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

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 path finding algorithms find near-optimal paths in very large sparse graphs where traditional search techniques fail to generate paths in acceptable time. Refinement-based path finding uses an abstraction of the graph to prune the search space substantially. In this paper we demonstrate that further performance gains are obtained by improving the match between refinement-based algorithm implementations and the memory hierarchy found in modern computers. Three simple transformations in the data structures used to store the vertices of the graphs resulted in consistently positive performance improvements upwards of 50%. Our testing of these implementations in four machines and four compilers indicates that the performance improvements (1) are robust, and (2) are orthogonal to compiler optimizations and search space reduction.

1 Introduction

Path-finding finds applications in many industries such as computer games, freight transport, passenger traveling, 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 path-finding for movements on the game map [13]. In these games path-finding consumes up to 50% of total computation time [10, 23]. Using a modification to Dijkstra's algorithm, the shortest path between two vertices in a graph $G(V, E)$ can be computed in $O(E \log V)$ [9]. However for applications where $|V|$ is very large we want to visit only a fraction of the vertices in V to find approximations to the shortest path [17]. *Refinement-based search* (RBS) is often used to restrict the search space [16]. *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 if 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 vertices identified; (3) using this abstract path to find a path in the original graph.

This paper presents a performance evaluation study of three techniques that improve spatial and temporal locality of Classic Refinement: (1) data duplication; (2) data reordering; (3) merging of independent data structures into a common memory area. These techniques benefit from wide cache lines and from increased data reuse. They result in performance improvements between 7.5% and 34%. These results are robust to changes in compilers and processor architectures. We compared the number of cache misses and TLB misses in several versions of the algorithm, and established a strong correlation between gains in performance and reduction in TLB misses. Finally, is hand crafting of data structures necessary when we have optimizing compilers? We compiled our various implementations of Classic Refinement in SGI, IBM, AMD, and Intel machines using both vendor compilers and GCC at optimization levels -O0 and -O3. In all cases the hand-crafted data structures and algorithms resulted in non-trivial additional performance improvements.

Section 2 presents the Classic Refinement algorithm. Section 3 describes the baseline implementation and our three techniques. Section 4 presents experimental results and analysis. Section 5 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

Let $G_0(V_0, E_0)$ be the input graph to Classic Refinement. Let $G_1(V_1, E_1)$ be an *abstraction* of G_0. 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 a hierarchy of abstractions. 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.

To generate an abstraction hierarchy we use the "max-degree" STAR algorithm [16]. Given G_0 and a constant r, the STAR algorithm partitions G_0 into subgraphs whose maximum diameter is at most $2r$.

2.1 Refinement-based Search

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 restricts its search space to the vertices in the pre-image of v_p^{a+1}.*[1]

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}.*[2]

2.2 Classic Refinement

Figure 1 presents pseudocode for Classic Refinement. 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 Classic Refinement (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 $NULL$ (steps 1-5). LookupVertexImage$(g^0, n-1)$ returns the image of g^0 at abstraction level $n-1$. FindPath$(s^{n-1}, g^{n-1}, n-1)$ returns a path from s^{n-1} to g^{n-1} at abstraction level $n-1$.

If a path is found, CR iterates through each level of abstraction (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 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. When the recursion finishes, CR returns the path P_0.

[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 contain at least one vertex outside the pre-image of v_p^{a+1}, and therefore is not a constrained path.

[2] 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. 1. Classic Refinement Algorithm.

3 The Three Data Layout Techniques

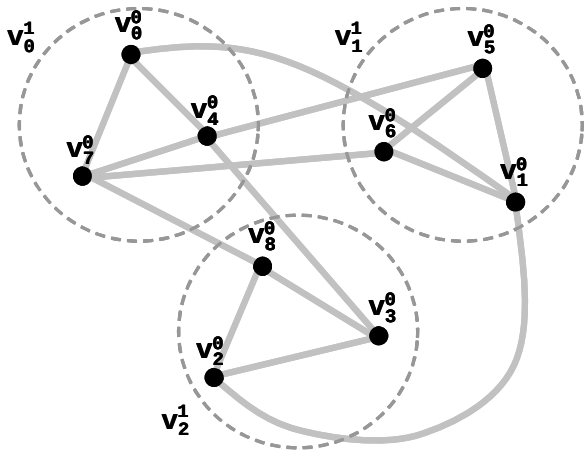


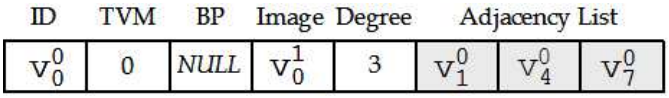
Fig. 2. Running Example.

Our baseline implementation of Classic Refinement is based on sound implementation techniques for sparse graph traversal algorithms based on adjacency lists representations. We use the graph in Figure 2 as an example. In the baseline the vertex v_0^0 of Figure 2 has the data structure shown in Figure 3(a). ID is a unique identification. The traversal visit marker (TVM) indicates if the vertex has been visited. BP is a back pointer. Image is a pointer to the vertex's image. Degree is the number of neighbors.

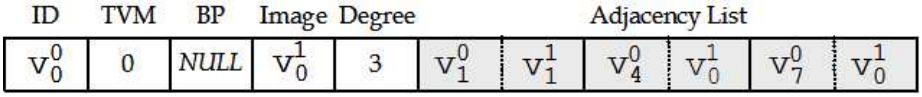
The use of a 32-bit field for the TVM allows us to not have to reinitialize it upon the start of each search. All TVMs are initialized to zero. A global search counter is maintained. Whenever a vertex v_a^i is visited during the z^{th} search, its TVM is set to z. Therefore any vertex that has a TVM smaller than z during search z, has not been visited yet.

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 [9]. However we found that the overhead of checking for wrap-around and overflow is high. We eliminate this bookkeeping by simply allocating enough memory to contain a pointer to each vertex in the graph. This practice is used in path finding engine implementations in video games [1, 13].

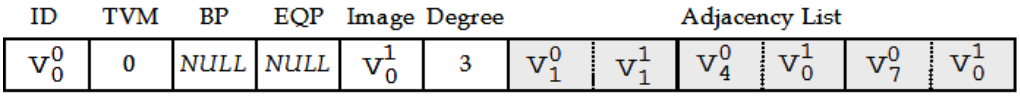
Vertex Clustering: Figure 4 shows a layout of the vertex data structures in memory for the graph of Figure 2. Each small box represents a 32-bit memory field. For convenience of drawing we present eight



(a) Baseline.



(b) Abstract map.



(c) Abstract map with embedded queue.

Fig. 3. Fields in the data structure of a vertex.

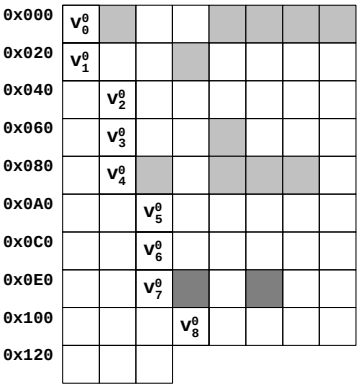


Fig. 4. Memory Layout without Vertex Clustering.

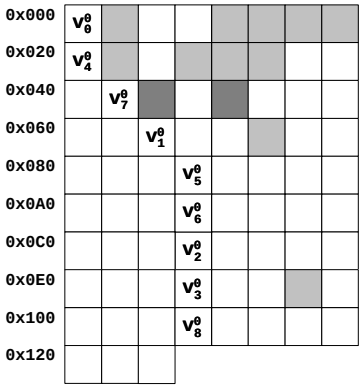


Fig. 5. Memory Layout with Vertex Clustering.

32-bit fields per line. We identify the 32-bit field where the data structure corresponding to each vertex starts. Consider a search, in Figure 2, for a constrained jump path from v_0^1 to v_2^1 starting at v_0^0 and ending at v_3^0. The shaded areas in Figure 4 show the memory locations that are accessed in this search. Besides the irregular memory access pattern shown in Figure 4, the baseline implementation also keeps a work queue in a separate region of memory. Accesses to this queue are interleaved with the accesses shown in Figure 4. Such accesses hurt spatial locality and are potential source of data cache conflict misses.

Our *vertex clustering* technique re-arranges the vertices, such that vertices that map to the same image are located in close proximity of each other in memory. Figure 5 shows the memory layout after of Figure 4after vertex clustering. Shaded areas are locations accessed for the same constrained jump path search. Notice how the memory accesses are much closer to each other in memory. We expect vertex clustering benefit abstractions generated with larger radius.

Image Mapping: Even after vertex clustering, the constrained jump path discussed above still has poor spatial locality. In order to search a path of vertices that map to v_0^1 we need to access the Image field of each vertex encountered during the search. As a result the search accesses memory locations that are far from the clustered vertices (see the shaded box at the bottom of Figure 5).

Our *image mapping* augments the vertex data structure as shown in Figure 3(b) to include the image field of each neighbor of the vertex. Thus when finding constrained jump paths or constrained paths the search does not access remote memory locations to determine the pre-image of a neighboring vertex.

Embedded Queue: The next source of poor memory reference pattern is the working queue of BFS that resides in a remote memory region. The interleaving of accesses between the vertex cluster region and the working queue region may cause *cache thrashing* — entries that will be used later are discarded because of memory conflicts — and reduces the benefits of the free prefetching due to large cache lines.

Our*embedded queue* technique stores 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 3(c). The EQP field contains a pointer to the last vertex that was added to the working queue.

3.1 The Embedded Queue Constrained Jump Path Algorithm

```
EmbeddedQueueConstrainedPath(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:                   BST_BP(v) ← w
9:                   return v
10:              if Image(v) ≠ Image(s)
11:                  continue
12:              if TVM(v) = TRUE
13:                  continue
14:              BST_BP(v) ← w
15:              EQP(v) ← w'
16:              w' ← v
17:              TVM(v) ← TRUE
18:          endfor
19:          w ← EQP(w)
20:      endwhile
21:      w ← w'
22:      w' ← NULL
23:  endwhile
```

Fig. 6. Embedded Queue Constrained Path Algorithm with Abstract Map

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, shown in Figure 6, 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 BFS, 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. 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).[3] Adjacency lists ensure that the accesses 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.

[3] The algorithm assumes that if the start vertex s is in abstraction level a, then G_{a+1} has an edge between the image of s and the destination image I.

Spatial locality is promoted because: (1) the comparison between the image of v and the image of the starting vertex s (step 10) accesses data within v (*image mapping*); and (2) accesses to EQP (steps 15 and 19) are also within v and w (*embedded queue*).

The direction in which the embedded queue is constructed and traversed 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.

4 Experimental Results

The results of our experiments can be summarized as follows:

- The combination of clustering vertices, embedding queues and mapping images produces consistent performance improvement upwards of 50%. This improvement is the result of improved page-level and cache line level locality.
- These techniques are robust to changes in the compiler, the processor architecture and memory hierarchy (see Figures 7-9 and Table 4). When used in isolation, the technique that produces the best result depends on the type of graph. Combining techniques often results in better performance improvements.

4.1 Experimental Framework

We studied the performance of our three techniques: embedded queues (`Q`), vertex clustering (`V`), and image mapping (`I`). We wrote eight versions of Classic Refinement: `Baseline`, `Q--`, `-V-`, `--I`, `QV-`, `Q-I`, `-VI`, and `QVI` (the three characters in the version denotes either the presence or the absence of each one of the features).

We selected three types of graph for our study:

2D-Plane A $h \times w$ two-dimensional plane. Except for borders, each vertex is connected by 4 edges. 2D-Planes represent 2D graphs such as street maps and RTS video-game game-worlds.

3D-Cube An $h \times w \times d$ three-dimensional cube. Except for borders, each vertex has degree 6. 3D-Cubes stand for graphs representing three-dimensional objects, such as buildings and bridges.

Airway-Road We fit the road network of the city of Pittsburgh into each vertex of an airline route graph. We vary the size of the road network from 512 to 3584 to obtain several graph sizes. These graphs are representative of graph used for trip and transportation planning.

The following is true for all experiments reported in this paper: (1) we report the wall clock time, measured in seconds with a C system function call, required to compute 10,000 paths between random pairs of vertices.[4] (2) The times reported include the construction of the actual path into a single linked-list through the traversal of the BP pointers of the vertices in the path.[5] (3) The abstraction generation uses a radius of 2, and recurses until it constructs an abstract graph with a single vertex. (4) The pre-processing time required to generate the abstraction is not included in the runtimes.[6] (5) Unless otherwise stated, we used a vendor compiler at optimization level -O3.

The machines and compilers used are listed in Table 1. For hardware counters, we used the Perfex library on the SGI machine, PMC on the AMD machine, and the PAPI libraries on the IBM and Intel machines. We compiled and ran the Baseline and the `QVI` versions of the program at two levels of optimization (-O0 and -O3) for all the compilers listed in Table 1. The combination of the data layout techniques discussed in the paper (`QVI`) consistently produced performance gains for all compilers, levels of optimization and machines.

[4] Because every graph is connected our experiments never include dead-end searches. To generate the random pairs we used the portable and deterministic random number generator described in[26].

[5] Building the actual path takes from less than 1% to 17% of the execution time.

[6] In practice, once an abstraction is generated, millions of path searches are performed.

Feature		SGI IP27	IBM p610	Intel P4	AMD 2000+
Processor		MIPS R12K	POWER3	Pentium 4	2000+XP
Data Cache L1	Size	32Kb	64KB	8 KB	64 KB
	Assoc.	2-Way	128-Way	4-Way	2-Way
	Line	32 Bytes	128 Bytes	64 Bytes	64 Bytes
Data Cache L2	Size	4 MB	8 MB	512 KB	256 KB
	Assoc.	2-Way	4-Way	8-Way	16-Way
	Line	128 Bytes	128 Bytes	64 Bytes	64 Bytes
Data TLB L1	Size	56 Entries	256 Entries	128 Entries	32 Entries
	Assoc.	Fully	2-Way	Fully	Fully
Data TLB L2		*none*	*none*	*none*	256 Ent./4-Way
VM Page Size		16Kb	4Kb	4KB	4KB
DRAM		1 GB	1GB	1 GB	1 GB
Clock Speed		350 MHz	450 MHz	2260 MHz	1667 MHz
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)

Table 1. Machines and compilers used in our experiments.

4.2 Results

The goal of our experiments are:

1. to determine if `QVI` is robust to changes in compilers, proc. architectures, and memory hierarchy — it is;
2. to study the performance of several types of graphs in various machines — `QVI` produces the largest performance improvements with Airway-Road graphs;
3. to identify the contribution of each technique to performance — it depends on the graph type;
4. to correlate the performance with cache misses and TLB misses — TLB misses are most significant.

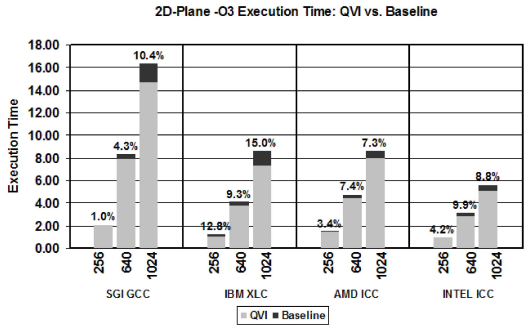


Fig. 7. 2D-Plane: Baseline and `QVI` execution time.

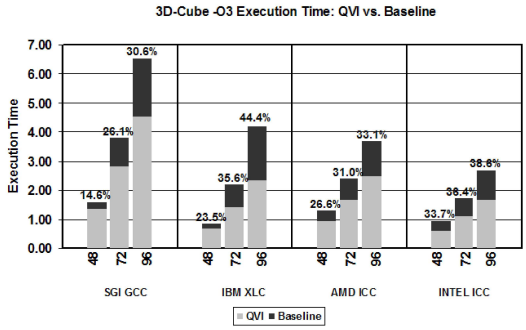


Fig. 8. 3D-Cube: Baseline and `QVI` execution time.

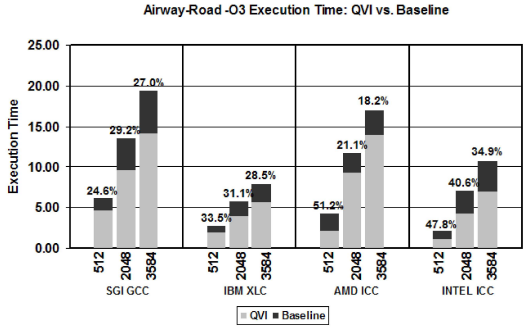


Fig. 9. Airway-Road: Baseline and `QVI` execution time.

Robust Techniques: The graphs in Figures 7 through 9 display the execution time for the baseline version and the `QVI` version of the code in all four machines for three problem sizes of each graph type. For each experiment the lighter part of the bar is the execution time for `QVI` and the darker part is the additional execution time required for the baseline. The percentage reduction in execution time is printed on the top of each bar. The number below each bar is the graph size. QVI speed improvements vary from 1.0% to 51.2%. Table 2 shows the minimum, maximum, and average execution time reduction (in percentage) for each graph type when eight graph sizes are taken into consideration. The 3D-Cube and Airway-Road graphs resulted in more performance improvement than the 2D-Plane graphs on all machines and for all compilers.

Figures 7 through 9 show that `QVI` is robust to changes in machine architectures. Is `QVI` also robust to compiler changes? To answer this question Table 4 shows the execution times, at optimization level -O3, for two different compilers in each machine. `QVI` consistently performs better than the baseline. Also, not surprisingly most vendor compilers outperform GCC. However, the difference is not great: GCC performance is very close to most vendor compilers for our programs.

Machine	1024 2D-Plane			96 3D-Cube			3584 Airway-Road		
	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
SGI	0.1	10.4	4.9	14.6	30.6	24.5	24.6	30.0	28.4
IBM	9.3	15.0	11.7	23.5	44.4	34.9	25.8	33.5	29.6
AMD	3.4	7.6	6.7	26.6	33.1	30.4	18.2	51.2	26.9
INTEL	4.2	10.2	8.9	31.6	38.6	35.7	34.7	51.8	41.3

Table 2. Percentage reduction in execution time for `QVI` over the Baseline for each graph type

Machine	1024 2D-Plane			96 3D-Cube			3584 Airway-Road		
	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
SGI	0.0	15.6	8.3	17.2	32.9	27.4	20.0	27.8	23.8
IBM	-21.1	8.3	0.5	15.3	42.0	30.1	21.1	28.0	25.3
AMD	4.8	8.4	7.3	28.1	34.1	31.5	18.5	50.7	27.9
INTEL	5.3	11.5	9.9	34.8	40.1	37.4	35.9	46.9	41.3

Table 3. Percentage reduction in execution time for `-VI` over the Baseline for each graph type

Implem.	SGI		IBM		Intel		AMD		**Avg.**
	MIPS	GCC	XLC	GCC	ICC	GCC	ICC	GCC	
1024 2D-Plane									
Base	17.07	16.38	8.64	9.29	8.64	8.79	5.57	5.51	
`QVI`	14.75	14.67	7.34	7.84	8.01	8.04	5.08	5.00	
Improv.	**13.6%**	**10.4%**	**15.0%**	**15.6%**	**7.3%**	**8.5%**	**8.8%**	**9.3%**	**11.1%**
96 3D-Cube									
Base	6.83	6.53	4.19	4.53	3.69	3.75	2.67	2.73	
`QVI`	4.71	4.53	2.33	2.45	2.47	2.48	1.64	1.62	
Improv.	**31.0%**	**30.6%**	**44.4%**	**45.9%**	**33.1%**	**33.9%**	**38.6%**	**40.7%**	**37.3%**
3584 Airway-Road									
Base	22.33	19.40	7.89	9.79	16.95	17.07	10.74	10.85	
`QVI`	14.72	14.17	5.64	6.58	13.86	13.96	6.99	7.21	
Improv.	**34.1%**	**27.0%**	**28.5%**	**32.8%**	**18.2%**	**18.2%**	**34.9%**	**33.5%**	**28.4%**

Table 4. Comparison of execution times (measured in seconds) for the baseline and `QVI` implementations of the algorithm using different compilers in each machine.

Contribution of Individual Techniques Which technique (`Q`, `V` or `I`) contributes the most to performance? Could a single technique in isolation produce similar benefits? The best performances are highlighted in Table 5. Machines with large L2 caches, such as the IBM p610 (8 MB) and the SGI IP27 (4 MB), benefit from combining $\widehat{V}$ with $\widehat{Q}$. In machines with modest L2 caches, such as the Intel P4 (512 KB) and the AMD 2000+ (256 KB), the placement of vertex information and the queue entries in neighboring memory locations appears to be detrimental to performance. Combining techniques yields better performance than applying each technique in isolation. Though `-VI` and `QVI` yield similar performance, we recommend `QVI` because it always improve performance over the baseline whereas `-VI` occasionally degrades performance (see Table 3).

Machine	Implement.	Graph					
		2D-Plane		3D-Cube		Airway-Road	
		Time	Improv	Time	Improv	Time	Improv
SGI	Baseline	16.39	0.0%	6.53	0.0%	19.40	0.0%
	Q--	16.36	0.1%	6.63	-1.5%	18.87	2.7%
	-V-	15.76	3.8%	6.25	4.3%	16.20	16.5%
	--I	16.82	-2.7%	5.58	14.5%	18.63	4.0%
	QV-	16.11	1.6%	6.34	2.9%	15.97	17.7%
	Q-I	17.04	-4.0%	5.88	10.0%	19.57	-0.9%
	-VI	**13.83**	**15.6%**	**4.38**	**32.9%**	14.33	26.1%
	QVI	14.67	10.4%	4.53	30.6%	**14.17**	**27.0%**
IBM	Baseline	8.64	0.0%	4.19	0.0%	7.89	0.0%
	Q--	8.49	1.7%	4.30	-2.6%	7.90	-0.1%
	-V-	8.45	2.2%	3.80	9.3%	6.45	18.3%
	--I	9.31	-7.8%	3.57	14.8%	8.17	-3.5%
	QV-	8.22	4.9%	3.88	7.4%	6.35	19.5%
	Q-I	8.96	-3.7%	3.48	16.9%	7.85	0.5%
	-VI	7.92	8.3%	2.43	42.0%	5.97	24.3%
	QVI	**7.34**	**15.0%**	**2.33**	**44.4%**	**5.64**	**28.5%**
AMD	Baseline	8.64	0.0%	3.69	0.0%	16.95	0.0%
	Q--	9.04	-4.6%	3.84	-4.1%	17.36	-2.4%
	-V-	8.76	-1.4%	3.44	6.8%	16.27	4.0%
	--I	9.26	-7.2%	3.12	15.4%	16.75	1.2%
	QV-	8.88	-2.8%	3.55	3.8%	16.26	4.1%
	Q-I	9.57	-10.8%	3.20	13.3%	16.82	0.8%
	-VI	**7.99**	**7.5%**	**2.43**	**34.1%**	**13.81**	**18.5%**
	QVI	8.01	7.3%	2.47	33.1%	13.86	18.2%
Intel	Baseline	5.57	0.0%	2.67	0.0%	10.74	0.0%
	Q--	5.86	-5.2%	2.78	-4.1%	11.11	-3.4%
	-V-	5.22	6.3%	2.40	10.1%	8.00	25.5%
	--I	6.01	-7.9%	2.25	15.7%	11.35	-5.7%
	QV-	5.49	1.4%	2.50	6.4%	8.27	23.0%
	Q-I	6.20	-11.3%	2.30	13.9%	11.53	-7.4%
	-VI	**5.03**	**9.7%**	**1.60**	**40.1%**	**6.88**	**35.9%**
	QVI	5.08	8.8%	1.64	38.6%	6.99	34.9%

Table 5. Individual effect `Q`, `V`, and `I` on the performance of shortest path search. Time measured in seconds. Improvement in percentage reduction of execution time. Bold numbers are the best results for 1024 2D-Plane, 96 3D-Cube, and 3584 Airway-Road each machine.

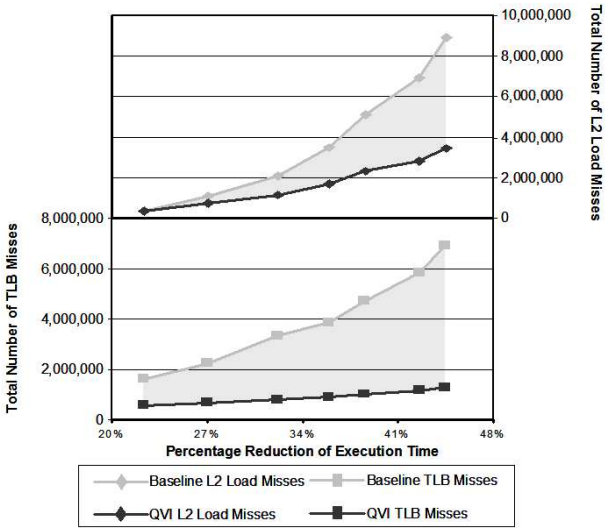


Fig. 10. TLB misses and L2 cache misses for the 3D-Cube graph on the IBM machine.

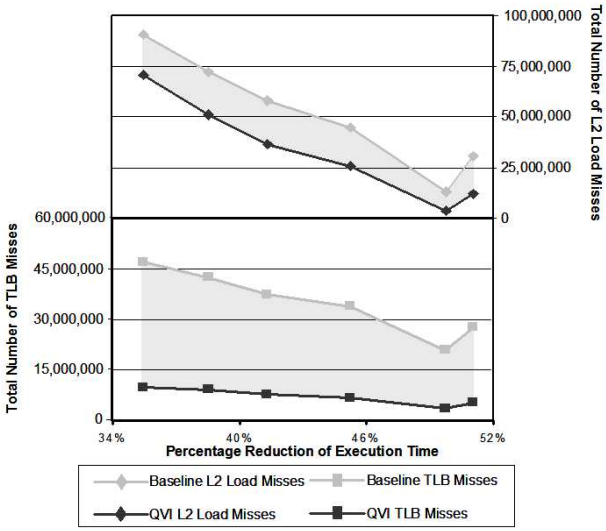


Fig. 11. TLB misses and L2 cache misses for the Airway-Road graph on the INTEL machine.

Cache and TLB Misses Using hardware counters we determined that all implementations issued and graduated similar number of instructions, and have a similar number of control hazards. Figures 10 and 11 show the variations in L2 and TLB misses for selected machine/graphs. To generate each graph

we ordered the results from several graph sizes according to the execution time reduction (shown in the horizontal axis). The graphs show the variations in the number of TLB misses and L2 cache misses between theBaseline and `QVI`. The graphs show a weak correlation between the performance improvement and the reduction in TLB misses. This behavior is weaker for some machines than for others (see [20] for more comprehensive results).

5 Related Work

We are not aware of previous work on improving the locality of abstraction search algorithms such as Classic Refinement. Edelkamp and Schrödl address the problem of thrashing of pages at the virtual memory level [11, 12]. They apply their localized A* to a route planning system. They improve reference locality by sorting vertices based on their relative geographic locations and by altering the order in which states are expanded during search. In the field of *external memory* algorithms we find v Various techniques, referred to *external memory* algorithms, improve the I/O efficiency of graph search [19, 21, 4, 7, 2, 25]. Typically these methods use vertex clustering (grouping) and image mapping (data redundance). For instance, blocking is used to minimize the number of page faults incurred during the traversal of paths in planar graphs. The idea of vertex clustering is applied to sparse matrix multiplication [22, 15].

Graph partitioning, needed for abstraction generation, is a well studied problem [18]. Improvements to partitioning of the source graph could yield improvements to page level locality. *Cache oblivious* algorithms aim at improving data access locality of algorithms independent of memory hierarchy parameters [14, 3, 5].

Russell's improvements to existing heuristic search algorithms is an example of the focus of the Artificial Intelligence community on search space reduction [24]. While reducing the search space may produce improvements of orders of magnitude, the gains obtained by the careful crafting of data structures presented in this paper are orthogonal to the search space reduction, and the two techniques can be easily combined. The improvements that we obtained with the redesign of data structures (10-50%) are 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 forsee 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.

6 Conclusion

Extensive research in the Artificial Intelligence and computer game communities has produced efficient approximation algorithms to quickly find short paths in very large sparse graphs. Researchers in these communities have been careful to ensure that the graphs and auxiliary data structures all fit in main memory to avoid swapping to disk. However, the effects of temporal and spatial locality in the implementation of these algorithms has been mostly overlooked. This paper demonstrates that simple changes to the data structures and algorithm implementations can yield consistent performance gains in path finding algorithms. Moreover our experimental results indicate that improvements are obtained in multiple architectures and with different compilers.

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. ACM Press, 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. ACM Press, 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. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, 1991.
10. Personal correspondence with David C. Pottinger of Ensemble Studios.
11. 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.
12. Stefan Edelkamp and Stefan Schrödl. Localizing A*. In *Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, pages 885–890, 2000.
13. Mark DeLoura (Editor). *Game Programming Gems Vol 1.* Charles River Media, 2000.
14. 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.
15. 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.
16. 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.
17. 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.
18. George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: application in vlsi domain. In *Proceedings of the 34th annual conference on Design automation conference*, pages 526–529. ACM Press, 1997.
19. 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.
20. R. Niewiadomski, J. N. Amaral, and R. C. Holte. Performance analysis of data layout techniques for the refinement-based path finding problem. Technical report, University of Alberta, 2003.
21. Mark H. Nodine, Michael T. Goodrich, and Jeffrey Scott Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.
22. 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.
23. David C. Pottinger. Terrain analysis in realtime strategy games. In *Game Developers Conference Proceedings*, 2000.
24. 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.
25. Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys (CSUR)*, 33(2):209–271, 2001.
26. William T. Vetterling William H. Press, Saul A. Teukolsky and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing 2nd Edition.* Cambridge University Press, 1992. pg. 284.