Priority Queues and Dijkstra's Algorithm *

Mo Chen † Rezaul Alam Chowdhury ‡ Vijaya Ramachandran §
David Lan Roche ¶ Lingling Tong ‖

UTCS Technical Report TR-07-54

October 12, 2007

Abstract

We study the impact of using different priority queues in the performance of Dijkstra's SSSP algorithm. We consider only general priority queues that can handle any type of keys (integer, floating point, etc.); the only exception is that we use as a benchmark the DIMACS Challenge SSSP code [1] which can handle only integer values for distances.

Our experiments were focussed on the following:

1. We study the performance of two variants of Dijkstra's algorithm: the well-known version that uses a priority queue that supports the *Decrease-Key* operation, and another that uses a basic priority queue that supports only *Insert* and *Delete-Min*. For the latter type of priority queue we include several for which high-performance code is available such as bottom-up binary heap, aligned 4-ary heap, and sequence heap [33].

2. We study the performance of Dijkstra's algorithm designed for flat memory relative to versions that try to be cache-efficient. For this, in main part, we study the difference in performance of Dijkstra's algorithm relative to the cache-efficiency of the priority queue used, both in-core and out-of-core. We also study the performance of an implementation of Dijkstra's algorithm that achieves a modest amount of additional cache-efficiency in undirected graphs through the use of two cache-efficient priority queues [25, 12]. This is theoretically the most cache-efficient implementation of Dijkstra's algorithm currently known.

Overall, our results show that using a standard priority queue without the decrease-key operation results in better performance than using one with the decrease-key operation in most cases; that cache-efficient priority queues improve the performance of Dijkstra's algorithm, both in-core and out-of-core on current processors; and that the dual priority queue version of Dijkstra's algorithm has a significant overhead in the constant factor, and hence is quite slow in in-core execution, though it performs by far the best on sparse graphs out-of-core.

*Supported in part by NSF Grant CCF-0514876 and NSF CISE Research Infrastructure Grant EIA-0303609.
†National Instruments Corporation, Austin, TX.
‡Department of Computer Sciences, University of Texas, Austin, TX. Email:shaikat@cs.utexas.edu.
§Department of Computer Sciences, University of Texas, Austin, TX. Email:vlr@cs.utexas.edu.
¶Google Inc., Mountain View, CA.
‖Microsoft Corporation, Redmond, WA.

1 Introduction

Dijkstra's single-source shortest path (SSSP) algorithm is the most widely used algorithm for computing shortest paths in a graph with non-negative edge-weights. A key ingredient in the efficient execution of Dijkstra's algorithm is the efficiency of the heap (i.e., priority queue) it uses. In this paper we present an experimental study on how the heap affects performance in Dijkstra's algorithm. We mainly consider heaps that are able to support arbitrary nonnegative key values so that the algorithm can be executed on graphs with general nonnegative edge-weights.

We consider the following two issues:

1. We study the relative performance of two variants of Dijkstra's algorithm: the traditional implementation that uses a heap with *Decrease-Key* operations (see Dijkstra-Dec in Section 1.1), and another simple variant that uses a basic heap with only *Insert* and *Delete-Min* operations (see Dijkstra-NoDec in Section 1.1).
2. We study how the performance of Dijkstra's algorithm varies with the cache-efficiency of the heap used, both in-core and out-of-core. We also study the performance of the theoretically best cache-efficient implementation of Dijkstra's algorithm for undirected graphs that uses two cache-efficient heaps (see Dijkstra-Ext in Section 1.1).

1.1 Dijkstra's SSSP Algorithm

We consider the following three implementations of Dijkstra's algorithm.

Dijkstra-Dec**.** This is the standard implementation of Dijkstra's algorithm that uses a heap that supports the *Decrease-Key* operation (see Function B.1 in the Appendix).

The heaps we use with Dijkstra-Dec are *standard binary heap* [40], which is perhaps the most widely-used heap, *pairing heap* [19], and cache-oblivious *buffer heap* [12], which is the only nontrivial cache-oblivious heap that supports the *Decrease-Key* operation. We implemented all three in-house.

The pairing heap was chosen as the best representative of the class of heaps that support *Decrease-Key* in amortized sub-logarithmic time. Though the Fibonacci heap [20] and others support it in $\mathcal{O}(1)$ amortized time and the pairing heap does not, an experimental study of the Prim-Dijkstra MST algorithm [29] found the pairing heap to be superior to other heaps considered (while standard binary heap performed better than the rest in most cases). In our preliminary experiments we used several other heaps including Fibonacci heap, and similar to [29] we found the pairing heap and the standard binary heap to be the fastest among the traditional (flat-memory) heaps.

Dijkstra-NoDec**.** This is an implementation that uses a heap with only *Insert* and *Delete-Min* operations (see Function B.2 in the Appendix). This implementation performs more heap operations and accesses to the graph data structure than Dijkstra-Dec. Further, theoretically this implementation is inferior to the asymptotic running time of Dijkstra-Dec when the latter is used with Fibonacci heap or other heap with amortized sub-logarithmic time support for *Decrease-Key*. However, since Dijkstra-NoDec can use a streamlined heap without the heavy machinery needed for supporting *Decrease-Key* (e.g., pointers in binary heap), the heap operations are likely to be more efficient than those in Dijkstra-NoDec. (see Appendix B.3.1 for more details).

The heaps we use with Dijkstra-NoDec are *bottom-up binary Heap*, *aligned 4-ary heap* and *sequence heap*, which are three highly-optimized heaps implemented by Peter Sanders [33], and two heaps coded in-house, the standard binary heap without support for *Decrease-Key* and auxiliary buffer heap, which is the buffer heap without support for *Decrease-Key*.

Dijkstra-Ext**.** This is an external-memory implementation for undirected graphs that uses two heaps: one with *Decrease-Key* operation, and the other one supporting only *Insert* and *Delete-Min* operations [25, 12] (see Function B.3 in the Appendix). This algorithm is asymptotically more cache-efficient (by a modest amount) than the two mentioned above, and this is achieved by reducing the number of I/Os for accessing the graph data structure at the cost of increasing the number of heap operations considerably (though only by a constant factor) relative to DIJKSTRA-NODEC. As a result, DIJKSTRA-EXT is expected to outperform the other implementations only in out-of-core computations, i.e., when the cost of accessing data in the external-memory becomes significant (see Section B.3.1 in the Appendix for more details). We note that there are undirected SSSP algorithms for graphs with bounded edge-weights [28, 3] that are more cache-efficient than DIJKSTRA-EXT, but these algorithms are not direct implementations of Dijkstra's algorithm as they use a hierarchical decomposition technique (similar to those in some flat-memory SSSP algorithms [37, 32]), and thus out of scope of this paper.

The theoretical I/O complexities of all heaps in our experiments are listed in Tables 1 and 2. In Table 3 we list the implemented I/O complexities for Dijkstra's algorithm.

For the purpose of comparison we include the following Dijkstra implementation which was used as the benchmark solver for the "*9th DIMACS Implementation Challenge – Shortest Paths*" [1].

DIJKSTRA-BUCKETS. An implementation of Dijkstra's algorithm with a heap based on a bucketing structure. This algorithm works only on graphs with integer edge-weights.

We performed our experiments on undirected $\mathcal{G}_{n,m}$ and directed power-law graphs, as well as on some real-world graphs from the benchmark instances of the 9th DIMACS Implementation Challenge [1].

Priority Queue	*Insert/Decrease-Key*	*Delete*	*Delete-Min*
Standard Binary Heap [40] (worst-case bounds)	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$
Two-Pass Pairing Heap [19, 30]	$\mathcal{O}\left(2^{2\sqrt{\log\log N}}\right)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$
Buffer Heap [12] (cache-oblivious)	$\mathcal{O}\left(\frac{1}{B}\log\frac{N}{M}\right)$	$\mathcal{O}\left(\frac{1}{B}\log\frac{N}{M}\right)$	$\mathcal{O}\left(\frac{1}{B}\log\frac{N}{M}\right)$

Table 1: Amortized I/O bounds for heaps with *Decrease-Keys* (N = # items in queue, B = block size, M = size of the cache/internal-memory).

Priority Queue	*Insert/Delete-Min*
Bottom-up Binary Heap [39] (worst-case bounds)	$\mathcal{O}(\log_2 N)$
Aligned 4-ary Heap [26] (worst-case bounds)	$\mathcal{O}(\log_4 N)$
Sequence Heap [33] (cache-aware)	$\mathcal{O}\left(\frac{1}{B}\log_k\frac{N}{l}+\frac{1}{k}+\frac{\log k}{l}\right)$
Auxiliary Buffer Heap (cache-oblivious, this paper)	$\mathcal{O}\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B}\right)$

Table 2: Amortized I/O bounds for heaps without *Decrease-Keys* (N = # items in queue, B = block size, M = size of the cache/internal-memory, $k=\Theta(M/B)$ and $l=\Theta(M)$).

1.2 Summary of Experimental Results

Briefly here are the conclusions of our experimental study:

- During in-core computations involving real-weighted sparse graphs such as undirected $\mathcal{G}_{n,m}$, road networks, and directed power-law, implementations based on DIJKSTRA-NODEC ran faster (often significantly) than DIJKSTRA-DEC implementations. However, this performance gap narrowed as the graphs became denser.
- Use of cache-efficient heaps generally improved performance of Dijkstra's algorithm both during in-core and out-of-core computations on low-diameter graphs such as $\mathcal{G}_{n,m}$ and power-law.
  – When the computation was in-core, an implementation using the cache-aware sequence heap ran the fastest followed by an implementation based on the cache-oblivious auxiliary buffer heap. Both ran faster than the DIMACS SSSP code. Among implementations of DIJKSTRA-DEC, the one using the cache-oblivious buffer heap was the fastest on Intel Xeon (which has strong prefetcher support), but not on AMD Opteron.

– When the computation was out-of-core, and the graph was not too dense, the external-memory implementation of Dijkstra's algorithm (DIJKSTRA-EXT) with cache-oblivious buffer heap and auxiliary buffer heap performed the fewest block transfers.

– For high-diameter graphs of geometric nature such as real-world road networks, Dijkstra's algorithm with traditional heaps performed almost as well as any cache-efficient heap when the computation was in-core.

Organization of the Paper. In Section 2 we give an overview of the heaps we used. We discuss our experimental setup in Section 3, and in Section 4 we discuss our experimental results.

2 Overview of Priority Queues

2.1 Flat-Memory Priority Queues

We implemented pairing heap and standard binary heap. Both were implemented so that they allocate and deallocate space from at most two arrays. Pointers were reduced to indices in the array. This allows us to limit the amount of internal-memory available to the heaps during out-of-core computations using STXXL (see Section 3).

Pairing Heap: We implemented four variants of the pairing heap: two-pass and multi-pass [19], and their auxiliary variants [36]. The two-pass variant has better theoretical bounds than the multi-pass variant and also ran faster in our experiments. The auxiliary variants ran only marginally (around 4%) faster than the corresponding primary variants, and so we will report results only for two-pass pairing heap. The two-pass pairing heap supports *Delete* and *Delete-Min* operations in $\mathcal{O}(\log N)$ and *Decrease-Key* operations in $\mathcal{O}\left(2^{\sqrt{\log\log N}}\right)$ amortized time and I/O bounds each.

Standard Binary Heap: We implemented the standard binary heap [40, 14]. However, we allocated nodes from a separate array and the heap stores only indices to those nodes. This ensures that whenever a new element is inserted all necessary information about it is stored at a location which remains static as long as the element remains in the heap. This allows *Decrease-Key* operations to be performed on a node by directly accessing it at any time. Standard binary heap supports *Insert*, *Delete*, *Delete-Min* and *Decrease-Key* operations in $\mathcal{O}(\log N)$ time and I/O bounds each.

Binary Heap w/o *Decrease-Keys*: This is a streamlined version of our implementation of the standard binary heap obtained by removing the overhead (e.g., pointers) of implementing *Decrease-Key* operations. This is a completely array-based heap.

Bottom-up Binary Heap: The bottom-up binary heap is a variant of binary heap which uses a bottom-up *Delete-Min* heuristic [39]. Compared to the traditional binary heap, this variant performs only half the number of comparisons on average per *Delete-Min*. We used Peter Sanders' highly optimized implementation of bottom-up binary heap [33] that has no redundant memory accesses or computations even in its assembler code, which supports only *Insert* and *Delete-Min*.

2.2 Cache-aware Priority Queues

Both of the following two heaps support only *Insert* and *Delete-Min* operations.

Aligned 4-ary Heap: In [26] this array-based heap was shown to outperform pointer based heaps. It aligns its data to cache blocks (so needs to know block size B) which reduces the number of cache-misses when accessing any data item. It supports *Insert* and *Delete-Min* operations in $\mathcal{O}(\log_4 N)$ time and block transfers each. In our experiments we used the optimized implementation of aligned 4-ary heap by Peter Sanders with the bottom-up *Delete-Min* heuristic [39].

Sequence Heap: The *sequence heap* is a cache-aware heap developed in [33]. It is based on k-way merging for some appropriate k. When the cache is fully associative, k is chosen to be $\Theta\left(\frac{M}{B}\right)$, and for some $l = \Theta(M)$ and $R = \left\lceil \log_k \frac{N}{l} \right\rceil$, it can perform N *Insert* and up to N *Delete-Min* operations in $\frac{2R}{B} + \mathcal{O}\left(\frac{1}{k} + \frac{\log k}{l}\right)$ amortized cache-misses and $\mathcal{O}(\log N + \log R + \log l + 1)$ amortized time each. For α-way set associative caches k is reduced by $\mathcal{O}\left(B^{\frac{1}{\alpha}}\right)$. We used a highly optimized version of the sequence heap implemented by Peter Sanders [33], and used $k = 128$ as suggested in [33].

2.3 Cache-oblivious Buffer Heap and Auxiliary Buffer Heap

Buffer Heap: The buffer heap is an efficient stack-based cache-oblivious heap [12, 13] (see [7] for a similar heap). It consists of $\log N$ levels with level $i \in [0, \log N)$ containing an element buffer B_i and an update buffer U_i, each of size at most 2^i. In our implementation we made design choices which do not always guarantee the best worst-case behavior but perform reasonably well in practice.

Size of Update Buffer. Our preliminary experiments suggested that bounding the sizes of update buffers slows down the operations by about 50%. Therefore, we chose to keep these sizes unrestricted as in [12]. This increases amortized I/O per operation to $\mathcal{O}\left(\frac{1}{B}\log_2 N\right)$ (up from $\mathcal{O}\left(\frac{1}{B}\log_2 \frac{N}{M}\right)$). This increase is reflected in the bound in Table 3.

Periodic Reconstruction. Periodic reconstruction ensures that the I/O complexity of buffer heap operations depends on the current size of the data structure and not on the total number of operations performed on it. However, in our preliminary experiments it slowed down the buffer heap operations by about 33%, and so we chose to avoid it.

Sorting U_0. We used randomized quicksort [23, 14] to sort U_0 when a *Delete-Min* is performed. It is simple, in-place and with high probability runs in $\mathcal{O}(N \log N)$ time and $\mathcal{O}\left(\frac{N}{B}\log_2 N\right)$ I/O operations. We chose not to use optimal cache-oblivious sorting [21] since this is complicated to implement, and randomized quicksort is sufficient to guarantee the I/O bounds of buffer heap.

Selection Algorithm for Redistribution. A selection algorithm is required to partition the elements collected for redistribution after a *Delete-Min* operation is performed. We used the classical randomized selection algorithm [22, 14] which is in-place and runs in $\Theta(N)$ expected time and performs $\Theta\left(\frac{N}{B}\right)$ expected I/O operations on a sequence of N elements.

Single Stack Implementation. We stored all buffers in a single stack with the update buffer on top of the element buffer at the same level. All temporary space was allocated on top of the stack. This implementation can benefit more from the limited number of prefetchers available for the caches than a multiple-array implementation, and also allowed us to limit the amount of internal-memory available to the data structure during our out-of-core experiments using STXXL (see Section 3).

Pipelining. Before we update any B_i (with updates in U_i) during a *Delete-Min* operation, we need to merge the segments of U_i. In our implementation, we applied the updates on B_i on the fly as we kept generating them during the merging process, which saved several extra scans over the updates.

Detailed description of most of our design choices for the buffer heap is available in the undergraduate honors thesis of Lingling Tong [38].

Auxiliary Buffer Heap: A simplified version of the buffer heap, the *auxiliary buffer heap*, that supports *Insert* and *Delete-Min* operations in optimal $\mathcal{O}\left(\frac{1}{B}\log_{\frac{M}{B}} \frac{N}{B}\right)$ I/Os, but no *Decrease-Keys* is described briefly in Appendix A. For our experiments, we implemented a streamlined version of this optimal version, derived directly from the buffer heap, whose amortized I/O per operation remains $\mathcal{O}\left(\frac{1}{B}\log_2 N\right)$ as in buffer heap, but the operations have much lower overhead. There are two main reasons why we chose to implement this streamlined non-optimal version. Firstly, we wanted to

Implementation	Base Routine	Priority Queue(s)	I/O Complexity
Bin-Dij	Dijkstra-Dec	Standard Binary Heap	$\mathcal{O}\left(m + (n+D)\cdot\log n\right)$
Pair-Dij	Dijkstra-Dec	Two-Pass Pairing Heap	$\mathcal{O}\left(m + n\cdot\log n + D\cdot 2^{2\sqrt{\log\log n}}\right)$
BH-Dij	Dijkstra-Dec	Buffer Heap	$\mathcal{O}\left(m + \frac{n+D}{B}\cdot\log(n+D)\right)$
SBin-Dij	Dijkstra-NoDec	Binary Heap w/o *Dec-key*	$\mathcal{O}\left(m + (n+D)\cdot\log(n+D)\right)$
FBin-Dij	Dijkstra-NoDec	Bottom-up Binary Heap	$\mathcal{O}\left(m + (n+D)\cdot\log(n+D)\right)$
Al4-Dij	Dijkstra-NoDec	Aligned 4-ary Heap	$\mathcal{O}\left(m + (n+D)\cdot\log(n+D)\right)$
Seq-Dij	Dijkstra-NoDec	Sequence Heap	$\mathcal{O}\left(m + \frac{n+D}{B}\cdot\left(\log_k\frac{n+D}{l} + \frac{1}{k} + \frac{\log k}{l}\right)\right)$
AH-Dij	Dijkstra-NoDec	Auxiliary Buffer Heap	$\mathcal{O}\left(m + \frac{n+D}{B}\cdot\log(n+D)\right)$
Dual-Dij	Dijkstra-Ext (undirected graphs only)	Buffer Heap & Auxiliary Buffer Heap	$\mathcal{O}\left(n + \frac{n+m}{B}\cdot\log m\right)$
DIMACS-Dij	Dijkstra-Bucket (integer edge-weights)	Buckets + Caliber Heuristic	$\mathcal{O}\left(m+n\right)$ (expected)

Table 3: Different implementations of Dijkstra's algorithm evaluated in this paper, where D ($\leq m$) is the number of *Decrease-Key*s performed by Dijkstra-Dec, B is the block size, $k = \Theta(M/B)$ and $l = \Theta(M)$.

use this implementation in experiments that attempt to measure the improvement (if any) in the performance of Dijkstra's SSSP algorithm if we remove the overhead of implementing *Decrease-Key* from a given priority queue and use it in Dijkstra-NoDec. Secondly, this version is easier to implement and is likely to have much lower overhead in practice than the optimal version. Other optimal cache-oblivious priority queues supporting only *Insert* and *Delete-Min* [4, 6] also appear to be more complicated to implement than the streamlined non-optimal auxiliary buffer heap.

Major features of the streamlined auxiliary buffer heap we implemented are as follows.

No Selection. There are $\log N$ levels, and contents of the buffers in each level are kept sorted by key value instead of element id and time stamp as in buffer heap. As a result during the redistribution step we do not need a selection algorithm.

Insertion and Delete-Min Buffers. Newly inserted elements are collected in a small *insertion buffer*. A small *delete-min buffer* holds the smallest few elements in the heap (excluding the insertion buffer) in sorted order. Whenever the insertion buffer overflows or a *Delete-Min* operation needs to be performed, appropriate elements from the buffer are moved to the heap. An underflowing delete-min buffer is filled up with the smallest elements from the heap.

Efficient Merge. We use the optimized 3-way merge technique described in [33] for merging an update buffer with the (at most two) segments of the update buffer in the next higher level.

Less Space. Uses less space than buffer heap since it does not store timestamps with each element.

Table 3 lists I/O complexities of different implementations we ran of Dijkstra's algorithm.

3 Experimental Set-up

We ran our experiments on the following two machines.

Model	Processors	Speed	L1 Cache	L2 Cache	RAM
Intel P4 Xeon	2	3.06 GHz	8 KB (4-way, $B = 64$ B)	512 KB (8-way, $B = 64$ B)	4 GB
AMD Opteron 250	2	2.4 GHz	64 KB (2-way, $B = 64$ B)	1 MB (8-way, $B = 64$ B)	4 GB

Table 4: Machines used for experiments.

Both machines ran Ubuntu Linux 5.10 "Breezy Badger". The Intel Xeon machine was connected to a 73.5 GB 10K RPM Fujitsu MAP3735NC hard disk with an 8 MB data buffer. The average seek time for reads and writes were 4.5 and 5.0 ms, respectively. The maximum data transfer rate (to/from media) was 106.9 MB/s. All experiments were run on a single processor.

We used the *Cachegrind* profiler [35] for simulating cache effects.

We implemented all algorithms in C++ using a uniform programming style, and compiled using the g++ 3.3.4 compiler with optimization level −O3.

For out-of-core experiments we used STXXL library version 0.9. The STXXL library [15, 16] is an implementation of the C++ standard template library STL for external memory computations, and is used primarily for experimentation with huge data sets. The STXXL library maintains its own fully associative cache in RAM with pages from the disk. We compiled STXXL with DIRECT-I/O turned on, which ensures that the OS does not cache the data read from or written to the hard disk. We also configured STXXL (more specifically the STXXL vectors) to use LRU paging.

We store the entire graph in a single vector so that the total amount of internal-memory available to the graph during out-of-core computations can be regulated by changing the STXXL parameters of the vector. The initial portion of the vector stores information on the vertices in increasing order of vertex id (each vertex is assumed to have a unique integer id from 1 to n) and the remaining portion stores the adjacency lists of the vertices in the same order. We store two pieces of information for each vertex: its distance value from the source vertex and a pointer to its adjacency list. For SSSP algorithms based on internal-memory heaps with *Decrease-Keys* (e.g., standard binary heap and pairing heap) we also store the pointer returned by the heap when the vertex is inserted into it for the first time. This pointer is used by all subsequent *Decrease-Key* operations performed on the vertex. For each edge in the adjacency list of a vertex we store the other endpoint of the edge and the edge-weight. Each undirected edge (u, v) is stored twice: once in the adjacency list of u and again in the adjacency list of v. For each graph we use a one-time preprocessing step that puts the graph in the format described above.

Graph Classes. We ran our experiments on three graph classes obtained through the *9th DIMACS Implementation Challenge* [1]: Two synthetic classes, *undirected* $\mathcal{G}_{n,m}$ *(PR)* [31] and *directed power-law graphs (GT)* [5], and the real-world class of *undirected U.S. road networks* [34]. In Appendices D and E we present experimental results on the following additional (undirected) graph classes: *regular* [31], grid [31], geometric [31] and layered [2]. A more detailed description of these graphs is included in Appendix C.

4 Experimental Results

We present a detailed description of our experimental results on Intel Xeon, and summarize our results on AMD Opteron in Section 4.6. Unless specified otherwise, all experimental results presented in this section are averages of three independent runs from three random source vertices on a randomly chosen graph from the graph class under consideration, and they do not include the cost of the one-time preprocessing step that puts the graph in the format mentioned in Section 3.

4.1 In-Core Results for $\mathcal{G}_{n,m}$

We consider graphs in which we keep the average degree of vertices fixed and vary the number of edges (by varying the number of vertices), and also graphs in which we keep the number of edges fixed and vary the average degree of vertices (again by varying the number of vertices).

4.1.1 $\mathcal{G}_{n,m}$ with Fixed Average Degree

Figure 1 shows the in-core performance of all implementations (except *Dual-Dij* which was much slower than all others in-core) on $\mathcal{G}_{n,m}$ with a fixed average degree 8, i.e., $\frac{m}{n} = 8$.

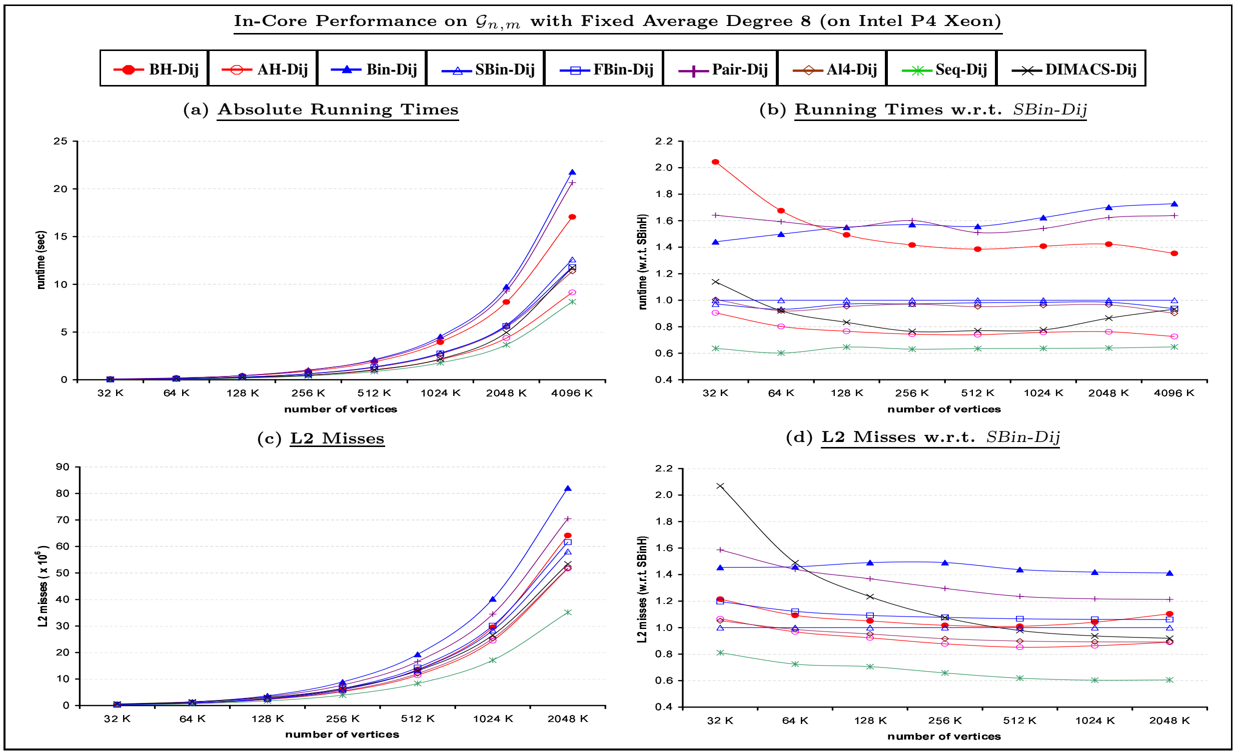

Figure 1: In-core performance of algorithms on $\mathcal{G}_{n,m}$ with fixed average degree 8 (on Intel P4 Xeon).

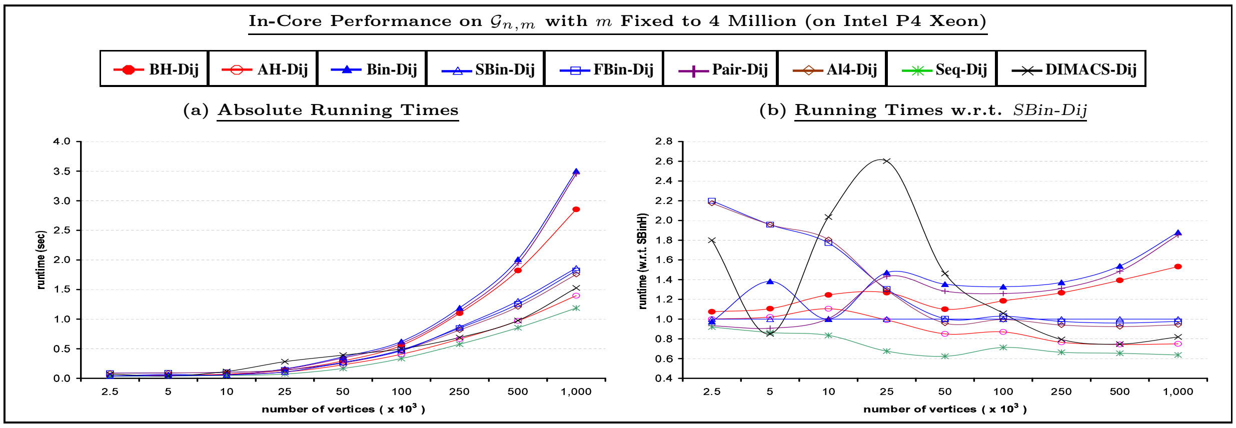

Figure 2: In-core performance of algorithms on $\mathcal{G}_{n,m}$ with m fixed to 4 million (on Intel P4 Xeon).

Running Times. Figures 1(a) and 1(b) show that as n was varied from 2^{15} to 2^{22}, all DIJKSTRA-NODEC implementations (i.e., *AH-Dij*, *FBin-Dij*, *SBin-Dij*, *Al4-Dij* and *Seq-Dij*) ran at least 1.4 times faster than any DIJKSTRA-DEC implementation (i.e., *BH-Dij*, *Bin-Dij* and *Pair-Dij*). We will investigate this observation in more detail in Section 4.2.

Among all implementations, *Seq-Dij* consistently ran the fastest, while *AH-Dij* was consistently faster than the remaining implementations. *Seq-Dij* ran around 25% faster than *AH-Dij*. *FBin-Dij*, *SBin-Dij*, *Al4-Dij* and *DIMACS-Dij* ran at almost the same speed, and were consistently 25% slower than *AH-Dij*. *BH-Dij* was the fastest among DIJKSTRA-DEC for $n \geq 128$, and ran up to 25% faster than the remaining two. The slowest of all implementations was *Bin-Dij*.

Cache Performance. Figures 1(c) and 1(d) plot the L2 cache misses incurred by different implementations (except *Dual-Dij*). As expected, cache-aware *Seq-Dij* incurred the fewest cache-misses followed by cache-oblivious *AH-Dij*. The cache-oblivious *BH-Dij* incurred more cache-misses than *AH-Dij*, but fewer than any Dijkstra-Dec implementation.

As n grows larger the cache performance of *BH-Dij* degrades with respect to *Bin-Dij* and *Pair-Dij* which can be explained as follows. All Dijkstra-Dec implementations perform exactly the same number of *Decrease-Key* operations (our experimental results suggest that this number is $\approx 0.8n$ for $\mathcal{G}_{n,m}$ with average degree 8). The flat-memory priority queues we used support *Decrease-Key* operations more efficiently on average than in the worst case, which is not the case with buffer heap. Hence the cache-performance of *BH-Dij* degrades as a whole with respect to that of *Bin-Dij* and *Pair-Dij* as n increases. Surprisingly, however, Figure 1(b) shows that the running time of *BH-Dij* improves with respect to most other implementations as n increases. We believe this happens because of the prefetchers in Intel Xeon. As the operations of buffer heap involves only sequential scans it benefits more from the prefetchers than the internal-memory heaps. The cachegrind profiler does not take hardware prefetching into account and as a result, Figures 1(c) and 1(d) failed to reveal their impact. The same phenomenon is seen to a lesser extent for *AH-Dij*.

4.1.2 $\mathcal{G}_{n,m}$ with Fixed Number of Edges

Figures 2(a) and 2(b) plot running times as the number of vertices is increased from 2500 to 1 million while keeping m fixed to 4 million (i.e. average degree is decreased from 1600 down to 4). As before, cache-aware *Seq-Dij* consistently ran the fastest followed by the cache-oblivious *AH-Dij*. When the graph was sparse all implementations based on Dijkstra-NoDec ran significantly faster than any implementation based on Dijkstra-Dec, but this performance gap narrowed as the graph became denser. As in Section 4.1.1, for Dijkstra-Dec the cache-oblivious *BH-Dij* ran faster than flat-memory implementations (i.e., *Bin-Dij* & *Pair-Dij*). As the average degree of the graph decreased down to 160, performance of the *DIMACS-Dij* solver degraded significantly, but after that its performance improved dramatically.

Remarks on Dijkstra-Ext. *Dual-Dij* ran considerably slower than other implementations as it performs significantly more heap operations compared to any of them (see Table 7 in the Appendix). For example, on $\mathcal{G}_{n,m}$ with average degree 8, *Dual-Dij* consistently performed at least 6 times more heap operations and ran at least 6 times slower than any other implementations.

4.2 Dijkstra-Dec vs. Dijkstra-NoDec

In Figure 3 we take a closer look at the performance gap between Dijkstra-NoDec and Dijkstra-Dec. In Figures 3(a) and 3(b) we plot the running times of *Bin-Dij*, *SBin-Dij*, *BH-Dij* and *AH-Dij* on $\mathcal{G}_{n,1000000}$ as n (and thus edge-density) varies. Both *Bin-Dij* and *BH-Dij* are based on Dijkstra-Dec, and they use standard binary heap and cache-oblivious buffer heap, respectively, as heaps. In contrast, *SBin-Dij* and *AH-Dij* are based on Dijkstra-NoDec, and they use streamlined versions of the standard binary heap and the buffer heap, respectively. These two streamlined heaps are obtained by removing the heavy machinery for supporting *Decrease-Key* operations from the original heaps (see Section 2 for details), and thus support only *Insert* and *Delete-Min* operations. Since these light-weight heap implementations have much lower overhead compared to their heavy-weight counterparts, they contribute to more efficient shortest path computation in spite of the fact that Dijkstra-NoDec performs more heap operations than Dijkstra-Dec. Figures 3(a) and 3(b) show that for sparse graphs *SBin-Dij* and *AH-Dij* run significantly faster than *Bin-Dij* and *BH-Dij*, respectively, though the performance gap diminishes as the graphs become denser.

In order to understand the relative performance of Dijkstra-Dec and Dijkstra-NoDec, let us compute their theoretical complexity in terms of the heap operations performed. We observe that Dijkstra-Dec performs n *Insert* and *Delete-Min* operations each, and the corresponding figure for Dijkstra-NoDec is $n + D$, where D is the number of *Decrease-Key* operations executed

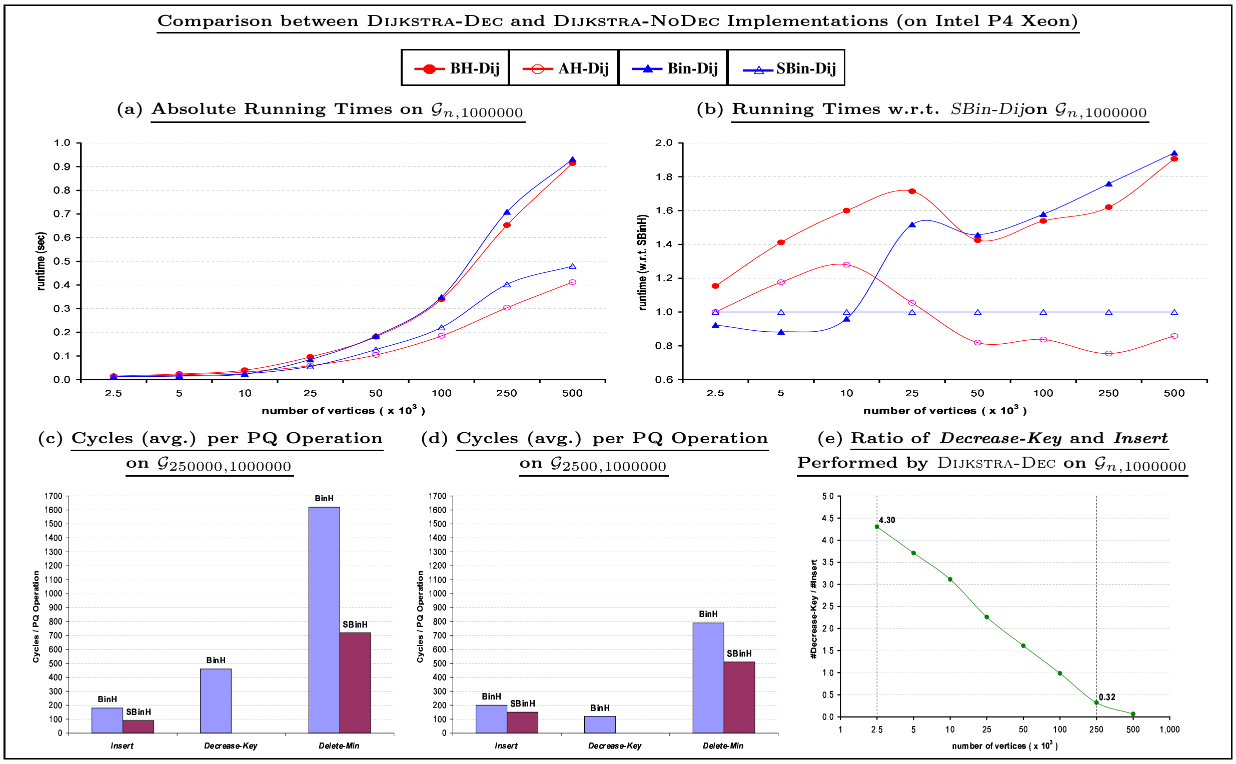

Figure 3: Plots (a) and (b) compare runtimes of *Bin-Dij* & *BH-Dij*, and *SBin-Dij* & *AH-Dij*, respectively, on $\mathcal{G}_{n,1000000}$ as n varies. Plots (c) and (d) show the avg. number of cycles per heap operation performed by *Bin-Dij* and *SBin-Dij* on $\mathcal{G}_{250000,1000000}$ and $\mathcal{G}_{2500,1000000}$, respectively. Plot (e) shows the ratio of the number of *Decrease-Key* and *Insert* operations performed by Dijkstra-Dec on $\mathcal{G}_{n,1000000}$ as n varies.

by Dijkstra-Dec. Let the heap used by Dijkstra-Dec support each *Insert*, *Delete-Min* and *Decrease-Key* operation in c_{ins}, c_{del} and c_{dec} clock cycles (avg), respectively, and let the heap for Dijkstra-NoDec support each *Insert* and *Delete-Min* operation in c'_{ins} and c'_{del} clock cycles (avg), respectively. Hence, if Dijkstra-Dec requires Δ extra clock cycles compared to Dijkstra-NoDec to perform all heap operations, then $\Delta = n(c_{ins} - c'_{ins}) + n(c_{del} - c'_{del}) + D(c_{dec} - c'_{ins} - c'_{del})$.

In Figures 3(c) and 3(d), we compare empirically obtained (using Callgrind [35]) values of c_{ins}, c_{del} and c_{dec} with those of c'_{ins} and c'_{del} for $\mathcal{G}_{250000,1000000}$ and $\mathcal{G}_{2500,1000000}$, respectively. In Figure 3(e), we plot the empirical ratio of the number of *Decrease-Key* operations ($= D$) to the number of *Insert* operations ($\approx n$) performed by Dijkstra-Dec on $\mathcal{G}_{n,1000000}$ for different values of n. Now for $\mathcal{G}_{250000,1000000}$, from Figure 3$(e)$ we obtain $D \approx 0.32n$, and using this value we obtain from Figure 3(c), $\Delta = 990n - 350D \approx 220 \times 10^6$. Therefore, *SBin-Dij*, indeed, spends fewer clock cycles on heap operations than *Bin-Dij* in this case, and Figures 3(a) and 3(b) show that this translates into a better overall running time for *SBin-Dij*. Similarly, for $\mathcal{G}_{2500,1000000}$, we obtain $\Delta = 330n - 540D$ from Figure 3(d), and $D \approx 4.3n$ from Figure 3(e). Thus $\Delta \approx -5 \times 10^6$, and hence *Bin-Dij* will perform better than *SBin-Dij* in this case. Figures 3(a) and 3(b) confirm our conclusion.

More experimental results on the relative performance of *Bin-Dij* and *SBin-Dij* can be found in the undergraduate honors thesis of Mo Chen [10].

The relative performance of *BH-Dij* and *AH-Dij* follows a trend similar to that of *Bin-Dij* and *SBin-Dij* (see Figures 3(a) and 3(b)), which can be explained similarly.

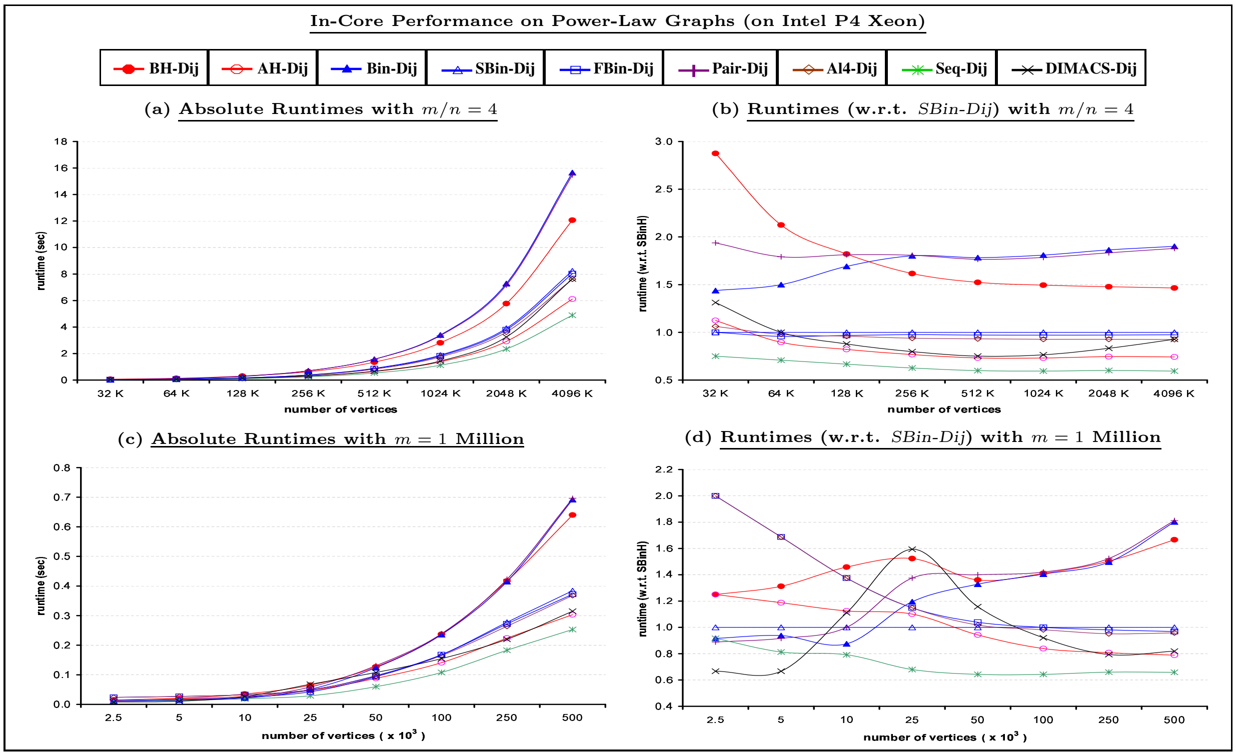

Figure 4: In-core performance of algorithms on power-law graphs.

4.3 In-Core Results for Directed Power-Law Graphs

In Figures 4(a) and 4(b), we plot the running times of different implementations on directed power-law graphs with fixed average degree 4 as the number of vertices is varied. Figures 4(c) and 4(d) plot running times as the average degree of the graph is varied by keeping the number of edges fixed to 1 million. The trends in both cases are similar to those observed for $\mathcal{G}_{n,m}$ in Section 4.1.

4.4 Out-of-Core Results for Undirected $\mathcal{G}_{n,m}$

We report our experimental results for *fully external* computation where neither the vertex set nor the edge set completely fits in internal memory, and the heap also is too large to fit in internal memory. We fixed the amount of internal memory available to store the graph and the heap separately by fixing the STXXL parameters of the corresponding vectors. We fixed the block size B to 4 KB and internal-memory size M to 4 MB. We have not included *Seq-Dij* in these experiments because it was quite difficult to reimplement it in order to make it compatible with STXXL.

Figures 5(a) and 5(b) plot the number of blocks transferred by different implementations for $\mathcal{G}_{n,m}$ as the average degree of the graph is varied while keeping the number of edges fixed to 2 million. They show that both *BH-Dij* and *AH-Dij* perform over 2 times more block transfers than *Dual-Dij*, while *FBin-Dij* and *Al4-Dij* perform over 2.5 times more. We believe that this is due to the $\mathcal{O}(m)$ I/O overhead of *BH-Dij*, *AH-Dij*, *FBin-Dij* and *Al4-Dij*, for accessing the graph data structure compared to only $\mathcal{O}\left(n + \frac{m}{B}\right)$ I/O operations performed by *Dual-Dij* for the same (see Table 6 in the Appendix).

More information on the out-of-core performance of these implementations can be found in the undergraduate honors thesis of David Lan Roche [27].

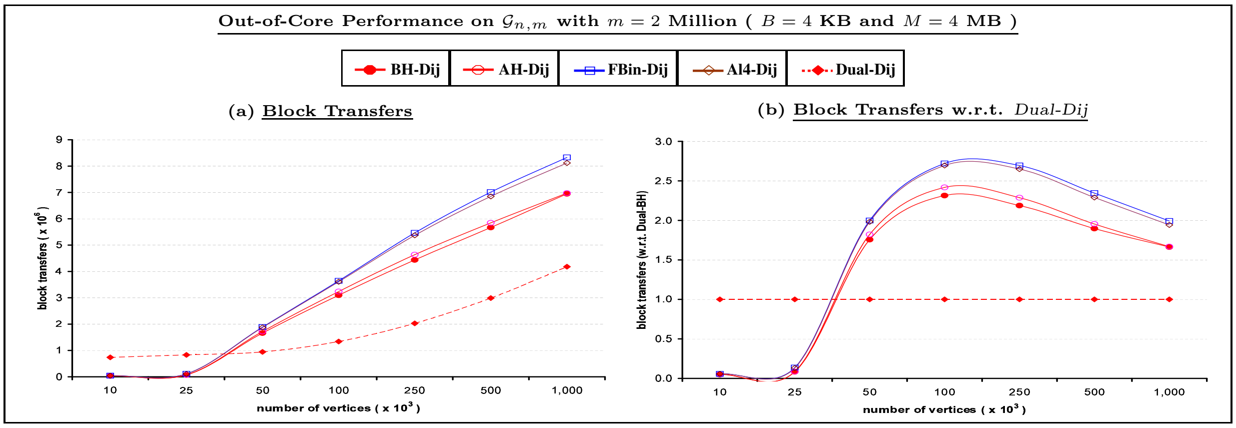

Figure 5: Out-of-core performance on $\mathcal{G}_{n,m}$ with m fixed to 2 million.

4.5 Performance on Real-World Graphs

In Table 5 we tabulate the results for road networks of different regions of the United States. These regional networks were downloaded from the DIMACS Challenge 9 website [1]. We chose the physical distance between the endpoints as the weight of each edge.

The *Seq-Dij* implementation was the fastest followed by *FBin-Dij*, *SBin-Dij*, *AH-Dij* and *Al4-Dij*, all four of which ran at around the same speed. As usual, all DIJKSTRA-DEC implementations (i.e., *Bin-Dij*, *Pair-Dij* and *BH-Dij*) were slower than all DIJKSTRA-NODEC implementations with *BH-Dij* being the slowest. Though for smaller networks the *DIMACS-Dij* benchmark code was competitive with DIJKSTRA-NODEC implementations, it slowed down as the networks became larger.

The road networks are almost planar and their edge distributions ensure that nodes are connected to only nearby nodes on the plane. Since Dijkstra's algorithm only stores fringe nodes in the heap, the nature of these graphs keep the size of the heap very small. Empirical evidence suggests that in our experiments the heaps were so small in size that they often fit in L2 cache. In the absence of any significant savings in L2 misses, cache-efficient heaps did not perform significantly better than flat-memory heaps, and other overheads in the implementations of buffer heap caused *BH-Dij* to run slower than all flat-memory heap based implementations.

4.6 Experimental Results on AMD Opteron

We ran the in-core experiments on $\mathcal{G}_{n,m}$ and power-law graphs described in Sections 4.1– 4.3 on AMD Opteron. We have included the resulting plots in Figures 12 and 13 in Appendix E. We summarize the results below.

As on Intel Xeon, DIJKSTRA-NODEC-based implementations easily outperformed implementations based on DIJKSTRA-DEC when the graph was not too dense. Among DIJKSTRA-NODEC-based implementations, the ones using cache-efficient priority queues performed better than the ones using flat-memory priority queues, and *Seq-Dij* ran the fastest followed by *AH-Dij*. However, among DIJKSTRA-DEC implementations, the flat-memory versions, i.e., *Bin-Dij* and *Pair-Dij*, often outperformed the cache-efficient *BH-Dij*. We believe that on Intel Xeon, *BH-Dij* was aided by the prefetchers (since *BH-Dij* involves a large number of sequential scans) to overcome the overheads in its implementation, and thus was able to beat *Bin-Dij* and *Pair-Dij*. But on AMD Opteron the prefetchers are perhaps not as effective as those on Xeon, and as a result, *BH-Dij* ran slower.

	Running Time in Milliseconds (on Intel P4 Xeon)								
Region	*Bin-Dij*	*SBin-Dij*	*FBin-Dij*	*Pair-Dij*	*Al4-Dij*	*Seq-Dij*	*BH-Dij*	*AH-Dij*	*DIMACS-Dij*
New York City	104	90	82	156	94	76	336	91	82
San Fran. Bay Area	120	101	96	179	107	87	381	106	94
Colorado	171	139	132	242	149	121	532	149	137
Florida	441	358	336	612	379	310	1,328	374	353
Northwest USA	540	427	420	761	456	381	1,578	445	455
Northeast USA	738	603	579	1,037	633	529	2,084	614	645
California & Nevada	876	715	688	1,266	783	645	2,531	746	791
Great Lakes	1,329	1,106	1,063	1,958	1,172	986	3,779	1,138	1,213
Eastern USA	1,966	1,606	1,563	2,731	1,686	1,453	5,150	1,642	1,842
Western USA	3,698	3,114	3,019	5,096	3,210	2,792	9,665	3,139	3,654
Central USA	13,965	11,803	11,854	17,236	12,341	11,326	27,450	12,109	15,225

Table 5: Running time on US road networks (TIGER/Line). Rows are in increasing order of region sizes ranging from $n = 0.26 \times 10^6$ and $m = 0.73 \times 10^6$ for New York City to $n = 0.26 \times 10^6$ and $m = 0.73 \times 10^6$ for Central USA.

References

[1] 9th DIMACS Implementation Challenge — Shortest Paths, 2006. http://www.dis.uniroma1.it/~challenge9/.

[2] D. Ajwani, R. Dementiev, and U. Meyer. Graph generators for external memory bfs algorithms. url: http://www.mpi-sb.mpg.de/~ajwani/graph_gen/.

[3] L. Allulli, P. Lichodzijewski, and N. Zeh. A faster cache-oblivious shortest-path algorithms for undirected graphs with bounded edge lengths. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 910–919, New Orleans, Louisiana, 2007.

[4] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the 24th ACM Symposium on Theory of Computing*.

[5] D. Bader and K. Madduri. GTgraph: A suite of synthetic graph generators. url: http://www-static.cc.gatech.edu/~kamesh/GTgraph/.

[6] G. S. Brodal and R. Fagerberg. Funnel heap – a cache oblivious priority queue. In *Proceedings of the 13th Annual International Symposium on Algorithms and Computation*, LNCS 2518, Vancouver, BC, Canada. Springer-Verlag.

[7] G.S. Brodal, R. Fagerberg, U. Meyer, and N. Zeh. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In *Proceedings of the 3rd Scandinavian Workshop on Algorithm Theory*, pages 480–492, Humlebæk, Denmark, July 2004.

[8] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proc. 11th ACM-SIAM SODA*, pages 859–860, 2000.

[9] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proc. 4th SIAM Intl Conf on Data Mining*, Orlando, Florida, 2004.

[10] M. Chen. Measuring and improving the performance of cache-efficient priority queues in dijkstra's algorithm, 2007. Undergraduate Honors Thesis, CS-HR-07-36, Univ of Texas, Austin, Dept of Comp Sci.

[11] R. A. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *Proc. 17th ACM-SIAM SODA*.

[12] R. A. Chowdhury and V. Ramachandran. Cache-oblivious shortest paths in graphs using buffer heap. In *Proc. 16th ACM SPAA*.

[13] R. A. Chowdhury and V. Ramachandran. External-memory exact and approximate all-pairs shortest paths in undirected graphs. In *Proc. 16th ACM-SIAM SODA*.

[14] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

[15] R. Dementiev. STXXL homepage, documentation and tutorial. url: http://stxxl.sourceforge.net/.

[16] R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard template library for XXL data sets. In *Proc. 13th ESA*, LNCS 1004, pages 640–651. Springer-Verlag, 2005.

[17] E.W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[18] P. Erdös and A. Rényi. On the evolution of random graphs. *Mat. Kuttató. Int. Közl.*, 5:17–60, 1960.

[19] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1:111–129, 1986.

[20] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their use in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[21] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. 40th FOCS*, pages 285–297, 1999.

[22] C. A. R. Hoare. Algorithm 63 (PARTITION) and algorithm 65 (FIND). *Comm. ACM*, 4(7):321–322, 1961.

[23] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[24] I. Katriel and U. Meyer. Elementary graph algorithms in external memory. In U. Meyer, P. Sanders, and J.F. Sibeyn, editors, *Alg. for Mem. Hierarchies*, LNCS 2625. Springer-Verlag.

[25] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. 8th IEEE SPDP*, pages 169–177, 1996.

[26] A. LaMarca and Ladner R. The influence of caches on the performance of heaps. *Journal of Experimental Algorithmics*, 1:4, 1996.

[27] D. Lan Roche. Experimental study of high performance priority queues, 2007. Undergraduate Honors Thesis, CS-TR-07-34, The University of Texas at Austin, Department of Computer Sciences.

[28] U. Meyer and N. Zeh. I/O-efficient undirected shortest paths. In *Proceedings of the 11th European Symposium on Algorithms*, LNCS 2832, pages 434–445. Springer-Verlag, 2003.

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

[30] S. Pettie. Towards a final analysis for pairing heaps. In *Proc. 46th FOCS*, pages 174–183, 2005.

[31] S. Pettie and V. Ramachandran. Command line tools generating various families of random graphs. url: http://www.dis.uniroma1.it/~challenge9/code/Randgraph.tar.gz.

[32] S. Pettie and V. Ramachandran. A shortest path algorithm for real-weighted undirected graphs. *SIAM Jour. on Comput*, 34:1398–1431, 2005.

[33] P. Sanders. Fast priority queues for cached memory. *Jour. Exp. Algorithmics*, 5:1–25, 2000.

[34] P. Sanders and D. Schultes. United states road networks (tiger/line). Data Source: U.S. Census Bureau, Washington, DC, url: http://www.dis.uniroma1.it/~challenge9/data/tiger/.

[35] J. Seward and N. Nethercote. Valgrind (debugging and profiling tool for x86-Linux programs). url: http://valgrind.kde.org/index.html.

[36] J. T. Stasko and J. S. Vitter. Pairing heaps: experiments and analysis. *Comm. ACM*, 30:234–249, 1987.

[37] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46:362–394, 1999.

[38] L. Tong. Implementation and experimental evaluation of the cache-oblivious buffer heap, 2006. Undergraduate Honors Thesis, CS-TR-06-21, Univ of Texas, Austin, Dept of Comp Sci.

[39] I. Wegner. BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT, beating, on an average, QUICKSORT (if n is not very small). *Theoretical Computer Science*, 118(1):81–98, 1993.

[40] J. W. J. Williams. Algorithm 232 (HEAPSORT). *Comm. ACM*, 7:347–348, 1964.

Appendix

A The Optimal Auxiliary Buffer Heap

This is a simplified version of the buffer heap which does not support *Decrease-Keys*, but supports *Insert* and *Delete-Min* operations in optimal amortized $\mathcal{O}\left(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{M}\right)$ I/Os each.

In contrast to the original buffer heap [11] which has $\log N$ levels, the optimal version has $\log\log N$ levels. For $i \in [0, \log\log N)$, level i has an element buffer B_i containing at most $2^{2^{i+1}}$ elements, and an update buffer U_i containing at most 2^{2^i} updates (i.e., *Insert* operations). Element buffer B_i is divided into at most 2^{2^i} segments with each segment containing between $\frac{1}{2}\cdot 2^{2^i}$ and $2\cdot 2^{2^i}$ elements. The segments of B_i are kept linked together in a chain such that each element in segment $j+1$ is at least as large as any element in segment j. We also keep track of the largest element value in each segment which we call the splitter value. However, elements in a given segment are not sorted. The updates in U_i are not kept sorted either. When applying the updates in U_i on B_i, we first sort U_i by value, and then distribute the elements generated by U_i to appropriate segments of B_i by scanning U_i and the sorted array of splitters of B_i simultaneously. Whenever a segment in B_i overflows, we split it into two segments of roughly equal size. If, after applying U_i, B_i contains more than $2^{2^{i+1}}$ elements, we move enough segments from the end of the segment-chain of B_i (i.e., segments containing large values) to U_{i+1} so that the remaining segments contain no more than $2^{2^{i+1}}$ elements. As in the analysis of buffer heap (see, e.g., [11]), we observe that items in update buffers typically move downward, and items in element buffers typically move upward, and in the amortized sense, each such item undergoes a constant number of sorting steps at each level. Therefore, each item causes a total of $\mathcal{O}\left(\sum_{i=0}^{\log\log N-1}\left(\frac{1}{B}\log_{M/B}\frac{2^{2^i}}{M}\right)\right) = \mathcal{O}\left(\frac{1}{B}\log_{M/B}\frac{N}{M}\right)$ amortized I/O operations, which is optimal.

B Implementations of Dijkstra's SSSP Algorithm

In our study, we considered the following three implementations of Dijkstra's SSSP algorithm.

B.1 Dijkstra's SSSP Algorithm with *Decrease-Keys*

Dijkstra's SSSP algorithm for directed graphs [17] works by maintaining an upper bound on the shortest distance (a *tentative distance*) to every vertex from the source vertex s and visiting the vertices one by one in non-decreasing order of tentative distances (see function Dijkstra-Dec in Figure 6). The next vertex to be visited is the one with the smallest tentative distance extracted from the set of unvisited vertices (with finite tentative distance) kept in a heap Q. After a vertex has been extracted from Q it is considered *settled*, and each of its unvisited neighbors is either inserted into Q with a finite tentative distance or has its tentative distance updated if it already resides in Q. Dijkstra's algorithm performs n *Insert* and *Delete-Min* operations each and $\mathcal{O}(m-n)$ *Decrease-Key* operations on Q.

Using a binary heap that supports *Insert*, *Decrease-Key* and *Delete-Min* operations in $\mathcal{O}(\log n)$ time and I/O operations each Dijkstra's algorithm can be implemented to run in $\mathcal{O}((n+m)\cdot\log n)$ time and perform $\mathcal{O}((n+m)\cdot\log n)$ I/Os. If a Fibonacci heap is used that supports *Insert*/*Decrease-Key* operations in constant amortized time and I/Os, both the time and the I/O complexity of the algorithm reduces to $\mathcal{O}(m+n\cdot\log n)$. However, if a buffer heap is used as the heap, Dijkstra's algorithm runs in $\mathcal{O}((n+m)\cdot\log n)$ time, but performs $\mathcal{O}\left(m+\frac{n+m}{B}\cdot\log n\right)$ I/O operations which

Function **B.1.** Dijkstra-Dec(G, w, s, d)	Function **B.2.** Dijkstra-NoDec(G, w, s, d)
{*Dijkstra's SSSP algorithm [17] with a heap that supports Decrease-Keys*}	{*Dijkstra's SSSP algorithm [17] with a heap that does not support Decrease-Keys*}
1. perform the following initializations:	**1.** perform the following initializations:
(i) $Q \leftarrow \emptyset$	(i) $Q \leftarrow \emptyset$
(ii) ***for*** each $v \in V[G]$ ***do*** $d[v] \leftarrow +\infty$	(ii) ***for*** each $v \in V[G]$ ***do*** $d[v] \leftarrow +\infty$
(iii) $\text{Insert}_{(Q)}(\ s,\ 0\ )$	(iii) $\text{Insert}_{(Q)}(\ s,\ 0\ )$
2. ***while*** $Q \neq \emptyset$ ***do***	**2.** ***while*** $Q \neq \emptyset$ ***do***
(i) $(u, k) \leftarrow \text{Delete-Min}_{(Q)}(\ )$, $d[u] \leftarrow k$	(i) $(u, k) \leftarrow \text{Delete-Min}_{(Q)}(\ )$
(ii) ***for*** each $(u, v) \in E[G]$ ***do***	(ii) ***if*** $k < d[u]$ ***then***
if $d[u] + w(u, v) < d[v]$ ***then***	$d[u] \leftarrow k$
if $d[v] = +\infty$ ***then***	***for*** each $(u, v) \in E[G]$ ***do***
$\text{Insert}_{(Q)}(\ v,\ d[u] + w(u, v)\ )$	***if*** $d[u] + w(u, v) < d[v]$ ***then***
else $\text{Decrease-Key}_{(Q)}(\ v,\ d[u] + w(u, v)\ )$	$\text{Insert}_{(Q)}(\ v,\ d[u] + w(u, v)\ )$
$d[v] \leftarrow d[u] + w(u, v)$	$d[v] \leftarrow d[u] + w(u, v)$
Dijkstra-Dec Ends	Dijkstra-NoDec Ends

Figure 6: Given a directed graph G with vertex set $V[G]$ (each vertex is identified with a unique integer in $[1, |V[G]|]$), edge set $E[G]$, a weight function $w : E[G] \rightarrow \Re$ and a source vertex $s \in V[G]$, both functions compute the shortest distance from s to each vertex $v \in V[G]$ and stores it in $d[v]$.

is a factor of $\Theta(\log n)$ improvement over the I/O complexity of Dijkstra's algorithm with Fibonacci heap provided the graph is very sparse (i.e., $m = \mathcal{O}(n)$), and $B \gg \log n$ which typically holds for memory levels deeper in the hierarchy such as the disk.

B.2 Dijkstra's Algorithm without *Decrease-keys*

It is straight-forward to implement Dijkstra's algorithm using a heap that supports only *Insert* and *Delete-Min* operations (see function Dijkstra-NoDec in Figure 6). The implementation performs $\mathcal{O}(m)$ *Insert* and *Delete-Min* operations and runs in $\mathcal{O}(m \cdot \log n)$ time and performs $\mathcal{O}(m \cdot \log n)$ I/O operations using an internal-memory heap. However, if a buffer heap or an auxiliary buffer heap is used, the algorithm continues to run in $\mathcal{O}(m \cdot \log n)$ time but performs $\mathcal{O}\left(m + \frac{m}{B} \cdot \log m\right)$ I/O operations which a $\Theta(\log n)$ factor improvement over the I/O bound using an internal-memory heap if the graph is very sparse (i.e., $m = \mathcal{O}(n)$) and $B \gg \log n$.

B.3 External-Memory Implementation of Dijkstra's Algorithm for Undirected Graphs

The traditional implementations of Dijkstra's algorithm (see Figure 6) do not perform *Decrease-Key* operations on vertices that are already settled, but in the process they incur $\Theta(1)$ cache misses per edge. If one allows such *Decrease-Key* operations then either one must be able to identify after each *Delete-Min* operation whether the deleted vertex has been settled before which again causes $\Theta(m)$ additional cache misses, or be able to remove those extra *Decrease-Key* operations from the heap before they are extracted by a *Delete-Min* operation.

In [25] (see also [24]) Kumar & Schwabe presented an external-memory implementation of Dijkstra's algorithm for undirected graphs that allows spurious *Decrease-Key* operations to be performed

FUNCTION **B.3.** DIJKSTRA-EXT($G,\ w,\ s,\ d$)
{*Kumar & Schwabe's external-memory implementation of Dijkstra's algorithm [25]*}

1. perform the following initializations:

- (*i*) $Q \leftarrow \emptyset$, $Q' \leftarrow \emptyset$ {*Q and Q' are both external-memory heaps; Q supports Decrease-Key, Delete and Delete-Min while Q' supports only Insert and Delete-Min*}
- (*ii*) ***for*** each $v \in V[G]$ ***do*** $d[v] \leftarrow +\infty$
- (*iii*) DECREASE-KEY$_{(Q)}$($s,\ 0$) {*insert vertex s with key (i.e., distance) 0 into Q*}

2. ***while*** $Q \neq \emptyset$ ***do***

- (*i*) $(u,k) \leftarrow$ FIND-MIN$_{(Q)}$( ), $(u',k') \leftarrow$ FIND-MIN$_{(Q')}$( )
- (*ii*) ***if*** $k \leq k'$ ***then*** {*a new shortest distance (k) has been found* }
  - (*a*) DELETE$_{(Q)}$(u), $d[u] \leftarrow k$ {*k is the shortest distance from s to u*}
  - (*b*) ***for*** each $(u,v) \in E[G]$ ***do***
    DECREASE-KEY$_{(Q)}$($v,\ d[u] + w(u,v)$) {*relax edge (u,v)*}
    INSERT$_{(Q')}$($v,\ d[u] + w(u,v)$) {*guard for a possible spurious update on u*}

  else {*$k > k'$: shortest distance to u' has already been computed*}
  - (*a*) DELETE$_{(Q)}$(u'), DELETE-MIN$_{(Q')}$( ) {*remove spurious vertex u'*}

DIJKSTRA-EXT ENDS

Figure 7: Given an undirected graph G with vertex set $V[G]$ (each vertex is identified with a unique integer in $[1, |V[G]|]$), edge set $E[G]$, a weight function $w : E[G] \rightarrow \Re$ and a source vertex $s \in V[G]$, this function computes the shortest distance from s to each vertex $v \in V[G]$ and stores it in $d[v]$.

on the primary heap Q but uses a mechanism to remove those operations from Q using an auxiliary heap Q' (see Figure 7). This mechanism eliminates the need for identifying settled vertices directly and thus saves $\Theta(m)$ cache misses. The auxiliary heap only needs to support *Insert* and *Delete-Min* operations. The algorithm performs m *Decrease-Key* operations and about $n+m$ *Delete* operations on Q, and about m *Insert* and *Delete-Min* operations each on Q'.

The algorithm can be used to solve the SSSP problem on undirected graphs cache-obliviously in $\mathcal{O}\left(n + \frac{m}{B}\log m\right)$ I/O operations by replacing Q with a buffer heap and Q' with an auxiliary buffer heap [12].

In [12] we show how to implement Dijkstra's SSSP algorithm for directed graphs cache-obliviously in $\mathcal{O}\left((n + \frac{m}{B}) \cdot \log \frac{n}{B}\right)$ I/Os under the tall cache assumption. The implementation requires one additional data structure called the *buffered repository tree* [8] for remembering settled vertices. Since we performed our experiments mainly on sparse graphs for which implementations in Sections B.1 and B.2 give better bounds we have not considered this implementation.

I/O Cost of Accessing the Graph Data Structure Only			
Implementation	Accessing/updating tentative distances	Accessing adjacency lists	Total
DIJKSTRA-DEC	$n+m$	$n+\frac{m}{B}$	$2n+m+\frac{m}{B}$
DIJKSTRA-NODEC	$n+m+D$	$n+\frac{m}{B}$	$2n+m+\frac{m}{B}+D$
DIJKSTRA-EXT	*none*	$n+\frac{m}{B}$	$n+\frac{m}{B}$

Table 6: I/O complexity of different implementations of Dijkstra's algorithm for accessing the graph data structure only, where D ($\leq m$) is the number of *Decrease-Key*s performed by DIJKSTRA-DEC and B is the block size.

Number of Priority Queue Operations Performed				
Implementation	*Insert*	*Decrease-Key*	*Delete/Delete-Min*	Total
Dijkstra-Dec	n	D	n	$2n + D$
Dijkstra-NoDec	$n + D$	*none*	$n + D$	$2n + 2D$
Dijkstra-Ext	Primary (Q)			
	none	$2m$	$n + 2m$	$n + 4m$
	Auxiliary (Q')			
	$2m$	*none*	$2m$	$4m$
	Total (Q & Q')			
	$2m$	$2m$	$n + 4m$	$n + 8m$

Table 7: Number of heap operations performed by different implementations of Dijkstra's algorithm, where D ($\leq m$) is the number of *Decrease-Key*s performed by Dijkstra-Dec.

B.3.1 Discussion on the Relative Performance of the Three Implementations.

Table 6 lists the I/O cost of accessing the graph data structure by the three implementations while Table 7 lists the number of different heap operations performed by them.

We observe that Dijkstra-NoDec performs more I/O operations for accessing the graph data structure as well as more heap operations compared to Dijkstra-Dec. However, Dijkstra-NoDec can use a more efficient heap than Dijkstra-Dec since unlike Dijkstra-Dec it does not require the heap to support *Decrease-Key* operations. As a result Dijkstra-NoDec is likely to run faster than Dijkstra-Dec for in-core computations on very sparse graphs. For example, consider running the two implementations in-core on a large graph with $m = \Theta(n)$. In that case, $D \leq kn$ for some constant k and $\log n > B$, and Dijkstra-NoDec will run faster than Dijkstra-Dec provided it uses a heap that runs at least $1 + k$ times faster than the heap used by Dijkstra-Dec.

The external-memory implementation Dijkstra-Ext performs the smallest number of I/O operations for accessing the graph data structure than the other two implementations. However, this reduction in graph operations comes at the cost of considerably increasing the number of heap operations performed. For example, for $\mathcal{G}_{n,m}$ with average degree 8 for which $D \leq n$ typically holds, Dijkstra-Ext performs at least 6 times more heap operations than the other two implementations. However, if Dijkstra-Ext uses I/O-efficient heaps such as the buffer heap and the auxiliary buffer heap, and the block size B is sufficiently large then the I/O cost of performing the heap operations will no longer dominate its running time. In such a scenario (e.g., out-of-core computations) Dijkstra-Ext will outperform the other two implementations because of the relatively smaller number of graph operations it performs.

C Graph Classes Considered

We ran our experiments on three graph classes. The synthetic graphs were generated using generators (PR [31], GT [5]) contributed by *9th DIMACS Implementation Challenge* participants [1].

Undirected $\mathcal{G}_{n,m}$ (PR [31]). The $\mathcal{G}_{n,m}$ distribution chooses a graph uniformly at random from all graphs with n labeled vertices and m edges [18]. Such a graph can be constructed by choosing m random edges with equal probability (and with replacement) from all possible ($n \times n - n$) edges (the PR-generator avoids choosing self-loops).

Directed Power-Law Graphs (GT [5]). The GT-generator generates random graphs with power-law degree distributions and small-world characteristics using the *recursive matrix* (R-MAT) graph model [9]. The model has four non-zero parameters a, b, c and d with $a + b + c + d = 1$. Given the number of vertices n and the number of edges m, the GT-generator starts off with an empty $n \times n$ adjacency matrix for the graph, recursively divides the matrix into four quadrants, and distributes the edges to the top-left, top-right, bottom-left and bottom-right quadrants with probabilities a, b, c and d, respectively. It has been conjectured in [9] that many real-world graphs have $a : b \approx 3 : 1$, $a : c \approx 3 : 1$ and $a \geq d$, and accordingly we have used $a = 0.45$, $b = c = 0.15$ and $d = 0.25$ which are also the default values used by the GT-generator. The resulting graph is directed.

Undirected U.S. Road Networks ([34]). These are undirected weighted graphs representing the road networks of 50 U.S. states and the District of Columbia. Edge weights are given both as the spatial distance between the endpoints (i.e., the great circle distance in meters between the endpoints) and as the travel time between them (i.e., spatial distance divided by some average speed that depends on the road category). Merging all networks produces a graph containing about 24 million nodes and 29 million edges.

In Appendices D and E we present experimental results on the following additional graph classes. All the following graphs are undirected.

Regular Graphs (PR [31]). A graph is d-regular if all its vertices have the same degree d.

Grid Graphs (PR [31]). We used $\sqrt{n} \times \sqrt{n}$ grid graphs with uniformly distributed edge-weights.

Geometric Graphs (PR [31]). This graph class is a natural alternative to the classical $\mathcal{G}_{n,m}$ class. In contrast to $\mathcal{G}_{n,m}$ where each edge is chosen independently and with equal probability, a random geometric graph is constructed by randomly distributing a set of vertices over some metric space and then connecting two vertices with an edge if the distance between them is sufficiently small. Given the number of vertices n, we used the PR-generator to generate a random geometric graph by distributing these n points randomly in the unit square and connecting two vertices if they are within distance $\frac{1.5}{\sqrt{n}}$. The average degree of the resulting graph is about 7. The weight of an edge connecting two vertices corresponds to the distance between them.

Layered Graphs (MPI [2]). A d-layer random graph on n nodes consists of $d+1$ levels with level 0 containing a single node (called the source node) and each of the remaining d levels containing exactly $\frac{(n-1)}{d}$ nodes (assuming d divides $n-1$) [2]. The source node at level 0 is connected to every node at level 1, and for $1 \leq i < d$ each node at level i is connected to 3 random nodes at level $i+1$. The nodes in each level are connected with a Hamiltonian cycle. All edges are directed and inter-level edges are directed from lower to higher levels.

D Additional Experimental Results on Intel P4 Xeon

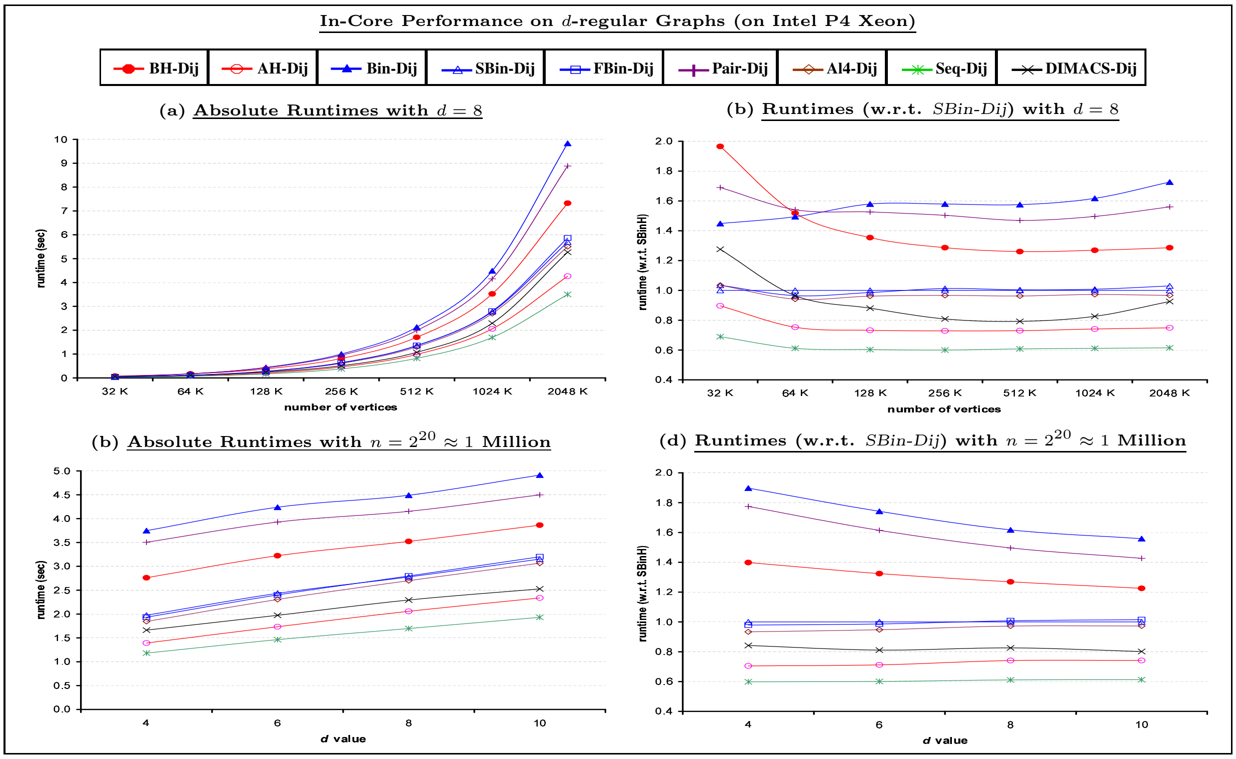

Figure 8: In-core performance of algorithms on d-regular graphs (on Intel P4 Xeon).

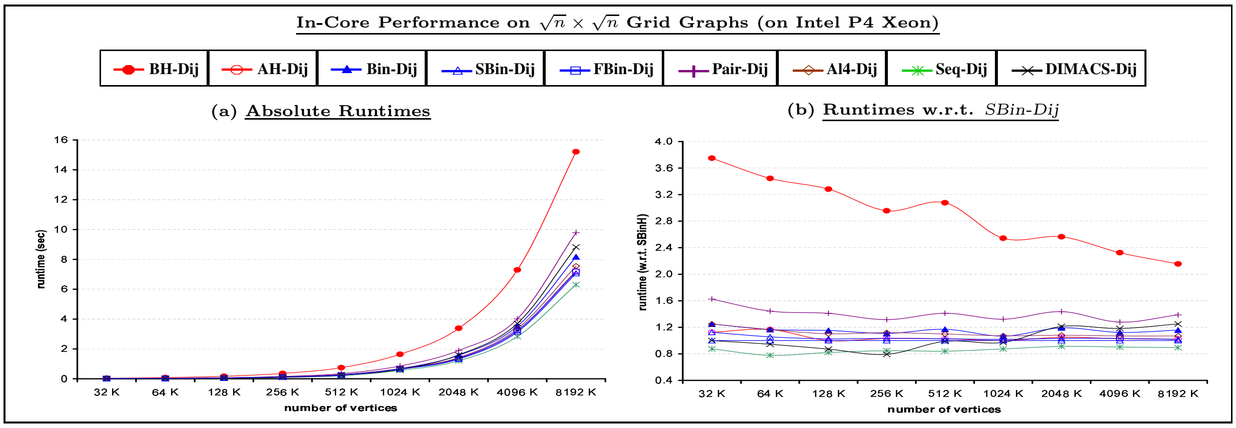

Figure 9: In-core performance of algorithms on $\sqrt{n} \times \sqrt{n}$ grid graphs (on Intel P4 Xeon).

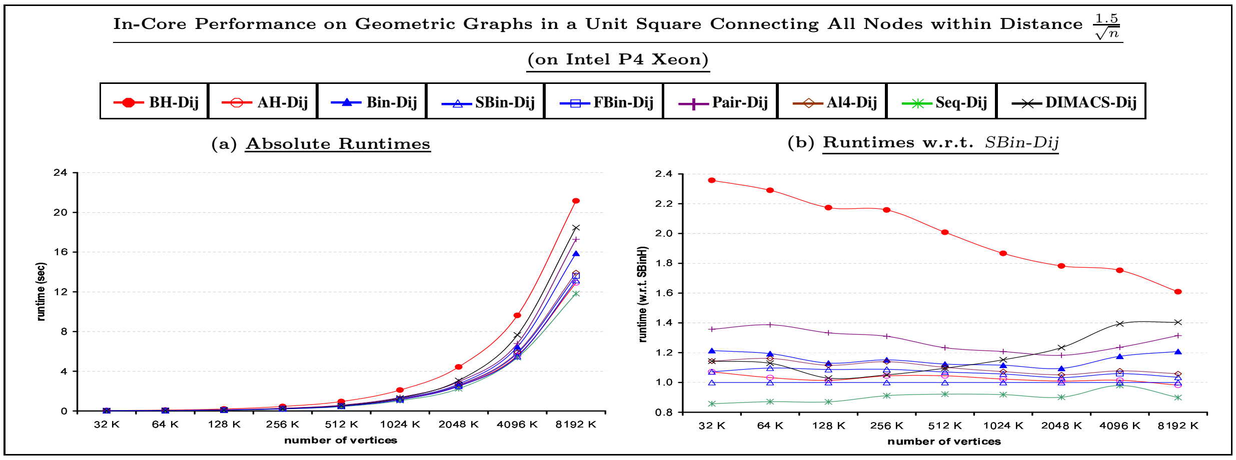

Figure 10: In-core performance of algorithms on geometric graphs in a unit square connecting all nodes within distance $\frac{1.5}{\sqrt{n}}$ (on Intel P4 Xeon).

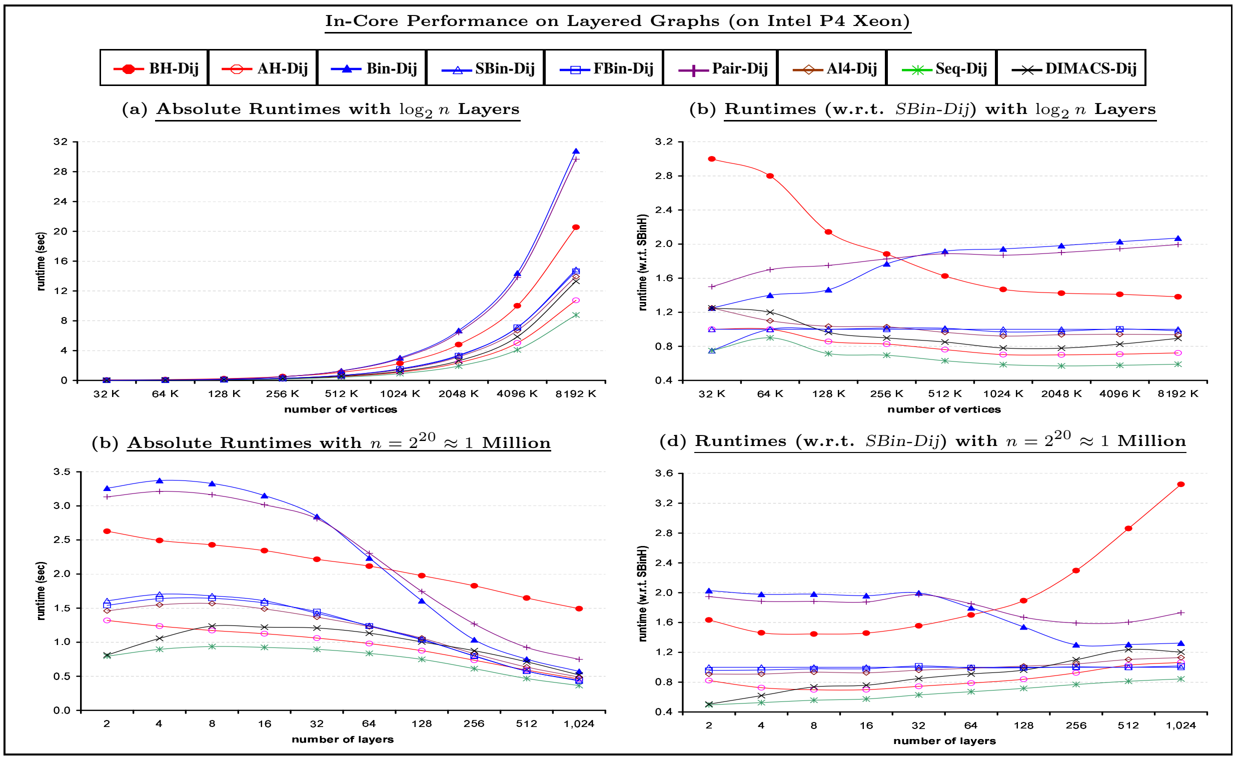

Figure 11: In-core performance of algorithms on layered graphs (on Intel P4 Xeon).

E Experimental Results on AMD Opteron

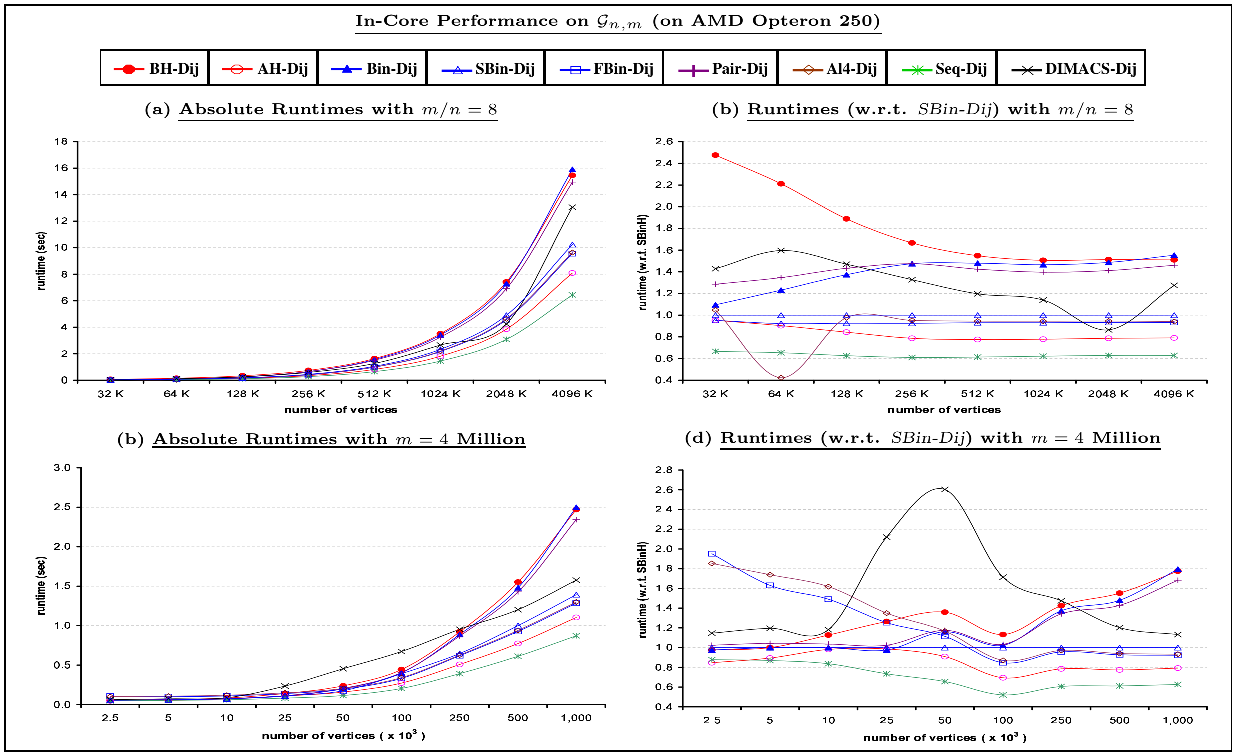

Figure 12: In-core performance of algorithms on $\mathcal{G}_{n,m}$ (on AMD Opteron 250).

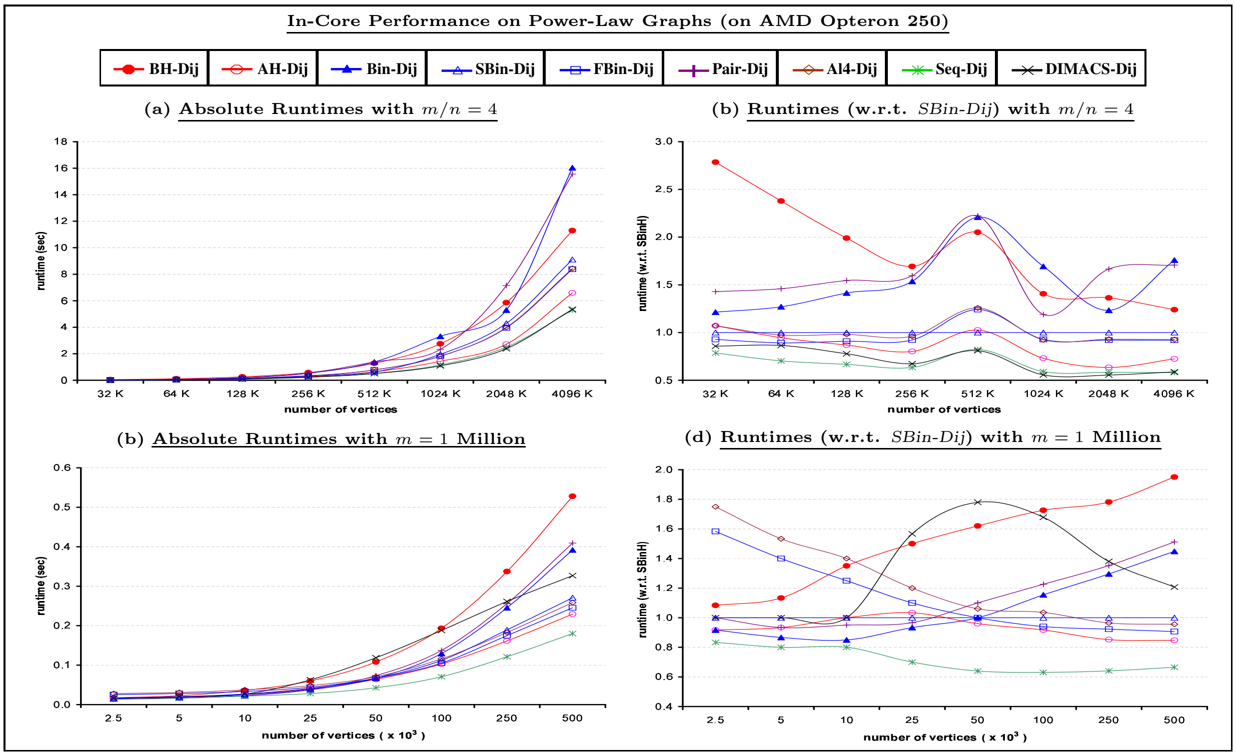

Figure 13: In-core performance of algorithms on power-law graphs (on AMD Opteron 250).

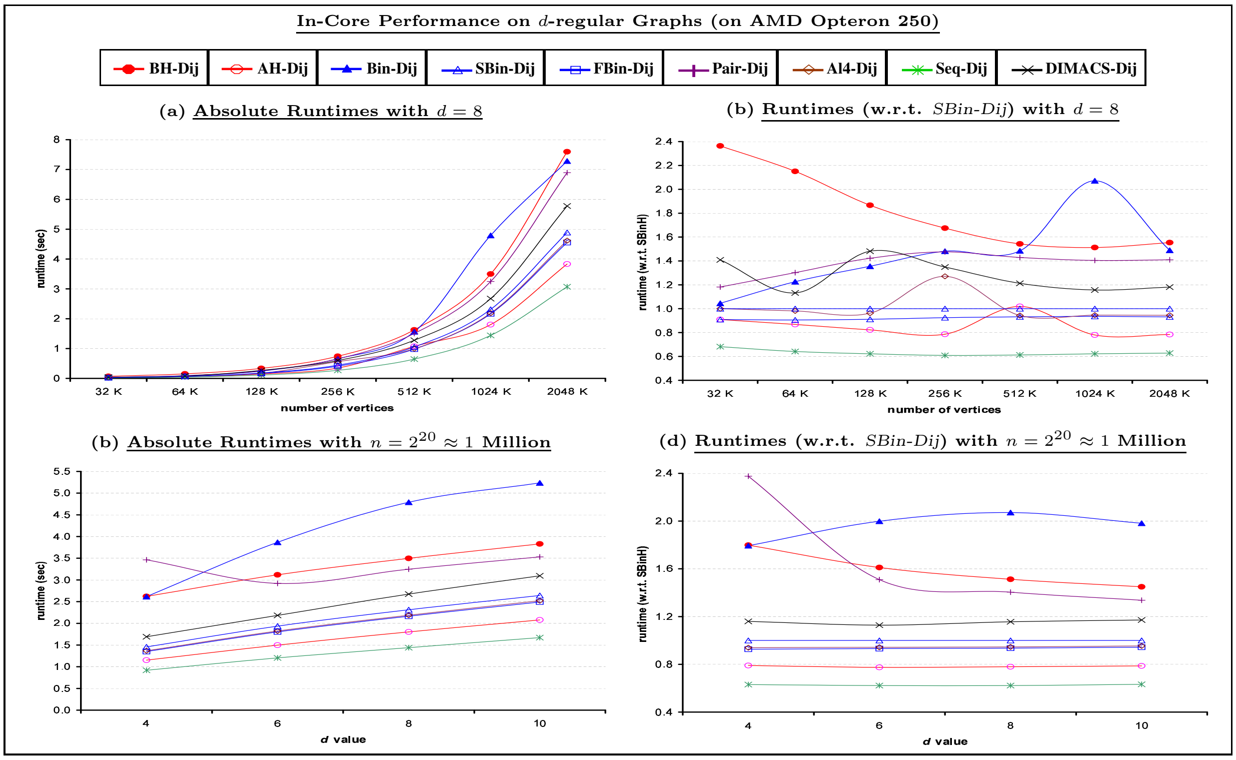

Figure 14: In-core performance of algorithms on d-regular graphs (on AMD Opteron 250).

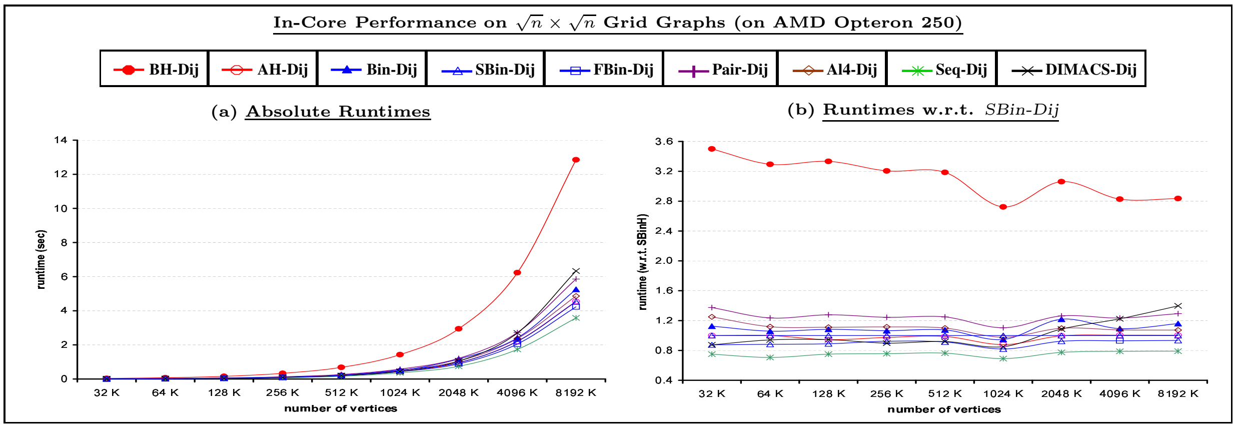

Figure 15: In-core performance of algorithms on $\sqrt{n} \times \sqrt{n}$ grid graphs (on Intel P4 opteron).

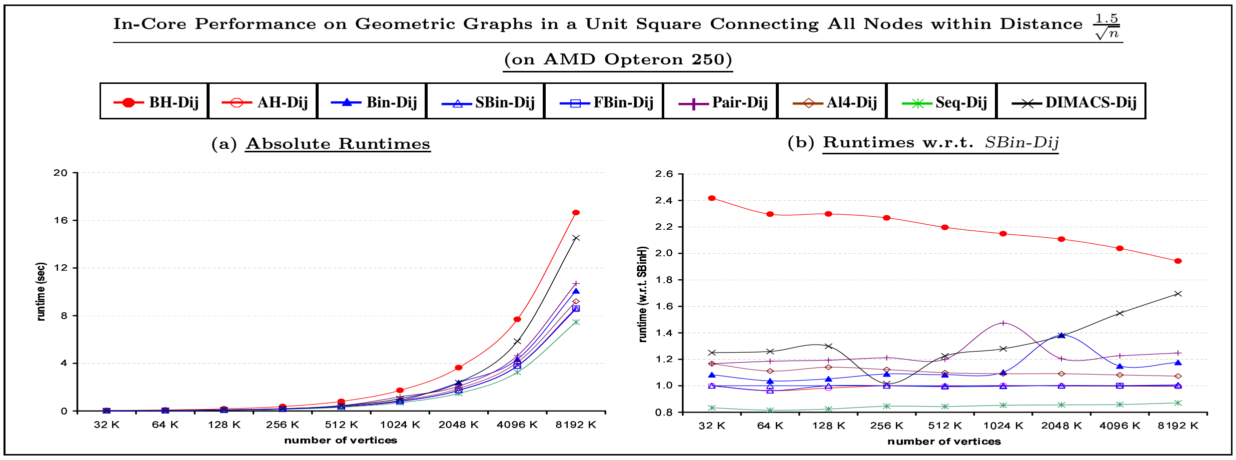

Figure 16: In-core performance of algorithms on geometric graphs in a unit square connecting all nodes within distance $\frac{1.5}{\sqrt{n}}$ (on Intel P4 opteron).

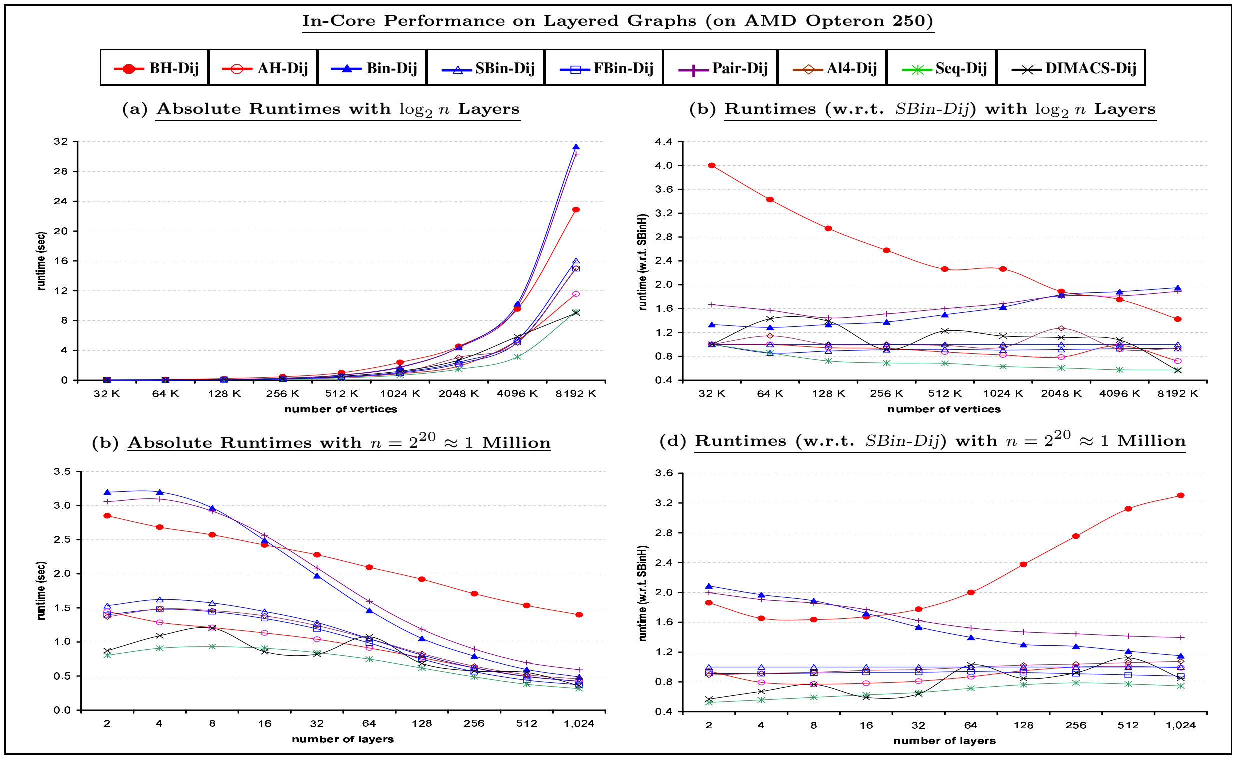

Figure 17: In-core performance of algorithms on layered graphs (on AMD Opteron 250).