Crafting Data Structures: A Study of Reference Locality in Refinement-Based Pathfinding*

Robert Niewiadomski, José Nelson Amaral, Robert C. Holte

{*niewiado, amaral, holte*}*@cs.ualberta.ca*
Department of Computing Science, University of Alberta
Edmonton, AB, Canada

Abstract. The widening gap between processor speed and memory latency increases the importance of crafting data structures and algorithms to exploit temporal and spatial locality. Refinement-based pathfinding algorithms, such as Classic Refinement, find near-optimal paths in very large sparse graphs where traditional search techniques fail to generate paths in acceptable time. In this paper we present a performance evaluation study of three simple data structure transformations aimed at improving the data reference locality of Classic Refinement. These transformations are robust to changes in processor architecture, memory hierarchy, and compiler optimizations. We tested our alternative designs on four contemporary architectures, using two compilers for each machine. In our experiments these techniques improved data reference locality resulting in performance improvements of up to 67% with consistent improvements above 15%. We investigated the source of these improvements and identified the elimination of TLB and cache misses as the dominant factors.

1 Introduction

Pathfinding in graphs is an important problem with applications in many industries such as computer games, freight transport, travel planning, circuit routing, network packet routing, etc. For instance, in the Real Time Strategy (RTS) video-game genre refinement-based search and its variants are used to conduct pathfinding for movements on the game map [15]. In these games pathfinding consumes up to 50% of total computation time [11, 26]. Using a modification to Dijkstra's algorithm, the shortest path between two vertices in a graph $G(V, E)$ can be computed in $O(E \log V)$ [10]. However, in some time sensitive applications where $|V|$ is very large we may want to visit only a fraction of the vertices in V to find an approximation of the shortest path [19]. *Refinement-based search* (RBS) is often used to restrict the search space to generate quality paths [18]. *Classic Refinement* (CR), a variation of RBS, partitions a large graph into many subgraphs, and generates an *abstract graph* that describes the interconnections among the subgraphs. A path between two vertices, u and v in the original graph is found by: (1) identifying the vertices in the abstract graph that correspond to the partitions containing u and v; (2) finding a path, in the abstract graph, between the identified vertices; (3) using this abstract path to find a path in the original graph.

This paper presents a performance evaluation study of three techniques for improving the data reference locality of CR: (1) data duplication; (2) data reordering; (3) and merging of independent data structures into a common memory area. We demonstrate that combining these techniques can result in performance improvements of up to 67% with consistent improvements above 15%. Through analysis of hardware counter profiles, as well as memory access trace data, we demonstrate that these results stem from improved data reference locality at the page-level and to a lesser extent at the cache line level. By testing on four different architectures with GCC and vendor based compilers we also demonstrate that our techniques are robust to changes in hardware as well as the level of compiler optimization.

Section 2 presents a generic method for graph abstraction for refinement-based search as well as the Classic Refinement algorithm. Section 3 describes the baseline implementation and our three techniques. Section 4 presents our experimental framework while Section 5 contains the results and subsequent analysis of our experiments. Finally, Section 6 discusses related work.

* This research is partially funded by grants from the Natural Sciences and Engineering Research Council of Canada and by the Alberta Ingenuity Fund.

2 Abstraction and Search

2.1 Graph Abstraction

Let $G_0(V_0, E_0)$ be the input graph, also called the *source graph*, to a refinement-based search algorithm. Let $G_1(V_1, E_1)$ be an *abstraction* of G_0 as defined below. Both G_0 and G_1 are undirected and unweighted graphs.

G_0 is partitioned into connected subgraphs. The abstract graph G_1 must have one vertex for each subgraph of G_0. If a vertex v_i^0 in G_0 maps to a vertex v_p^1 in G_1, we say that v_p^1 is the *image* of v_i^0 at abstraction level 1 (Note: v_p^1 should be read as "vertex p at abstraction level 1"). We also say that the set of vertices in G_0 that map to vertex v_p^1 in G_1 is the *pre-image* of v_p^1. The abstract graph G_1 has an edge (v_p^1, v_q^1) if and only if there is an edge (v_i^0, v_j^0) in G_0 such that v_i^0 belongs to the pre-image of v_p^1 and v_j^0 belongs to the pre-image of v_q^1. This transformation ensures that paths in G_0 can be mapped to corresponding paths in G_1.

Because we can create an abstract graph for any undirected graph we can create an abstraction of an abstraction to generate an *abstraction hierarchy*. A sequence of graphs $\{G_0, G_1 \ldots, G_{n-1}\}$ is an abstraction hierarchy for source graph G_0 if for $0 \leq i < n-1$ G_{i+1} is an abstraction of G_i. An example of a three-level abstraction hierarchy is shown in Figure 1.

2.2 Abstraction Generation

To generate an abstraction hierarchy we use the "max-degree" STAR algorithm [18]. The input is the source graph G_0 and a radius r. Given a graph G_a, the first step is to generate an abstraction G_{a+1} by partitioning G_a into connected subgraphs. The STAR algorithm partitions G_a into subgraphs whose maximum diameter is at most $2r$. It selects a vertex v_i^a in G_a. If v_i^a is not in a subgraph, a new subgraph is created. Next a breadth-first traversal of G_a starting in v_i^a is used to find the other vertices of this subgraph. This traversal ignores vertices that already belong to another subgraph. The traversal stops at vertices that are r edges away from v_i^a. Once all vertices in G_a are assigned to subgraphs, we proceed to generate G_{a+1}.

2.3 Refinement-based Search

Refinement-based search algorithms use an abstraction hierarchy to find short paths between a start vertex and a goal vertex. Examples of refinement-based search algorithms are Classic Refinement, Path Marking, and Alternating Opportunism [18]. These algorithms vary in terms of efficiency and the quality of the paths generated. This paper focuses on Classic Refinement.

Definitions and Notations

Definition 1. *An ordered list of G_a vertices, $P = \{v_0^a, v_1^a, \ldots, v_{k-1}^a\}$, is a* **path** *in G_a if and only if G_a contains the edges (v_0^a, v_1^a), (v_1^a, v_2^a), …, (v_{k-2}^a, v_{k-1}^a). We use the notation $P[j]$ to refer to the j^{th} element in path P.*

Definition 2. *A path $P = \{v_0^a, v_1^a, \ldots, v_{k-1}^a\}$ in G_a is a* **constrained path** *if and only if it is the shortest path between v_0^a and v_{k-1}^a, such that vertices $v_0^a, v_1^a, \ldots, v_{k-1}^a$ belong to the pre-image of the same vertex v_p^{a+1}. Because the pre-image of v_p^{a+1} is a connected subgraph of G_a, when computing a constrained path, a search algorithm can restrict its search space to the vertices in the pre-image of v_p^{a+1}.*

In a constrained path all vertices are in the same pre-image. G_0 may contain a shorter path between v_0^a and v_{k-1}^a than the constrained path P, but any such path will contain at least one vertex outside the pre-image of v_p^{a+1}, and therefore will not be a constrained path.

Definition 3. *Let v_p^{a+1} and v_q^{a+1} be two vertices in G_{a+1} such that (v_p^{a+1}, v_q^{a+1}) is an edge in G_{a+1}. Let v_i^a be a vertex in the pre-image of v_p^{a+1}. Then there exist a path from v_i^a to any vertex in the pre-image of v_q^{a+1}. A* **constrained jump path** *J from v_i^a to the pre-image of v_q^{a+1} is the shortest path between v_i^a and any vertex in the pre-image of v_q^{a+1}, such that any edge traversed by J connects vertices that belong either to the pre-image of v_p^{a+1} or to the pre-image of v_q^{a+1}.*

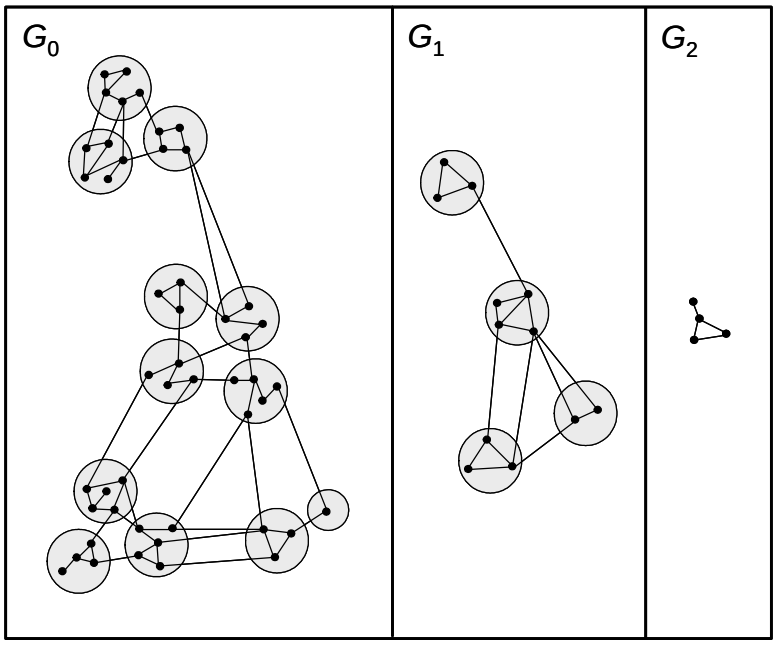


Fig. 1. A three-level abstraction hierarchy.

Again, a shorter path from v_i^a to the pre-image of v_p^{a+1} may exist in G_0, but it would have to include at least one vertex outside the pre-image of v_p^{a+1} or v_q^{a+1} and thus not be constrained.

```
ClassicRefinement(A, s^0, g^0, n)
1:  s^{n-1} ← LookupVertexImage(s^0, n − 1)
2:  g^{n-1} ← LookupVertexImage(g^0, n − 1)
3:  P_{n-1} ← FindPath(s^{n-1}, g^{n-1}, n − 1)
4:  if |P_{n-1}| = 0 then
5:      return NULL
6:  for i = n − 2 to i = 0
7:      P_i ← {}
8:      b ← LookupVertexImage(s^0, i)
9:      for j ← 0 to j = |P_{i+1}| − 1
10:         J ← FindConstrainedJumpPath(G_i, b, P_{i+1}[j + 1])
11:         P_i ←Append(P_i, J)
12:         b ←LastVertex(J)
13:     endfor
14:     g_i ← LookupVertexImage(g^0, i)
15:     C ← FindConstrainedPath(b, g_i, i)
16:     P_i ←Append(P_i, C)
17: endfor
18: return P_0
```

Fig. 2. Classic Refinement Algorithm.

Classic Refinement Figure 2 presents pseudocode for the Classic Refinement (CR) algorithm. Given a source graph G_0 and an abstraction hierarchy $A = \{G_0, G_1, \ldots, G_{n-1}\}$ we are interested in finding a path in G_0

between a source vertex s^0 and a goal vertex g^0. The CR algorithm starts by finding a path, P_{n-1}, between s^{n-1} and g^{n-1}, the images of the source and goal vertices in the highest level of the hierarchy, G_{n-1}. If no such path exists then the algorithm returns a $NULL$ path indicating that no path exists between s^0 and g^0 in G_0 (steps 1-5). If a path was found in G_{n-1}, CR iterates through each remaining level of abstraction, starting at G_{n-2} and going down the abstraction hierarchy to G_0 (for loop at step 6).

Let $P_{i+1} = \{s^{i+1}, v_1^{i+1}, \ldots, v_{k-2}^{i+1}, g^{i+1}\}$ be the path found in G_{i+1}. In order to compute the path P_i, CR initializes b to the image of s^0 at abstraction level i. CR then computes the constrained jump path J from b to a vertex in the pre-image of the next P_{i+1} vertex, $P_{i+1}[j+1]$ (step 10). By definition the last vertex in J is the first vertex in the pre-image of $P_{i+1}[j+1]$ visited by J. CR appends the constrained jump path J to P_i and updates b so that it is now the first vertex visited in $P_{i+1}[j+1]$ and iterates until the pre-image of g^{i+1} is reached.

Finally, when b is the initial vertex in the pre-image of g^{i+1}, CR computes a constrained path C between b and g^i, the image of g^0 in G_i (step 15) and appends C to P_i.

Once the algorithm reaches the bottom-most level of the abstraction and P_0 has been generated, the algorithm is finished and path P_0 is returned.

3 Data Structure Transformations

This section describes the data structure transformations that we propose to improve the performance of classic refinement. We use the graph shown in Figure 3 as an example in our presentation. We begin with a description of our baseline implementation.

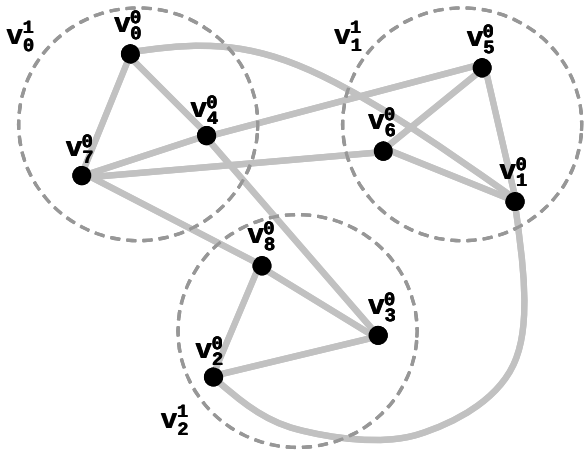


Fig. 3. Running Example.

3.1 Baseline

Our baseline implementation of CR is based on sound implementation techniques for sparse graph traversal algorithms. Because we are interested on graphs with a low degree of connectivity we utilize adjacency lists to represent graphs. Figure 4 shows the 32-bit fields in the data structure used to represent a vertex v_a^i in the baseline implementation. ID is the vertex unique identification, the traversal visit marker (TVM) indicates if the vertex has been visited, BP is a back pointer used to store the previous node visited. BP is used to reconstruct

ID	TVM	BP	IMG	DEG	Adjacency List		
v_0^0	0	*NULL*	v_0^1	3	v_1^0	v_4^0	v_7^0

Fig. 4. Fields in the data structure of a vertex.

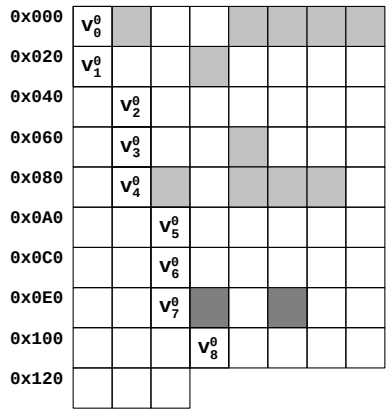


Fig. 5. Memory Layout without Vertex Clustering.

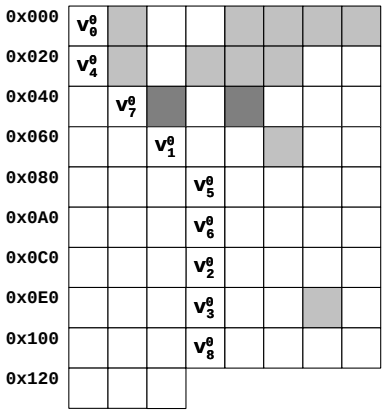


Fig. 6. Memory Layout with Vertex Clustering.

the path once the goal node is encountered. IMG is a pointer to the vertex's image, and DEG is the degree of v_a^i. Figure 4 illustrates the data structure for v_0^0 in Figure 3.

The TVM is used to determine if the vertex has already been visited in the current search. The TVM of every vertex is initialized to zero before any search takes place. A global search counter (GSC) is maintained. Whenever a vertex v_a^i is visited during z^{th} search, its TVM is set to z. When the z^{th} search completes, z is incremented. Therefore any vertex that has a TVM smaller than z during search z, has not been visited yet. An alternative design that would be very conservative in space usage would use a bit vector to represent the visited state, with one bit per vertex. Such a design has three disadvantages when compared with our TVM approach: (1) it requires bit vector manipulation, thus making the code more difficult to maintain; (2) it requires the re-initialization of the visited field before the start of each search; (3) the memory locations that contain the bits may be distant from the data structure that contains the vertex information in memory, and thus may not benefit from the spatial locality encountered when the TVM is embedded within the vertex structure. Because the TVM is a 32-bit integer, we do not have to reinitialize it unless we search more than four-billion times.

Through manual data placement we ensure that all vertices in a given graph are stored in a single contiguous region of memory with one vertex data structure immediately followed by another. Such an approach is advantageous compared to dynamically allocating memory on a per node basis. In addition to being faster, our manual data placement method ensures that the in-memory distance between vertices is consistent between different trials of the same experiment.

Constrained Path Finding Algorithm and Queue Representation We use Breadth First Search (BFS) to search for constrained paths and constrained jump paths. BFS stores vertices to be visited in a working queue. This queue is sometimes implemented as a circular buffer to save memory [10]. However, we found that the overhead of checking for wrap-around and overflow is high. Our approach eliminates this bookkeeping by allocating two buffers, each of which is large enough to accommodate a pointer to every vertex in the graph. During search we alternate between buffers, using one buffer to dequeue nodes discovered in the previous round of node expansion and the other buffer to queue up newly discovered nodes. Even though we could just as easily

use only one of such buffers to implement the queue for BFS, we found the two queues superior to a single queue. Using buffers instead of circular queues is a practice found in pathfinding engine implementations in video games [1, 15], as well as BFS based implicit graph search [21].

3.2 Vertex Clustering

Consider the input graph shown in Figure 3. A naive implementation of the abstraction generation algorithm stores the vertices in memory in the order in which they appear in the original graph. Figure 5 shows the resulting layout of the vertex data structures in memory. In this figure, each small box represents a 32-bit memory field. For convenience of drawing we present eight 32-bit fields per line. We identify the 32-bit field where the data structure corresponding to each vertex of Figure 3 starts. Consider the search for a constrained jump path from v_0^1 to v_2^1 starting at v_0^0 and ending at v_3^0 in the example Figure 3. The shaded areas in Figure 5 show the memory locations that are accessed in this search. The ligher shade denotes a single access while the darker shade marks a location accessed twice during the search. Besides the irregular memory access pattern shown in Figure 5, the baseline implementation also keeps a separate work queue, and therefore performs accesses to a separate region of memory that are interleaved with the accesses shown in Figure 5. These accesses not only do not benefit from spatial locality, they also are potential source of conflict misses in the data caches.

We use the technique of *vertex clustering* that consists of re-arranging the vertex data structures in memory, such that vertices that map to the same image are located in close proximity of each other in memory. The order in which vertices are arranged is ad hoc. Figure 6 shows a memory layout after one possible application of vertex clustering. The shaded areas in this figure are the memory locations that are accessed for the same constrained jump path search from v_0^1 to v_2^1 starting at v_0^0 and ending at v_3^0. Notice how the memory accesses are much closer to each other in memory. Though not addressed in this paper, we expect the benefits of vertex clustering to be more pronounced in abstractions generated with a larger radius.

3.3 Image Mapping

With the data structure shown in Figure 4 pathfinding exhibits poor spatial locality even after vertex clustering has been applied. For instance, consider the constrained jump path example in Section 3.2. In order to constrain search to vertices that map to v_0^1 we need to access the IMG field of each vertex encountered during search. As a result, in Figure 6 we see that the baseline implementation accesses memory locations that are far from the locations where we find data structures for vertices in the pre-image of v_0^1.

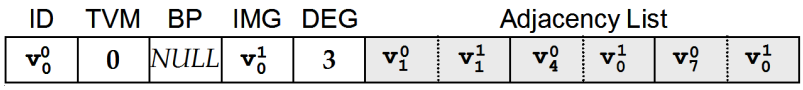


Fig. 7. Vertex data structure for abstract map.

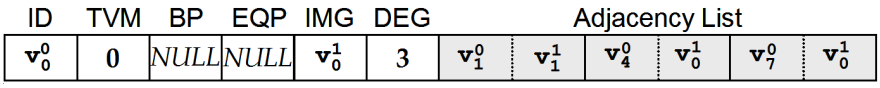


Fig. 8. Vertex data structure for abstract map with embedded queue.

The *image mapping* technique eliminates this unfavorable memory access pattern through an augmentation of the vertex data structure as shown in Figure 7. The change from Figure 4 is that the adjacency list contains

not only a pointer to the neighboring vertices, but also the IMG field of each neighbor. Thus when finding paths we do not need to access remote memory locations to determine the image of a given vertex.

3.4 Embedded Queue

The next source of poor memory reference pattern is the working queue of BFS. This queue is likely to reside in a remote memory region. Thus, a constrained path search exhibits interleaved accesses to two separate regions of memory: (1) the graph vertex region, and (2) the BFS working queue region. Frequent switching between two potentially distant memory regions has two adverse effects. First, it may cause *cache thrashing*, i.e., entries that will be used later are discarded because of memory conflicts. Second, it may prevent the memory accesses from benefiting from the free pre-fetching that most cache memories offer in the form of large cache lines. In other words, because of poor spatial locality in the memory accesses, the algorithm may not benefit from "free loading" of the vertex pointers in the working queue.

We propose the use of an *embedded queue* technique that keeps the information about vertices yet to be visited by BFS within the vertex's data structures. To implement a BFS embedded queue we augment the vertex data structure with an additional field, the embedded queue pointer (EQP), as shown in Figure 8. The EQP field of a vertex contains a pointer to the last vertex that was added to the working queue. We now present the modified constrained jump path search algorithm with these three modifications.

3.5 The Embedded Queue Constrained Jump Path Algorithm

```
EmbeddedQueueConstrainedJumpPath(G(V,E), s, I)
1:  EQP(s) ←NULL
2:  w ← s
3:  w' ← NULL
4:  while TRUE
5:      while w ≠ NULL
6:          for v ∈ V such that (w, v) ∈ E
7:              if Image(v) = I
8:                  BP(v) ← w
9:                  return v
10:             if Image(v) ≠ Image(s)
11:                 continue
12:             if TVM(v) = GSC
13:                 continue
14:             BP(v) ← w
15:             EQP(v) ← w'
16:             w' ← v
17:             TVM(v) ← GSC
18:         endfor
19:         w ← EQP(w)
20:     endwhile
21:     w ← w'
22:     w' ← NULL
23: endwhile
```

Fig. 9. Embedded Queue Constrained Path Algorithm with Abstract Map

An important component of CR is the algorithm that finds a constrained jump path from a start vertex s to any vertex in an image I. The constrained jump path algorithm assumes that if vertex s is in abstraction level a, then G_{a+1} has an edge between the image of s and the vertex representing I. Figure 9 presents the modified constrained jump path algorithm that uses embedded queues to store the work queue.

The pattern of vertex visitation in BFS can be viewed as an expanding wave that starts at the initial vertex. If we divide this expansion into phases, in phase 0 we visit the starting vertex s, in phase 1 we visit all the immediate neighbors of s. In phase 2 we visit all the vertices that are two hops away from the starting vertex, and so on. The embedded queue algorithm uses w to access the linked list formed by the embedded queue pointers (EQP) of the vertices that are being visited in the current phase. It uses w' to build the linked list of the vertices to be visited in the next phase.

When traversing a list in a given phase of the BFS algorithm, we use EQP to find the next vertex to be visited. In the initialization (steps 1-3) the EQP of the starting vertex s is assigned NULL to ensure that the phase 0 will terminate, and NULL is also assigned to w' to ensure that the next phase will also terminate. The first vertex of phase 0 is s. The algorithm will terminate when a vertex whose image is I is encountered (step 7). The data structure of each vertex contains an adjacency list, thus the accesses to the neighbors of v in the for loop (step 6) benefit from spatial locality. Vertices that are not in the same image as the starting vertex (step 10) or that have already being visited (step 12) are not included in the working list for the next phase.

The image mapping technique ensures that the comparison between the image of v and the image of the starting vertex s (step 10) accesses data internal to the data structure of v and thus benefits from spatial locality. With the use of the embedded queue technique the accesses to EQP (steps 15 and 19) are within the data structures of vertices v and w and thus also benefit from spatial locality.

The direction in which the embedded queue is constructed and traversed in the algorithm shown in Figure 9 matters. We build a backward queue in the sense that the newly discovered vertex v is placed at the front of w', not the rear. The advantage of this traversal direction is that when we finish building the queue, we start to visit vertices in the reverse order in which they were added to the queue. Thus we are likely to visit vertices that we have recently visited and benefit from temporal locality in the cache memories.

4 Experimental Framework

In order to test the effect of the techniques described in Section 3 we implemented eight versions of the constrained path finding algorithm. We extensively tested the performance of these implementations on four different machines using two compilers on each system and four regular graphs as input. This section describes the algorithm implementations, the machines and compilers used, the input graphs, and the conditions under which the various experiments are performed.

4.1 Our Implementations

Implementation	Embedded Queues	Vertex Clustering	Image Mapping
`Baseline`			
`Q--`	√		
`-V-`		√	
`--I`			√
`QV-`	√	√	
`Q-I`	√		√
`-VI`		√	√
`QVI`	√	√	√

Table 1. Techniques featured in each implementation.

Using ANSI C we wrote eight CR implementations. Each implementation features a different mix of our techniques of queue embedding (`Q`), vertex clustering (`V`), and image mapping (`I`). The `Baseline` implementation features none of our techniques. Other implementations are referred to by three character names, where each

character indicates either the presence or absence of a given technique. For example, `Q-I` features embedded queues and image mapping, but not vertex clustering. Table 1 shows how the implementations are named.

Feature		SGI	IBM	AMD	INTEL
Processor	Type	MIPS R12K	POWER3	2000+ XP	Pentium 4
	Clock Frequency	350 MHz	450 MHz	1667 MHz	2260 MHz
Data Cache L1	Capacity	32 KB	64 KB	64 KB	8 KB
	Associativity	2-Way	128-Way	2-Way	4-Way
	Line Size	32 bytes	128 bytes	64 bytes	64 bytes
Data Cache L2	Capacity	4 MB	8 MB	256 KB	512 KB
	Associativity	2-Way	4-Way	16-Way	8-Way
	Line Size	128 bytes	128 bytes	64 bytes	64 bytes
Data TLB L1	Capacity	56 Entries	256 Entries	32 Entries	128 Entries
	Associativity	Fully	2-Way	Fully	Fully
Data TLB L2	Capacity	*none*	*none*	256 Entries	*none*
	Associativity			4-Way	
VM Page Size		16 KB	4 KB	4 KB	4 KB
Data TLB Coverage		896 KB	1,024 KB	1,152 KB	512 KB
TLB Miss Handler		Software	Hardware	Hardware	Hardware
DRAM Capacity		1 GB	1 GB	1 GB	1 GB
Compilers		MIPSpro (7.2.1) GCC (2.7.2)	IBM XLC (6.0) GCC (2.9)	Intel (6.0) GCC (2.96)	Intel (6.0) GCC (2.96)
Hardware Event Measurement Library		Perfex	PMAPI	PMC	PAPI

Table 2. Systems and compilers used in our experiments.

4.2 Experiments and Hardware

We conducted experiments on four systems based on different processors: `SGI` (MIPS R12K), `IBM` (IBM Power3), `AMD` (AMD Athlon XP) and `INTEL` (Intel Pentium 4). On each system we used GCC and the processor vendor's compiler to build our implementations with `-O0` and `-O3`.[1] Unless specified otherwise, all results presented in this paper were obtained with `-O3` and a vendor based compiler, except in the case of the `SGI` system where GCC was used. Detailed specifications of each system as well as information about compilers can be found in Table 2.

All measurements encompassed the computation of $10,000$ paths between random pairs of vertices.[2] Computing each path includes: (1) searching for the path and (2) reconstructing the path into a linked list via a BP pointer traversal.[3] Our main performance metric is execution time (measured as wall-clock time). We also used hardware event counters to measure various processor and memory events.[4] All execution times and hardware event totals were obtained on an unloaded machine. We always report the best result from several runs of each experiment.

We also collected traces of memory accesses made to heap memory associated with the abstraction hierarchy and, when applicable, with the queue vectors. This additional data was collected in separate runs to prevent interference with the time and hardware event measurements. This data enables us to construct various metrics. For instance, we computed the average number of distinct memory blocks accessed during search. By using block sizes corresponding to the cache line and page capacities found in our systems, and assuming minimal amounts

[1] On the `AMD` we used the Intel C compiler as the processor vendor's compiler.
[2] We used a low-overhead portable and deterministic random number generator described in[27].
[3] Reconstruction of paths took between 1% and 17% of the execution time.
[4] We used hardware event measurement libraries (See Table 2).

of data reuse between successive searches, we can approximate the amount of traffic at each level of the memory hierarchy.

We tried to minimize code discrepancies between implementations. The main changes from the `Baseline` are as follows: (1) for vertex clustering the code stays the same; (2) for image mapping we slightly alter the lines of code that determine the image of a neighbor vertex; (3) for embedded queues, however, we had to replace the queue vector code for a pointer manipulating code that traverses the embedded queues.

4.3 Input Graphs

The core our experiments were performed using four types of input graphs. We considered several instances of each graph type:

`2D-Grid`**:** A two-dimensional grid with a height of h and a width of w. A `2D-Grid` has $h \times w$ vertices. Except for border vertices, a vertex located at (x, y) has edges to all vertices located at $(x + a, y)$ and $(x, y + b)$, where $a, b \in (-1, 1)$, thereby yielding an average vertex degree of approximately 4. We made $h = w$ and varied its value from $h = 256$ to $h = 1,204$ by increments of 128.

`2DD-Grid`**:** Same as `2D-Grid`, but the average vertex degree is approximately 8 due to the introduction of diagonal edges to all vertices located at $(x + a, y + b)$, where $a, b \in (-1, 1)$.

`3D-Grid`**:** A three-dimensional grid with height h, width w, and depth d. A `3D-grid` has $h \times w \times d$ vertices. Except for border vertices, a vertex located at (x, y, z) has edges to all vertices located at $(x + a, y, z)$, $(x, y + b, z)$, and $(x, y, z + c)$, where $a, b, c \in (-1, 1)$, thereby yielding an average vertex degree of approximately 6. We made $h = w = d$ and varied h from $s = 48$ to $s = 96$ by increments of 8.

`3DD-Grid`**:** Same as `3D-Grid`, but the average vertex degree is approximately 26 due to the introduction of diagonal edges. A vertex (x, y) has an edge to all vertices, except for itself, located at $(x + a, y + b, z + c)$, where $a, b, c \in (-1, 0, 1)$.

Graph Type	Size	G_0		G_1		G_2		G_3		G_4		G_5		G_6		G_7		G_8	
		$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$
`2D-Grid`	1,024	1,048,576	2,096,104	196,779	559,645	24,480	71,371	2,973	8,578	392	1,039	68	137	9	14	3	2	1	0
`2DD-Grid`	1,024	1,048,576	4,188,162	116,964	465,806	12,996	51,302	1,444	5,550	169	600	25	72	4	6	1	0	-	-
`3D-Grid`	96	884,736	2,626,560	116,784	644,850	9,369	63,754	699	3,375	154	281	68	67	1	0	-	-	-	-
`3DD-Grid`	96	884,736	11,254,460	32,768	398,908	1,331	14,230	64	468	8	28	1	0	-	-	-	-	-	-

Table 3. Number of vertices and edges at each level of the abstraction hierarchy for the largest instance of each graph. A dash indicated the absence of a level.

Graph Type	Size	G_0	G_1	G_2	G_3	G_4	G_5	G_6	G_7	G_8
`2D-Grid`	1,024	4.0	5.7	5.8	5.8	5.3	4.0	3.1	1.3	0
`2DD-Grid`	1,024	8.0	8.0	7.9	7.7	7.1	5.8	3.0	0	-
`3D-Grid`	96	5.9	11.0	13.6	9.7	3.6	2.0	0	-	-
`3DD-Grid`	96	25.4	24.3	21.4	14.6	7.0	0	-	-	-

Table 4. Average vertex degree at each level of the abstraction hierarchy for the largest instance of each graph. A dash indicated the absence of a level.

Collectively, these graphs are representative of graphs with two-dimensional and three-dimensional topologies. For example, `2D-Grid` and `2DD-Grid` graphs are similar to graphs representing RTS game-worlds and

road-maps. `3D-Grid` and `3DD-Grid` graphs on the other hand are similar to graphs used to represent three-dimensional objects, such as buildings and bridges. These grids are *empty* in the sense that they are not populated with obstacles as a grid in a computer game would be. In an empty grid the shortest path between any two points is always a straight line. However, our pathfinding algorithm does not use this knowledge. By using empty graphs to study our techniques we avoid side-effects caused by irregularities in the graphs. For completeness, in Section 5.6 we present a performance oriented case study using non-empty graphs of realistic terrains.

All graphs are connected thereby enabling the search for a path between two randomly selected vertices. To generate abstraction hierarchies we utilized the STAR method with a radius of 2. To create G_0 we read the input graph and place vertices in memory in the order in which they are read-in (see Section 5.1). To generate G_{i+1} we iterate through vertices in G_i in the order in which they appear in memory, selecting yet to be classified vertices as starting points for new subgraphs. Newly generated G_{i+1} vertices are placed in memory in the order they are created. We repeatedly apply the abstraction generation process until the resulting abstraction level has a single vertex. For implementations with vertex clustering, the vertex clustering is performed after the abstraction hierarchy is generated. Table 3 contains a summary of the abstraction hierarchy of the largest instance of each graph type in terms of vertex and edge totals present at each level of abstraction while Table 4 does the same for the average vertex degree.

5 Results and Analysis

Our experimental study findings can be summarized as follows:

- The combination of vertex clustering and image mapping (`-VI`) can reduce baseline execution times by upwards of 43%. If vertices in the input graph are not ordered according to their vicinity, `-VI` can reduce baseline execution time by as much as 67%.
- We observed that the reduction in execution time is correlated with the reduction in TLB misses, and to a lesser degree to cache miss reductions. This observation supports the assertion that our techniques improve search performance through improved data reference locality.
- The exeuction time improvements are robust to changes in the hardware architecture, compilers, and in the level of compiler optimization.
- Although the addition of queue embedding to `-VI` produces performance improvements in some system and input graph combinations, we do not recommend indiscriminate use of the queue embedding technique.
- While some of our techniques increase the memory space required to store the abstraction hierarchy, they can also reduce the dynamic memory footprint of search, thereby decreasing memory subsystem traffic as a whole, especially at the page level.
- The results of a case study involving graphs representing realistic terrains suggests that our techniques are applicable to improving pathfinding performance in a variety of graphs.

5.1 Initial Vertex Order

During our investigation we found that the order in which vertices appear in the input graph can have a significant effect on the performance gains produced by our techniques. In our experiments, with the absence of vertex clustering, the order in which G_0 vertices appear in memory reflects the order in which they appear in the input graph. In addition, due to the nature of our abstraction generation process, the vertex order present in the input graph may also influence the order of vertices at higher levels of abstraction. Thus, the order in which vertices appear in the input graph can have an effect on data reference locality by affecting the in-memory proximity of vertices belonging to the same pre-image. As a means of exploring this effect we consider two types of input graph vertex orderings for each graph type. In the *default vertex order* (`DVO`), vertices are ordered in a manner that we believe captures the natural vertex order for the given graph type. The `DVO` of each graph type is as follows: `2D-Grid` and `2DD-Grid` vertices are ordered as left-to-right rows stacked on top of each other in

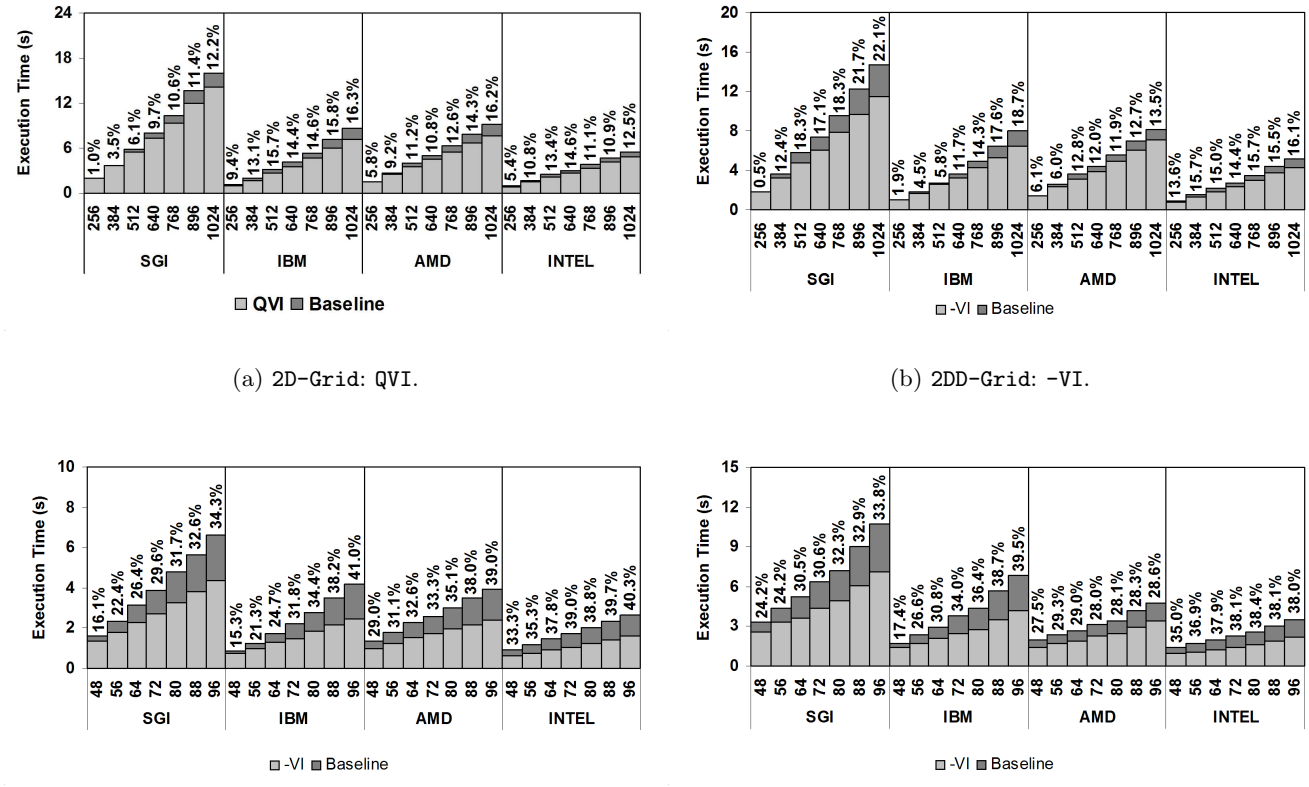
System	Version	DVO				RVO			
		2D-Grid	2DD-Grid	3D-Grid	3DD-Grid	2D-Grid	2DD-Grid	3D-Grid	3DD-Grid
SGI	Q--	3.0	-1.2	2.3	-16.4	-1.0	-1.8	2.0	0.6
	-V-	0.7	10.7	4.4	2.3	54.9	56.5	51.6	56.6
	--I	-2.8	0.7	14.7	24.0	18.9	19.5	33.6	40.0
	QV-	6.2	8.3	6.9	-13.3	52.9	54.0	49.8	51.3
	Q-I	-2.4	-6.0	11.3	0.7	17.1	17.5	31.2	38.0
	-VI	**15.5**	**22.1**	**34.3**	**33.8**	**65.8**	**62.1**	**67.8**	53.9
	QVI	12.2	8.8	30.2	5.1	62.6	58.5	64.9	**57.2**
IBM	Q--	3.4	3.8	-6.0	-9.2	-2.6	-2.3	-1.1	-1.0
	-V-	0.8	14.9	9.3	3.5	52.2	51.3	45.3	45.9
	--I	-9.5	-9.4	14.8	27.5	9.1	9.8	25.4	30.9
	QV-	6.3	13.8	10.2	-7.0	50.5	55.2	44.1	43.5
	Q-I	-4.9	0.1	17.1	14.8	7.3	11.0	23.9	30.5
	-VI	5.1	18.7	41.0	**39.5**	61.7	58.5	63.9	56.0
	QVI	**16.3**	**22.7**	**43.8**	26.9	**62.6**	**61.3**	**65.2**	**56.7**
AMD	Q--	1.2	-1.8	-1.0	-20.3	-2.9	-5.1	-6.3	-6.0
	-V-	4.4	10.0	11.4	5.0	42.8	46.3	34.9	48.2
	--I	-4.1	-3.7	18.7	17.7	8.4	1.2	19.9	21.9
	QV-	7.3	3.7	9.4	-14.0	40.4	41.5	31.0	30.7
	Q-I	-3.5	-7.2	17.0	-8.4	5.7	-3.2	16.3	16.8
	-VI	13.2	**13.5**	**39.0**	**28.6**	**54.2**	**51.8**	**55.5**	**54.0**
	QVI	**16.2**	7.6	37.2	3.3	52.5	47.8	52.6	47.1
INTEL	Q--	-0.7	2.7	-3.4	-9.1	-4.9	-5.9	-5.5	-8.8
	-V-	2.0	17.3	9.9	7.9	48.7	48.6	41.9	40.9
	--I	-7.2	-8.2	15.2	24.6	15.5	10.5	26.7	20.6
	QV-	4.9	15.0	7.6	-1.4	45.8	44.4	38.9	-8.8
	Q-I	-7.2	-4.1	12.9	11.6	12.9	4.2	23.0	13.9
	-VI	10.8	16.1	**40.3**	**38.0**	**58.9**	**52.9**	**63.1**	**55.8**
	QVI	**12.5**	**17.3**	37.6	24.6	57.0	50.6	61.3	47.3

Table 5. Percentage improvement over `Baseline` for seven combinations of the data layout techniques on four machines, using the largest instance of each graph type. The best improvement for each machine/graph combination is displayed in bold text.

a top-to-bottom manner, `3D-Grid` vertices are organized as front-to-back ordered instances of `2D-Grid` graphs, and `3DD-Grid` graph vertices are organized as front-to-back instances of `2DD-Grid` graphs. The *randomized vertex order* (`RVO`) of each graph type is generated by randomly scrambling the vertex order of `DVO`.

Performance gains achieved for `DVO` are indicative of the gains one can expect from the techniques presented in this paper. We use `RVO` to explore the upper bounds of the performance gains achievable with our techniques. Arbitrary vertex orderings such as `RVO` may seem artificial, after all, programmers don't tend to intentionally scramble the vertex order of their two-dimensional grids. However, arbitrary vertex orders can occur in practice. For example, consider a graph representing a national road-system where vertices correspond to cities and junctions. If the vertices appear in the input graph in an order that reflects an alphabetical sort of their labels (*i.e.,* city names), their ordering might have little correlation with their geographic proximity. Previous work has shown that it is beneficial to place vertices in memory in an order based on a sort of their geographical locations [14]. Comparing the `RVO` with the `DVO` performance gains in Table 5, we can confirm that observation. For the remainder of the experiments and analysis in this paper, we use `DVO` exclusively.

5.2 Execution Time



(a) `2D-Grid`: `QVI`.

(b) `2DD-Grid`: `-VI`.

(c) `3D-Grid -VI`.

(d) `3DD-Grid -VI`.

Fig. 10. Execution time for the best performing implementation for each graph type compared with the `Baseline` implementation. The top of each bar is annotated with the corresponding percentage reduction in execution time.

We begin by exploring how the performance of our various techniques compare with the `Baseline`. Which one is the best performing implementation? To determine the best performing implementation we consider the execution times of all implementations for experiments with `DVO` input graphs. From the results shown in Table 5, overall, `-VI` is the best performing implementation. With the exception of `2D-Grid` graphs, `-VI` generally produces the best performance improvements over `Baseline`. In the case of `2D-Grid` graphs however, `QVI` outperforms `-VI`.

To illustrate the performance gains achieved by `QVI` for `2D-Grid` graphs and those achieved by `-VI` for `2DD-Grid`, `3D-Grid`, and `3DD-Grid` graphs we present Figures 10(a) to 10(d). These figures showcase the percentage reductions of execution time produced by the best performing implementation for each graph type in comparison with the `Baseline` implementation. Each figure presents results for all sizes of the given graph type using all four systems. The bars are composed of two segments stacked on top of each other. The lighter segment denotes the execution time registered by `QVI`, in the case of Figure 10(a), and `-VI` in the case of Figures 10(b) through 10(d), while the darker segment corresponds to the additional execution time required by `Baseline`. The top of each bar is annotated with the magnitude of the performance improvement. Performance gains vary from graph type to graph type. For `2D-Grid` and `2DD-Grid` graphs the performance gains range from 0.5% to 22.1%, while for `3D-Grid` and `3DD-Grid` graphs the performance gains range from 15.3% to 40.3%. In all instances however, the performance gains are positive and generally consistent across hardware platforms for each graph type. This indicates that `QVI` and `-VI` are robust with respect to changes in the hardware architecture.

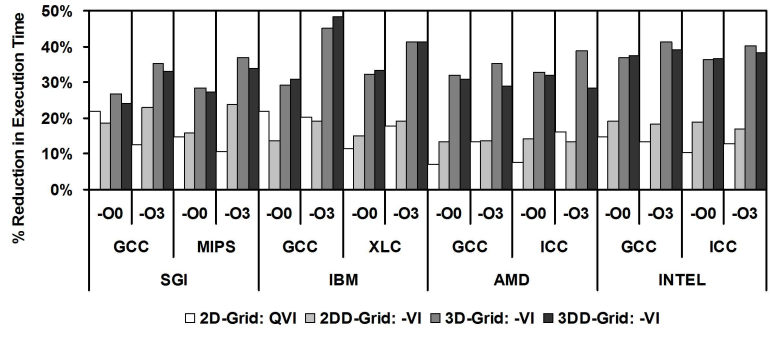


Fig. 11. As study of the percentage reduction in execution time produced by `QVI` or `-VI` over `Baseline` for the largest instance of each graph type. Results are presented for all systems, compilers, and two levels of optimization, `-O0` and `-O3`.

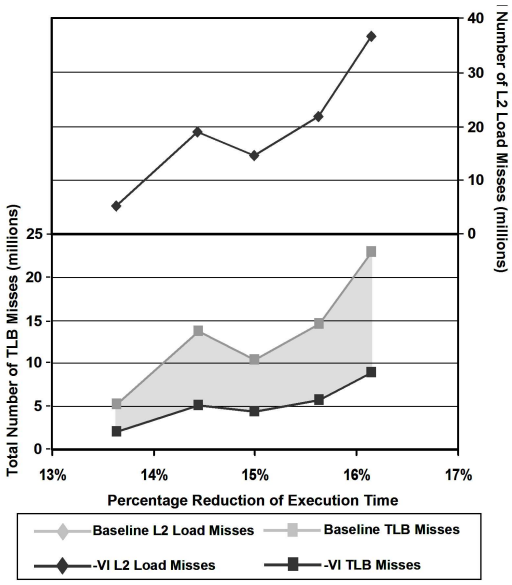
Are `QVI` and `-VI` robust to compiler changes as well? In short the answer is *Yes*. Figure 11 is a study of the percentage reduction in execution time produced by `QVI` or `-VI` over `Baseline` for the largest instance of each graph type. Results are presented for all systems, compilers, and two levels of optimizations, `-O0` and `-O3`. In all instances `QVI` and `-VI` produce non-trivial performance gains over `Baseline`, with improvments being generally similar for GCC and the vendor based compiler. Because of the relative similarity of the performance gains obtained with `-O0` and `-O3`, it would seem that the manner in which `QVI` and `-VI` improve performance is orthogonal with respect to compiler driven optimizations.

5.3 Data Reference Locality Improvement

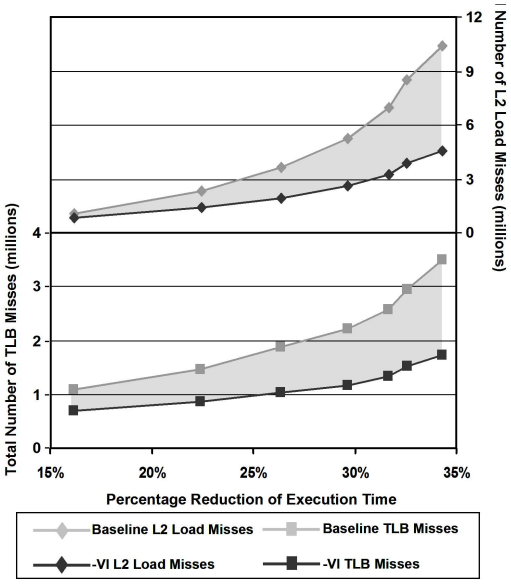
Using hardware event counters for runs presented in Figures 10(a) to 10(d), we examined the effectiveness of our best performing implementations in terms of their effect on data reference locality during search.

Figures 12(a) through 12(d) compare `Baseline` and the best performing implementation for select graph/system combinations in terms of the total number of L2 data cache and TLB misses. In each figure we consider all

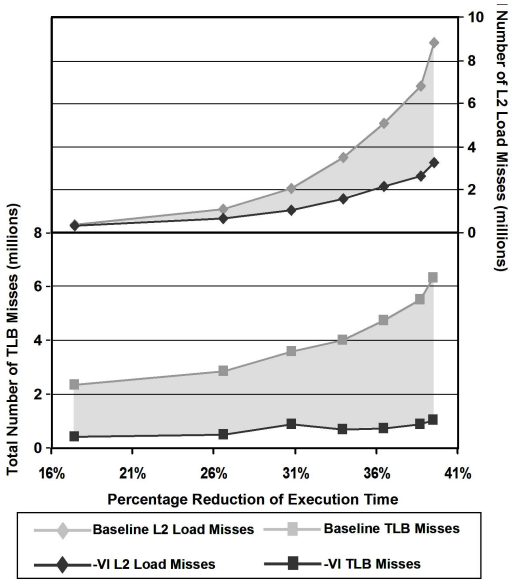
(a) `AMD:2D-Grid`...The AMD machine features two levels of TLB cache. This figure shows the L2 TLB miss totals.



(b) `INTEL:2DD-Grid`.



(c) `SGI:3D-Grid`.



(d) `IBM:3DD-Grid`.

Fig. 12. Study of the correlation between execution time reduction, reduction in TLB misses and changes in L2 cache misses for select implementations.

graph sizes and we order the measurements, along the horizontal axis, according to the percentage reduction in execution time. All figures show a correlation between the reduction in TLB misses and the reduction in execution time. This result highlights that page level data reference locality improves with our techniques and is responsible for some of the performance improvement.

Figures 12(a) and 12(b) show that data reference locality at the cache line level generally did not improve in the case of `2D-Grid` and `2DD-Grid` graphs.[5] We suspect that because the abstraction hierarchy levels of two-dimensional grids have lower average vertex degrees (see Table 4) than three-dimensional grid abstraction hierarchy levels, there is less opportunity to eliminate secondary cache misses with respect to the baseline implementation. On the other hand, Figures 12(c) and 12(d) show significant improvements in data reference locality at the cache line level for `3D-Grid` and `3DD-Grid` graphs. Collectively, these figures present a non-trivial link between performance gains and data reference locality improvements, especially at the page level.

5.4 Should Queue Embedding be Used?

Table 5 shows that the combination `-VI` produces definite improvements on all the machines and graphs studied. The advantage of queue embedding, however, is questionable at best. We originally proposed queue embedding with the intuition that placing the information stored in the queue in the same memory region as the vertex information would prevent frequent switches between two distant memory regions and thus improve performance. However, the execution time improvements presented in Table 5 indicate that queue embedding is not advantageous for several graphs and systems. Figure 13 underscores this point. This graph shows the percentage change in the execution time when embedded queue is added to a `-VI` implementation. In many instances adding queue embedding to `-VI` significantly hurts performance. As this result goes against our original intuition, we want to investigate why this is the case. Some immediate observations are: (1) `Q` is detrimental to the `3DD-Grid` on all systems; (2) for the remaining graphs, the influence of `Q` on performance is very different on the `IBM` than on the other systems.

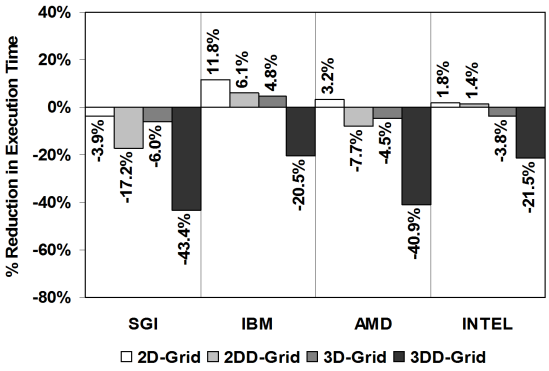


Fig. 13. Percentage reduction in execution time when `Q` is added to `-VI`.

Table 6 shows the variation in primary (DC) and secondary (DSC) cache misses, TLB misses, and number of instructions graduated for the `IBM` and `SGI` systems when `Q` is added to `-VI`. The number of TLB misses drops significantly when queue embedding is used, confirming our speculation of higher locality at the page level when the data accesses are not switched frequently from the area where the vertex's data is stored to the separate area where the queue is stored. However, embedding the queue results in a much higher rate of cache misses both for the primary and secondary caches. Our best explanation for this effect is that queue embedding accesses

[5] In fact, we would occasionally witness L2 cache misses increase in experiments involving `2D-Grid` and `2DD-Grid` graphs.

suffer from the "pointer chasing" problems, *i.e.,* a field in the current vertex must be accessed before the next element of the queue is known. In general, compilers have a hard time optimizing code exhibiting this kind of data access pattern [30, 9]. On the other hand, a vector-based queue lends itself well to optimizations both by the compiler and by the underlying hardware. For example, non-binding prefetch instructions can be issued by the hardware and/or inserted by the compiler. Another consideration is that a vector-based queue benefits from wide cache lines, in that a single cache line can contain several queue entries whereas with embedded queues a single cache is not likely to contain more than one or two queue entries. Queue embedding also results in a larger number of instructions graduated for graphs with a high degree of connectivity (high ratio of edges per vertex) such as `2DD-Grid` and `3DD-Grid`. In such graphs the working queue is much larger than in `2D-Grid` and `3D-Grid`. This empirical evidence is indicative of the code generator having less success with the linked-list nature of the queue embedded code.

We also experimented with comparisons between compilation at levels `-O0` and `-O3` on the `SGI` machine and discovered that the compiler eliminates approximately 10% more stores when operating at a higher level of optimization for `-VI` than for `QVI`. This is further evidence that the code generated for `QVI` is more difficult to optimize.

In summary, we do not recommend the use of queue embedding for graphs that have high connectivity because of the difficulty that compilers have optimizing the code. This lack of efficient optimization manifests itself in higher primary and secondary cache misses, and a larger number of instructions graduated.

System	Metric	2D-Grid		2DD-Grid		3D-Grid		3DD-Grid	
		-VI	QVI Change	-VI	QVI Change	-VI	QVI Change	-VI	QVI Change
SGI	DC Misses	80.3	10.0%	66.0	23.9%	22.3	14.7%	35.9	39.3%
	DSC Misses	12.1	40.2%	10.9	68.6%	4.6	28.3%	9.6	58.2%
	TLB Misses	7.2	-6.6%	5.6	-15.0%	1.7	-5.5%	0.87	-10.9%
	Instructions	2,782	-16.2%	2,480	1.9%	908	-8.4%	1,667	23.9%
IBM	DC Misses	48.6	-44.0%	36.7	-32.1%	7.0	8.0%	13.3	9.8%
	DSC Misses	7.3	9.2%	7.1	9.8%	3.3	4.2%	3.3	4.2%
	TLB Misses	10.3	-38.4%	6.8	-42.5%	1.3	1.7%	1.0	0.0%
	Instructions	2,257	-7.3%	1,775	9.9%	639	-1.1%	1,012	28.9%

Table 6. Misses in data cache (DC), secondary cache (DSC) and TLB, and graduated instructions (all measured in millions) for `-VI` and the percentile change for `QVI`.

5.5 Memory Space Requirements and Memory Footprints

With the exception of vertex clustering, the techniques described in this paper increase the memory space required to store the data for a given abstraction hierarchy because the data structure for each individual vertex is larger.

Figure 14 shows the growth in the memory footprint of the abstraction hierarchy for all combinations of graph instance and implementation used in our experiments. [6] Each bar is composed of four segments stacked on top of one another showing the abstraction hierarchy memory footprint of all implementations for a given graph instance. On average, queue embedding increased the memory footprint by 10.7% for `2D-Grid` graphs, 7.7% for `2DD-Grid` graphs, 8.7% for `3D-Grid` graphs, and 3.2% for `3DD-Grid` graphs. For image mapping, average increases were 46.2% for `2D-Grid` graphs, 61.4% for `2DD-Grid` graphs, 56.7% for `3D-Grid` graphs, and 83.4% for `3DD-Grid` graphs. When queue embedding and image mapping are combined, the memory footprint increase ends-up being the aggregate of the increases produced by the techniques when on their own. In all instances, the abstraction hierarchy memory footprint is always well below the main memory capacity of all of our test systems. Thus, we observed virtually no page faults.

[6] We refer to the amount of space required to store the abstraction hierarchy as its memory footprint.

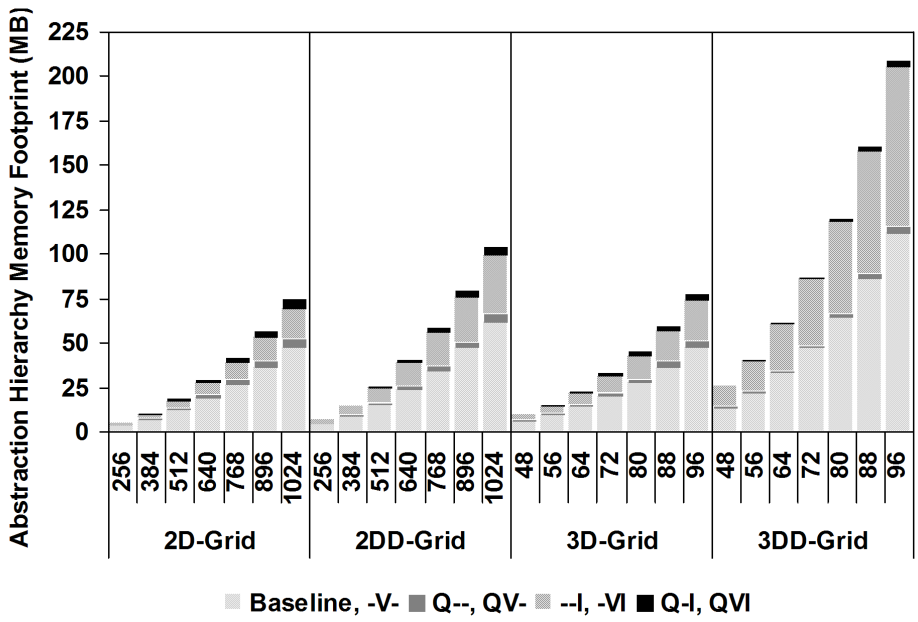
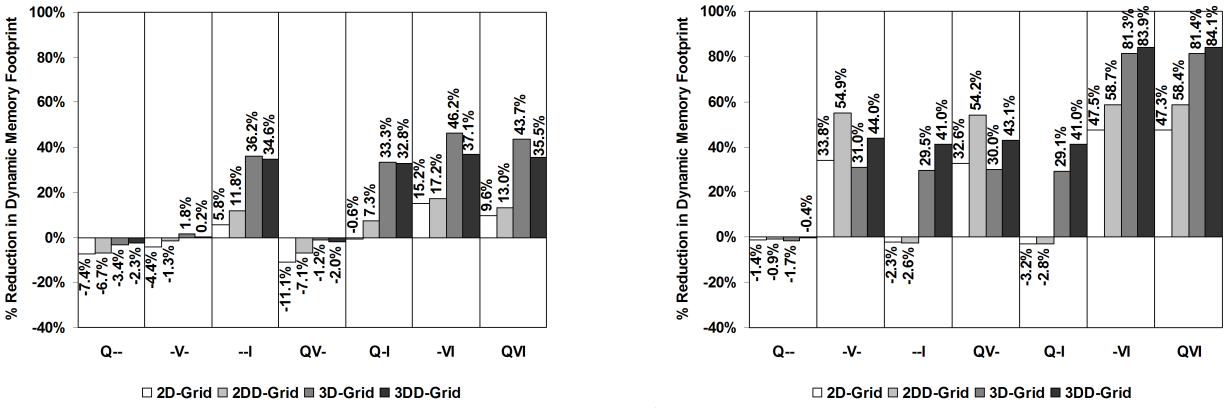


Fig. 14. Abstraction hierarchy memory footprints of all implementations for all instances of each graph type.



(a) Block size = 64-bytes.

(b) Block size = 4,096-bytes.

Fig. 15. The percentage reduction produced by each implementation over `Baseline` in the dynamic memory footprint. Results are presented for all graph types using two block sizes, capturing typical cache line and page capacities.

In contemporary architectures with virtual memory management, the memory space allocated for data storage is not necessarily a major concern. Instead, an algorithm designer seeking to improve performance should primarily be concerned with the amount of memory accessed during program execution, *i.e.,* the dynamic memory footprint. For example, relational database systems have long used data redundancy to improve query performance. Although the memory footprint of the database system may increase, the amount of memory accessed to execute a query decreases. With that in mind, we measured the dynamic memory footprint (DMF) as the average number of distinct memory blocks referenced during a single search.[7] The DMF measure encompasses stores and loads made to the memory region containing the abstraction hierarchy, and, when applicable, the queue vectors. In measuring the DMF of each implementation we used two block sizes, 64-bytes and 4,096-bytes, capturing typical cache line and page capacities. Assuming a negligible amount of data reuse from one search to the next, we can use DMF to gauge the amount of memory subsystem traffic generated by each implementation, both at the cache and at the page level.

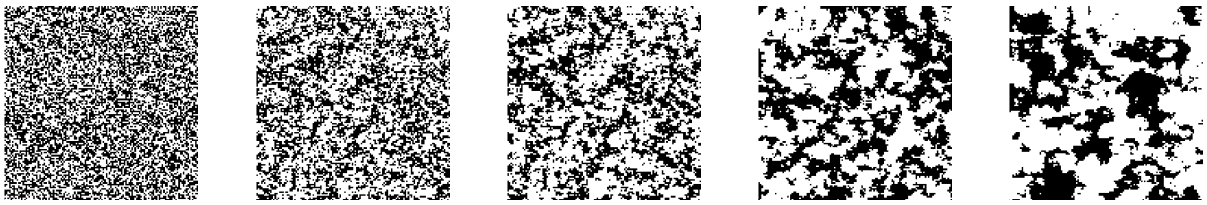
Figures 15(a) and 15(b) present a study of the effect of our techniques on reducing the DMF using block sizes of 64 and 4,096 bytes, respectively. The figures compare the DMF of each implementation to the DMF of `Baseline` for the largest instance of each graph type. Each bar represents the percentage reduction in the DMF. For example, -11.1% for the `2D-Grid` in Figure 15(a) means that `QV-` references 11.1% more distinct 64-byte memory blocks than `Baseline`.

At the cache line level, Figure 15(a), image mapping is the most effective technique in reducing the DMF. When used in isolation, techniques other than image mapping cause the DMF to increase. When image mapping is combined with the other techniques, the DMF improvement is greater than the sum of the DMF improvements of each technique. This result indicates that our techniques complement each other.

At the page level, Figure 15(b), the technique of vertex clustering produces the largest reductions in DMF. The technique of image mapping also yields DMF improvements, although only for three-dimensional grids. The use of queue embedding makes very little difference on the DMF at the page level, although it appears to be somewhat more effective than at the cache line level. Combining techniques appears to have a similar effect at the page level as it does at the cache line level.

Overall, our techniques yield significant decreases in the DMF of search. The largest improvements result from combining techniques and are generally better at the page level than at the cache line level. In addition, two-dimensional grids tend to benefit more than three-dimensional grids, especially at the cache line level.

5.6 Case Study

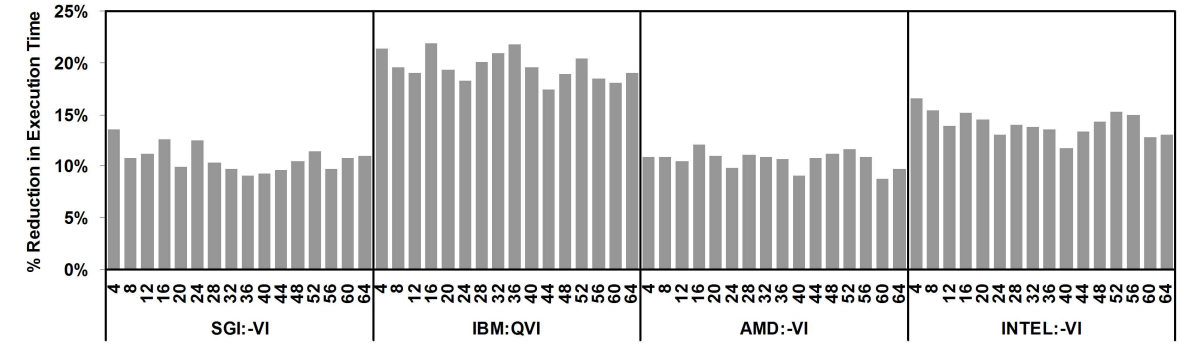


(a) 4. (b) 8. (c) 16. (d) 32. (e) 64.

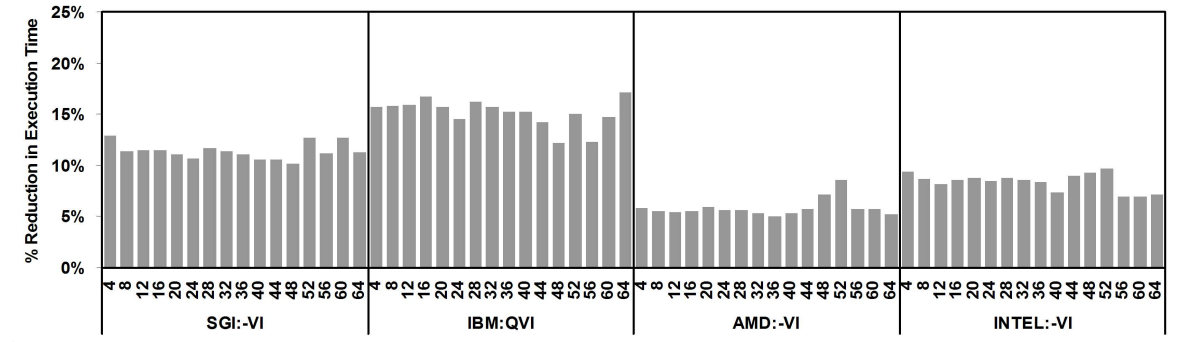
Fig. 16. Overhead snapshots of five of the sixteen terrain maps used in our non-empty graph performance study. Each snapshot is annotated with the grade of obstacle coarseness in the corresponding map.

So far our investigation has encompassed only empty graphs. What about non-empty graphs? Do our techniques improve search performance for non-empty graphs? To shed light on this issue we present a case study

[7] DMF is somewhat akin to the I/O measure commonly used in external memory algorithm analysis.



(a) 2D instances.



(b) 2DD instances.

Fig. 17. Summary of the performance gains attained by the best performing implementations on each system for both the `2D` and `2DD` graph instances of our fractal generated terrain maps. For each machine and implementation the bars correspond to the percentage reductions of `Baseline` execution times. Each bar corresponds to a given graph instance, with the number at the bottom the bar denoting the grade of obstacle coarseness in the corresponding map.

involving a class of non-empty graphs. In particular, we examine the performance of our implementations using fractal generated two-dimensional terrain maps.

Researchers in the field of *mobile robotics* often utilize fractal generated terrains to model both terrestrial and extraterrestrial terrains [32, 29]. We generated sixteen 1024×1024 fractal based terrain *maps*. Approximately 60% of the points in each map correspond to free space, with the remaining portion representing obstacles. The maps vary in terms of the coarseness of terrain obstacles, ranging from a grade of 4 to a grade of 64 in increments of four. [8] Figure 16 presents overhead snapshots of five of our maps with increasing grades of obstacle coarseness. Each map was converted into two explicit graph instances, a `2D` form and a `2DD` form. A `2D` instance is akin to a `2D-Grid` in that it features only perpendicular edges, while a `2DD` instance is similar to a `2DD-Grid` since it sports both perpendicular and diagonal edges. Because the maps are essentially two-dimensional grids, we used an input graph vertex order similar to the `DVO` of our `2D-Grid` and `2DD-Grid` graphs. We note however that unlike our empty graphs, our fractal generated graphs are not connected. Thus, a path between two randomly points on the graph is not always possible.

Our experimental results show our techniques yielding non-trivial performance gains for both the `2D` and `2DD` graph instances of the fractal generated terrain maps. In general, `QVI` was the best performing implementation on the `IBM` while `-VI` was superior to other implementations on the remaining systems. Figure 17 shows the percentage reduction in execution time achieved over `Baseline` by `QVI`, in the case of the `IBM`, and `-VI`, in the case of the remaining systems. Results are presented for all `2D` instances in Figure 17(a), and for all `2DD` instances in Figure 17(b). Both figures show performance gains on each system being relatively consistent accross all grades of obstacle coarseness. The average performance gain on each system varies between 10.6% to 19.6% for `2D` instances, and between 5.8% and 15.2% for `2DD` instances. How do these gains compare to those achieved for the empty `2D-Grid` and `2DD-Grid` graphs? As a basis of comparison we compare the results from our experiments involving the 768 two-dimensional grids, since the 768 grids feature approximately the same number of vertices as our maps. On average, the performance gains achieved for fractal instances were 11.9% larger than the ones attained for the 768 `2D-Grid` graph. In the case of `2DD` instances however, performance gains were typically 32.2% smaller than in the case of the 768 `2DD-Grid` graph. These experimental results with non-empty graphs indicate that our techniques improve path finding in irregular graphs.

6 Related Work

To our knowledge there is no previous work specifically addressing the locality of abstraction search algorithms such as CR. Nonetheless we find research addressing graph search locality in general. For instance, Edelkamp and Schrödl address the problem of thrashing of pages at the virtual memory level [13, 14]. They apply their localized A* to a route planning system. In a nutshell, their strategy involves improving reference locality by sorting vertices based on their relative geographic locations and altering the order in which states are expanded during search.

In the field of *external memory* algorithms we find various techniques for improving the I/O efficiency of external memory graph search [22, 23, 4, 7, 2, 31]. Typically, external memory algorithm techniques are akin to vertex clustering (grouping) and image mapping (data redundancy). For instance, blocking of data is used to minimize the number of page faults incurred during the traversal of paths in planar graphs. Variants of vertex grouping are also used to increase the performance of sparse matrix multiplication [25, 17].

Graph partitioning, needed for abstraction generation, is a well studied problem [12, 24, 20]. For example, consider the problem of partitioning a graph into k subgraphs such that each subgraph has roughly the same number of vertices and the number of inter-subgraph edges is minimized. The ability to partition graphs in this manner could enable the targetting of specific page and/or cache-line capacities during the abstraction generation process. Although, such an approach could lead to enhanced data reference locality, its effect on path-quality is unclear to us.

Also of interest are the ideas explored in the realm of *cache oblivious* algorithms [16, 3, 5]. In general, the aim of the cache oblivious paradigm is to improve the data access locality of algorithms independent of memory

[8] Coarseness grade is roughly equivalent to the maximum diameter of an obstacle.

hierarchy parameters. Although we have yet to come across a cache oblivious approach that enhances the locality of general sparse graph search, we find the concept interesting nonetheless. An interesting memory-oriented performance boosting approach has recently been proposed in the domain of implicit graph search. Korf finds that when searching an implicit graph with BFS, compared to checking for duplicates as you go, sorting the working queue before its expansion permits the elimination of duplicates, leading to better performance and possibly fewer data cache misses [21]. It remains to be seen if this approach can be effective in improving the performance of explicit graph BFS involving graphs such as our two and three dimensional grids.

The Artificial Intelligence community focuses on reducing the search space (for example, [28]), which can produce improvements of orders of magnitude. The gains obtained with the data structure transformation oriented techniques presented in this paper are orthogonal to the search space reduction, and the two techniques can be easily combined. They are also in line with performance improvements obtained through compiler transformations that improve data placement [6, 8]. Notice however that the automated techniques found in contemporary compilers are quite inept at improving data locality with respect to graph search in general. Even with the ongoing development of profile oriented compilation we expect this to continue to be the case because techniques such as our embedded queue and image mapping methods not only require a change in the manner data is layed out in memory but also require changes to the search algorithms themselves.

7 Conclusion

Research in the AI and computer game communities has produced algorithms to quickly find short paths in very large sparse graphs. However, the effects of temporal and spatial locality in the implementation of these algorithms has been mostly overlooked. This paper demonstrates that three simple data structure transformation techniques can consistently improve the performance of CR pathfinding for sparse graphs. In our experiments these techniques improved data reference locality resulting in performance improvements of up to 67% with consistent improvements above 15%. In addition, these techniques appear to be orthogonal to compiler optimizations and robust with respect to hardware architecture.

References

1. Freecraft real-time strategy gaming engine. http://www.freecraft.net.
2. Pankaj K. Agarwal, Lars Arge, T. M. Murali, Kasturi R. Varadarajan, and Jeffrey Scott Vitter. I/O-efficient algorithms for contour-line extraction and planar graph blocking. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete algorithms*, pages 117–126. Society for Industrial and Applied Mathematics, 1998.
3. Lars Arge, Michael A. Bender, Erik D. Demaine, Bryan Holland-Minkley, and J. Ian Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, pages 268–276, 2002.
4. Lars Arge and et al. On external memory MST, SSSP and multi-way planar graph separation (extended abstract).
5. Gerth Stlting Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. In *Proceedings of the Thirty-Fifth ACM Symposium on Theory of Computing*, pages 307–315, 2003.
6. B. Calder, K. Chandra, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, 1998.
7. Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Symposium on Discrete Algorithms*, pages 139–149, 1995.
8. Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.
9. Trishul M. Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *ACM SIGPLAN Conference on Programming language design and implementation (PLDI)*, pages 199–209, Berlin, Germany, June 2002.
10. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1991.
11. Personal correspondence with David C. Pottinger of Ensemble Studios.
12. Josep Daz, Jordi Petit, and Maria Serna. A survey of graph layout problems. *ACM Computing Surveys (CSUR)*, 34(3):313–356, 2002.

13. Stefan Edelkamp and Ulrich Meyer. Theory and practice of time-space trade-offs in memory limited search. *Lecture Notes in Computer Science*, 2174:169–??, 2001.
14. Stefan Edelkamp and Stefan Schrödl. Localizing A*. In *Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, pages 885–890, 2000.
15. Mark DeLoura (Editor). *Game Programming Gems Vol 1.* Charles River Media, 2000.
16. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, page 285. IEEE, 1999.
17. Norman E. Gibbs, Jr. William G. Poole, and Paul K. Stockmeyer. A comparison of several bandwidth and profile reduction algorithms. *ACM Transactions on Mathematical Software (TOMS)*, 2(4):322–330, 1976.
18. R. C. Holte, T. Mkadmi, R. M. Zimmer, and A. J. MacDonald. Speeding up problem solving by abstraction: A graph oriented approach. *Artificial Intelligence*, 85(1-2):321–361, 1996.
19. R.C. Holte, C. Drummond, M.B. Perez, R.M. Zimmer, and A.J. MacDonald. Searching with abstractions: A unifying framework and new high-performance algorithm. In *10th Canadian Conference on Artificial Intelligence (AI'94)*, pages 263–270. Morgan-Kaufman, 1994.
20. George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: application in VLSI domain. In *Proceedings of the Thirty-Fourth Annual Conference on Design Automation Conference*, pages 526–529, 1997.
21. Richard Korf. Delayed duplicate detection: Extended abstract. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1539–1541, August 2003.
22. Ulrich Meyer, Peter Sanders, and Jop F. Sibeyn, editors. *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002]*, volume 2625 of *Lecture Notes in Computer Science*. Springer, 2003.
23. Mark H. Nodine, Michael T. Goodrich, and Jeffrey Scott Vitter. Blocking for external graph searching. pages 222–232, 1993.
24. A. M. Patel and L. C. Cote. Partitioning for VLSI placement problems. In *Proceedings of the Eighteenth Design Automation Conference*, pages 411–418, 1981.
25. Ali Pinar and Michael T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 30. ACM Press, 1999.
26. David C. Pottinger. Terrain analysis in realtime strategy games. In *Game Developers Conference Proceedings*, 2000.
27. William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing 2nd Edition*. Cambridge University Press, 1992. pg. 284.
28. Stuart J. Russell. Efficient memory-bounded search methods. In *10th European Conference on Artificial Intelligence Proceedings (ECAI 92)*, pages 1–5, Vienna, Austria, 3–7 August 1992. Wiley.
29. Sanjiv Singh, Reid Simmons, Trey Smith, Anthony Stentz, Vandi Verma, Alex Yahja, and Kurt Schwehr. Recent progress in local and global traversability for planetary rovers. In *IEEE International Conference on Robotics and Automation*, April 2000.
30. A. Stoutchinin, J. N. Amaral, G. R. Gao, J. Dehnert, S. Jain, and A. Douillet. Speculative pointer prefetching of induction pointers. In Reinhard Wilhelm, editor, *Compiler Construction 2001 — European Joint Conferences on Theory and Practice of Software*, Lecture Notes in Computer Science, pages 289–303, Genova, Italy, April 2001. Springer-Verlag.
31. Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys (CSUR)*, 33(2):209–271, 2001.
32. Alex Yahja, Anthony Stentz, Snjiv Singh, and Barry L. Brumitt. Framed-quadtree path planning for mobile robots operating in sparse environments. In *IEEE International Conference on Robotics and Automation*, May 1998.