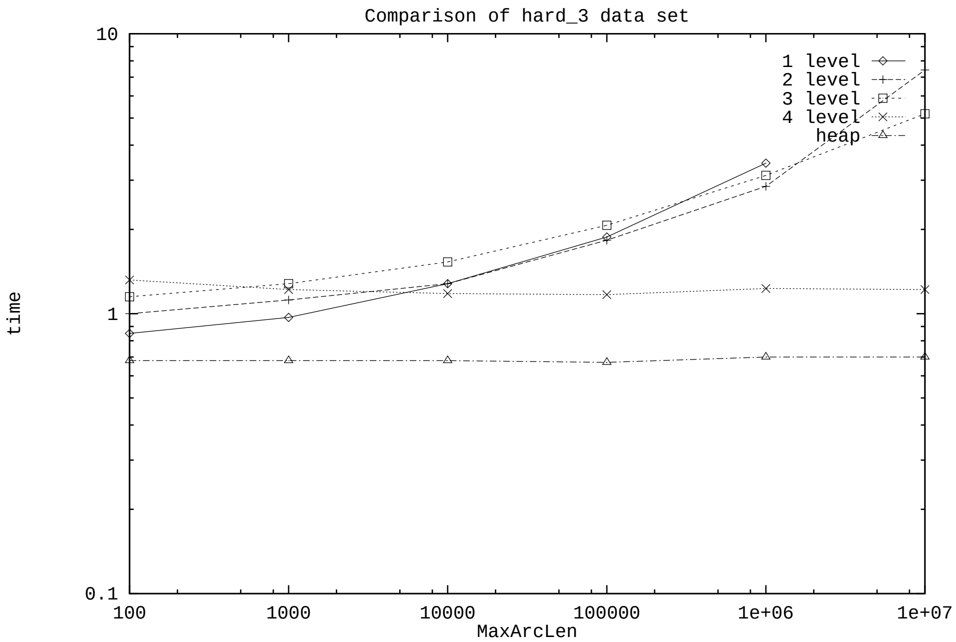
k	nodes	8192	16384	32768	65536	131072	262144	524288
1	**time**	**0.13** s	**0.29** s	**0.64** s	**1.39** s	**3.58** s	**6.06** s	**12.72** s
	empty	2097	4576	9881	22076	44989	100776	245506
	expanded	0	0	0	0	0	0	0
	moved	10927	21869	43878	87644	175342	350759	701758
2	**time**	**0.13** s	**0.31** s	**0.71** s	**1.49** s	**3.13** s	**6.49** s	**13.39** s
	empty	681	1819	3657	9211	18745	44764	91157
	expanded	7882	15748	32126	64241	129761	259524	521637
	moved	10776	21548	43540	86969	174679	349428	700382
3	**time**	**0.19** s	**0.34** s	**0.75** s	**1.59** s	**3.35** s	**7.00** s	**14.45** s
	empty	448	906	1820	5445	10742	21384	58362
	expanded	14124	30297	62888	121365	251878	513714	1008071
	moved	10100	20788	42671	83337	170474	345574	682181
4	**time**	**0.15** s	**0.37** s	**0.82** s	**1.70** s	**3.51** s	**7.33** s	**15.27** s
	empty	275	533	1087	3659	7254	14408	29163
	expanded	20383	44272	92933	165955	356957	746071	1530201
	moved	9896	20149	41684	79647	161881	333472	682081
h	**time**	**0.24** s	**0.63** s	**1.42** s	**3.32** s	**7.41** s	**16.49** s	**36.87** s
	moved	10928	21870	43878	87644	175344	350758	701756

Table 20: The performance on random graphs as the grid width and the maximum arc length grow together. $C = y/10$.



k	MaxArcLen	100	1000	10000	100000	1000000	10000000
1	**time**	**0.98** s	**1.23** s	**2.95** s	**9.78** s	**20.27** s	
	empty	917488	1966048	8257408	33422848	66976768	
	expanded	0	0	0	0	0	
	moved	131072	131072	131072	131072	131072	
2	**time**	**1.27** s	**1.53** s	**3.10** s	**9.37** s	**17.75** s	**68.68** s
	empty	917488	1966048	8257408	33422848	66976768	268300288
	expanded	131069	131069	131069	131069	131069	131069
	moved	131072	131072	131072	131072	131072	131072
3	**time**	**1.22** s	**1.18** s	**1.20** s	**1.32** s	**1.30** s	**1.52** s
	empty	3	7	131085	393209	393209	917489
	expanded	163837	147453	147453	147453	139261	139261
	moved	131072	131072	131072	131072	131072	131072
4	**time**	**1.53** s	**1.35** s	**1.43** s	**1.63** s	**2.18** s	**2.97** s
	empty	131069	131069	393209	917489	1966049	4063169
	expanded	294906	204796	200700	198652	266234	264186
	moved	131072	131072	131072	131072	131072	131072
h	**time**	**0.68** s	**0.70** s	**0.67** s	**0.68** s	**0.68** s	**0.68** s
	moved	131072	131072	131072	131072	131072	131072

Table 21: Hard problems for the 2-level implementation. $n = 131072$.



k	MaxArcLen	100	1000	10000	100000	1000000	10000000
1	**time**	**0.85** s	**0.97** s	**1.28** s	**1.88** s	**3.45** s	
	empty	393208	917488	1966048	4063168	8257408	
	expanded	0	0	0	0	0	
	moved	131072	131072	131072	131072	131072	
2	**time**	**1.00** s	**1.12** s	**1.28** s	**1.83** s	**2.85** s	**7.43** s
	empty	393208	917488	1966048	4063168	8257408	16645888
	expanded	65535	65535	32767	16383	16383	8191
	moved	131072	131072	131072	131072	131072	131072
3	**time**	**1.15** s	**1.28** s	**1.53** s	**2.07** s	**3.12** s	**5.18** s
	empty	393208	917488	1966048	4063168	8257408	16645888
	expanded	147452	139260	135164	133116	132092	131580
	moved	131072	131072	131072	131072	131072	131072
4	**time**	**1.32** s	**1.22** s	**1.18** s	**1.17** s	**1.23** s	**1.22** s
	empty	2	6	14	30	131069	131069
	expanded	212988	167932	148476	139516	147964	139388
	moved	131072	131072	131072	131072	131072	131072
h	**time**	**0.68** s	**0.68** s	**0.68** s	**0.67** s	**0.70** s	**0.70** s
	moved	131072	131072	131072	131072	131072	131072

Table 22: Hard problems for the 3-level implementation. $n = 131072$.

lowest level. The arcs out of the source have different length. For hard problems, this length is 0 and $p-1$; for easy problems, the length is 0 and 1. In the case of hard problems, the k-level algorithm never examines an empty bucket not at the lowest level, and on the lowest level the algorithm always examines all buckets even though only the first and the last ones are occupied. In the case of easy problems, the algorithm examines the first two buckets at the lowest level and skips the rest because of the end cutoff heuristic.

One expects hard problems to take a long time because the k-level implementation performs $p-2$ empty bucket operation for every two node scans. Recall that in our implementations the number of buckets at each level is a power of two. As a result, a multi-level implementation with less than k levels is also forced to examine all lowest level buckets, and the number of nodes in the lowest level buckets at any point of the execution is small. Therefore a problem designed to be hard for the k-level implementation is hard for implementations with fewer levels as well.

Easy problems take relatively little time because no empty buckets are examined by the corresponding implementation.

Tables 21 and 22 give data for problem families HARD-2 and HARD-3 designed to be hard for 2- and 3-level implementations, respectively. The results are as expected. For the HARD-2 family and large values of C, the number of empty bucket operations is much bigger for 2-level implementations. The 1-level implementation performs equally poorly (also, there was not enough memory for the largest problem instances). For the HARD-3 family and large values of C, the number of empty bucket operations is much bigger for 1-, 2-, and 3-level implementations.

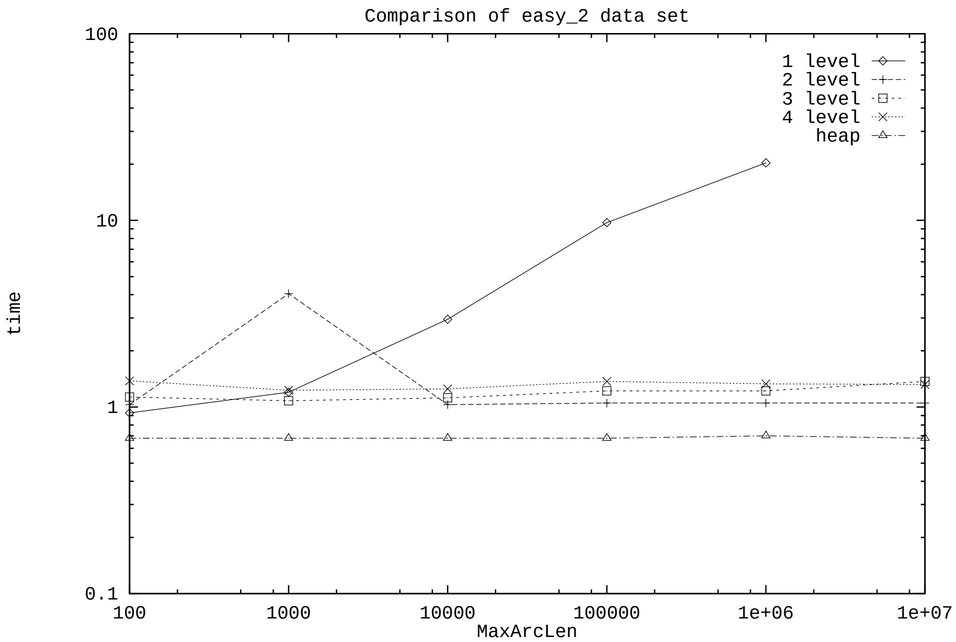
Table 23 gives data for the EASY-2 problem family designed to be easy for the 2-level implementation. This family is unusual because the number of empty bucket operations for multi-level implementations increases with the number of levels. The performance of the 2-level implementation is independent of C while the 3- and 4-level implementations run slower as C increases. Even for the largest value of C, however, the performance difference is not very large.

7.5 Heap Implementation Performance

The relative performance of the k-ary heap implementation depends mostly on the graph type.

On long grids, the heap contains very few nodes throughout the computation, heap operations are fast, and the heap implementation usually outperforms the multi-level bucket implementations. If C is small, however, overhead of the multi-level bucket implementations is small and they perform similarly to the heap implementation.

On the hard and easy problems the k-ary heap implementation contains at most two nodes



k	MaxArcLen	100	1000	10000	100000	1000000	10000000
1	**time**	**0.93** s	**1.20** s	**2.95** s	**9.73** s	**20.32** s	
	empty	802801	1904616	8192897	33292289	66911361	
	expanded	0	0	0	0	0	
	moved	131072	131072	131072	131072	131072	
2	**time**	**1.03** s	**4.05** s	**1.03** s	**1.05** s	**1.05** s	**1.05** s
	empty	2	6	14	158	574	1662
	expanded	131069	131069	131069	131069	131069	131069
	moved	131072	131072	131072	131072	131072	131072
3	**time**	**1.13** s	**1.08** s	**1.12** s	**1.22** s	**1.22** s	**1.37** s
	empty	49153	57349	172043	401400	430072	921584
	expanded	163836	147452	147452	147452	139260	139260
	moved	131072	131072	131072	131072	131072	131072
4	**time**	**1.38** s	**1.23** s	**1.25** s	**1.37** s	**1.33** s	**1.32** s
	empty	0	98301	229368	491509	17	25
	expanded	294905	204795	200699	198651	266233	264185
	moved	131072	131072	131072	131072	131072	131072
h	**time**	**0.68** s	**0.68** s	**0.68** s	**0.68** s	**0.70** s	**0.68** s
	moved	131072	131072	131072	131072	131072	131072

Table 23: Easy problems for the 2-level implementation. Here $n = 131072$.

on the heap at any point during the execution except at the beginning of the computation, when the heap contains at most four nodes. Thus the heap operation overhead is extremely small and the heap implementation is very fast on these problems.

On wide grid and random problem families, the heap contains relatively many nodes. On some of these problem families the multi-level bucket implementations have relatively large overhead for small n because C is large compared to n. Thus some of these implementations sometimes lose to the heap implementation for small n. For large n, however, the bucket implementations are always faster.

8 Conclusions

The previous study [5] concluded that the 2-level implementation often significantly outperforms, and never significantly underperforms, the 1-level implementation. The goal of our study was to evaluate the effect of the number of levels on performance. We studied the 1-, 2-, 3-, and 4-level implementations on a large collection of problem families and presented and explained the results for the most interesting subset of these.

Our study confirms that the 1-level implementation is not robust and should not be used unless the network depth D is not large compared to the number of nodes n.

The multi-level implementations perform consistently on most problem classes. The only exceptions in our study are the LONG-LARGE problems and the problems discussed in Section 7.4. Our results suggests that these implementations should exhibit consistent performance in most practical situations. In Section 7.4 we studied classes of problems designed to be hard or easy for certain implementations. The results show that it is possible to make the multi-level implementations perform poorly. No multi-level implementation dominates the others, but the 2-level implementation is less robust than the 3 and 4-level implementations. This is because the 2-level implementation loses by larger margins than it wins by. The 3-level implementation is less robust than the 4-level implementation. However, the 2-level implementation is competitive with the 3- and 4-level implementations unless C is very large. The 3- and 4-level implementations performed similarly except on the families specifically designed to differentiate them (see Section 7.4).

Note that multi-level implementations can handle huge arc lengths. For example, if $C = 2^{32}$, the number of buckets used by the 2-, 3-, and 4-level implementations is 2^{17}, $3 \cdot 2^{11}$, and 2^{10}, respectively. Even for 2-level implementations, the buckets require $128K$ words of memory, a small amount for modern computers. Note that C should be much smaller than 2^{32} for most applications, and 2^{32} is a natural bound for 32 bit computers. For $C = 2^{64}$, a natural bound for the 64 bit computers, the numbers change to 2^{33}, $3 \cdot 2^{22}$, and 2^{18}. The first number shows that the 2-level implementation requires too much memory. The 3-level implementation requires is $12M$ words for buckets in this case; too big for many of today's (but not tomorrow's) computers. The 4-level implementation requires only $256K$ words.

The 2-level bucket implementation has been suggested as a robust choice for shortest path problems with nonnegative arc length [5]. Our results confirm this conclusion and show that 3- and 4-level bucket implementations are even more robust choices.

Multi-level bucket data structure may be useful in other applications. One promissing application is the simulation event set problem, for which the calendar queue data structure, in some respects similar to the 2-level bucket data structure, appears to work very well in practice [3].

References

[1] R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan. Faster Algorithms for the Shortest Path Problem. *J. Assoc. Comput. Mach.*, 37(2):213–223, April 1990.

[2] R. E. Bellman. On a Routing Problem. *Quart. Appl. Math.*, 16:87–90, 1958.

[3] R. Brown. Calandar Queues: A Fast $O(1)$ Priority Queue Implementation for the Simulation Event Set Problem. *Comm. ACM*, 31:1220–1227, 1988.

[4] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest Paths Algorithms: Theory and Experimental Evaluation. Technical Report STAN-CS-93-1480, Department of Computer Science, Stanford University, 1993.

[5] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest Paths Algorithms: Theory and Experimental Evaluation. In *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms*, pages 516–525, 1994.

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

[7] R. B. Dial. Algorithm 360: Shortest Path Forest with Topological Ordering. *Comm. ACM*, 12:632–633, 1969.

[8] E. W. Dijkstra. A Note on Two Problems in Connection with Graphs. *Numer. Math.*, 1:269–271, 1959.

[9] E. A. Dinic. Economical algorithms for finding shortest paths in a network. In Yu.S. Popkov and B.L. Shmulyian, editors, *Transportation Modeling Systems*, pages 36–44. Institute for System Studies, Moscow, 1978. In Russian.

[10] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton Univ. Press, Princeton, NJ, 1962.

[11] M. L. Fredman and R. E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. Assoc. Comput. Mach.*, 34:596–615, 1987.

[12] M. L. Fredman and D. E. Willard. Trans-dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths. *J. Comp. and Syst. Sci.*, 48:533–551, 1994.

[13] G. Gallo and S. Pallottino. Shortest Paths Algorithms. *Annals of Oper. Res.*, 13:3–79, 1988.

[14] B. Ju. Levit and B. N. Livshits. *Neleneinye Setevye Transportnye Zadachi.* Transport, Moscow, 1972. In Russian.

[15] E. F. Moore. The Shortest Path Through a Maze. In *Proc. of the Int. Symp. on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[16] U. Pape. Implementation and Efficiency of Moore Algorithms for the Shortest Root Problem. *Math. Prog.*, 7:212–222, 1974.

[17] R. E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[18] R. A. Wagner. A shortest path algorithm for edge-sparse graphs. *J. Assoc. Comput. Mach.*, 23:50–57, 1976.