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 evaulation study of three simple data structure transformation oriented techniques aimed at improving the data reference locality of Classic Refinement. In our experiments these techniques improved data reference locality resulting in consistently positive performance improvements upwards of 51.2%. In addition, these techniques appear to be orthogonal to compiler optimizations and robust with respect to hardware architecture.

1 Introduction

Pathfinding has 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 [6]. In these games pathfinding consumes up to half of total computation time [1, 15]. Using a modification to Dijkstra's algorithm, the shortest path between two vertices in a graph $G(V, E)$ can be computed in $O(E \log V)$ [4]. 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 [10]. *Refinement-based search* (RBS) is often used to restrict the search space to generate quality paths [9]. *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 research is partially funded by grants from the Natural Sciences and Engineering Research Council of Canada and by the Alberta Ingenuity Fund.

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 upwards of 51.2% and that, through analysis of hardware event profiles, these results stem from improved page level and cache line level locality. By testing on four systems with various compilers and optimization settings we also demonstrate that our techniques are robust to changes in hardware architecture as well as the level of compiler optimization.

Section 2 presents the CR algorithm. Section 3 describes the baseline implementation and our three techniques. Section 4 presents experimental results and analysis. Section 5 discusses related work.

2 Abstraction and Search

Let $G_0(V_0, E_0)$ be the input graph to CR. Let $G_1(V_1, E_1)$ be an *abstraction* of G_0. G_0 and G_1 are both undirected, unweighted graphs. G_0 is partitioned into connected subgraphs. G_1 must have one vertex for each subgraph of G_0. If vertex v_i^0 in G_0 maps to vertex v_p^1 in G_1, v_p^1 is called the *image* of v_i^0 at abstraction level 1 (Note: v_p^1 should be read as "vertex p at abstraction level 1"). 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. 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 is in the pre-image of v_p^1 and v_j^0 is in the pre-image of v_q^1. This 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 [9]. Given G_0 and a constant r, the STAR algorithm partitions G_0 into subgraphs whose maximum diameter is at most $2r$.

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 $P[j]$ 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]

[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.

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]

```
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.

```
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. 2. Embedded Queue Constrained Path Algorithm with Abstract Map

Figure 1 presents pseudocode for the CR algorithm. Given a source graph G_0 and an abstraction hierarchy $A = \{G_0, G_1, \ldots, G_{n-1}\}$ CR finds 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 $NULL$ (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}. 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

[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.

in the pre-image of $P_{i+1}[j+1]$ visited by J. CR appends the constrained jump path J to P_i, updates b to be 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.

3 The Three Data Layout Techniques

Our baseline implementation of CR uses an adjacency list representation. We use the graph in Figure 3 as an example. In the baseline the vertex v_0^0 of Figure 3 has the data structure shown in Figure 4(a). ID is a unique identification field, the traversal visit marker (TVM) indicates if the vertex has been visited, BP is a back pointer, IMG is a pointer to the vertex's image, and DEG 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 and henceforth 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 due to performance and memory considerations [4]. 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.

Vertex Clustering: Figure 5 shows a layout of the vertex data structures in memory for the graph of Figure 3. 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 starts. Consider a search, in Figure 3, 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 5 show the memory locations that are accessed in this search. Besides the irregular memory access pattern shown in Figure 5, 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 5. 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 6 shows the memory layout after vertex clustering has been applied 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. Though not addressed in this paper we expect vertex clustering benefit abstractions generated with larger radius.

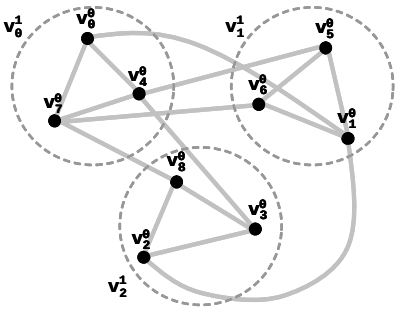
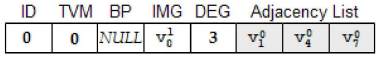
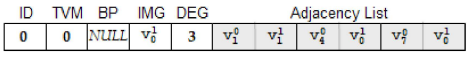


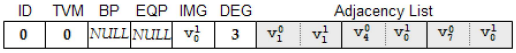
Fig. 3. Running Example.



(a) Baseline.



(b) Abstract map.



(c) Abstract map with embedded queue.

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

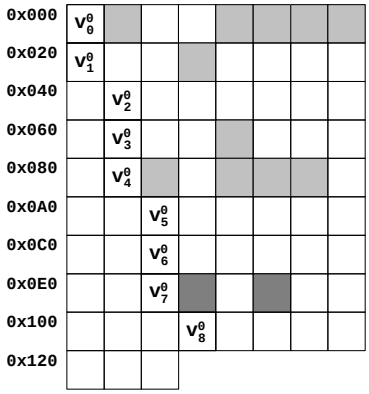


Fig. 5. Memory Layout without Vertex Clustering.

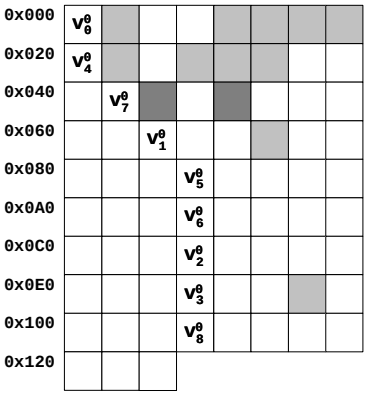


Fig. 6. Memory Layout with Vertex Clustering.

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 IMG 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 6). Our *image mapping* augments the vertex data structure as shown in Figure 4(b) to include the IMG 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 4(c). The EQP field contains a pointer to the last vertex that was added to the working queue. Using this technique entails a modification to the manner in which BFS is performed.

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 2, uses w to access the linked list formed by the EQP's 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. 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

We studied embedded queues (`Q`), vertex clustering (`V`), and image mapping (`I`) by writing eight versions of CR: `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 used three graph types: (1) `2D-Plane`, a $h \times w$ two-dimensional plane, (2) `3D-Cube`, a $h \times w \times d$ three-dimensional cube, and (3) `Airway-Road`, an airline route network where each vertex is an instance of a road network graph of the city of Pittsburgh. Multiple instances were generated for each graph type. For `2D-Plane` and `3D-Cube` graphs we varied dimensions while for `Airway-Road` graphs we varied the size of the road network graph.

[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.

The average vertex degree was 4 in `2D-Plane` graphs, 6 in `3D-Cube` graphs, and between 2.4 and 2.7 in `Airway-Road` graphs. Table 1 contains a summary of the systems utilized in our experiments as well as the compilers used to compile the implementions. For each system we used GCC and the processor vendor's compiler to compile each implementation with `-O0` and `-O3` (all results presented in this paper were obtained with `-O3` and a vendor compiler, except for `SGI` where we used GCC). All measurements encompassed the computation of 10,000 paths between random pairs of vertices.[4] Computing each path involved searching for it and reconstructing it into linked list form via BP pointer traversal.[5] To generate abstraction hierarchies we used the STAR method with a radius of 2. We recursed until an abstraction graph with a single vertex was constructed. [6]

We found that combining all three techniques results in the best overall performance improvements over `Baseline`. Although `-VI` ocassionally produced better performance gains than `QVI`, we prefer `QVI` because it always improved performance, whereas `-VI` did not. Figure 7 shows the percentage reduction in execution time produced by `QVI` over `Baseline` for all systems and graph instances. The figure shows consistently positive improvements on all systems ranging from 1.0% to 51.2%, highlighting `QVI` being robust to changes in hardware architecture. Is `QVI` also robust to compiler changes? In short the answer is *Yes*. Figure 8 presents the average percentage reduction in execution time produced by `QVI` over `Baseline` using all compilers with `-O0` and `-O3`. In all instances `QVI` produced non-trivial performance gains. In addition, because of the relative similarity of the performance gains obtained with `-O0` and `-O3`, it would seem that the manner in which `QVI` improves performance is orthogonal with respect to compiler driven optimizations.

Hardware event profiles indicate that in the case of `2D-Plane` graphs `QVI` improved data reference locality at the page level while degrading it at the cache line level. The same profiles show `QVI` improving locality at both levels for `3D-Cube` and `Airway-Road` graphs. As an example, we present Figures 9 and 10. The figures show the percentage reductions produced by `QVI` over `Baseline` in the total number of L1 and L2 cache misses and TLB misses on the `IBM` and `INTEL` systems. We also observed that queue embedding eliminated code otherwise required to maintain a detached working queue. As a result, `QVI` generally graduated about 10% less instructions than `Baseline`. Our results suggest that the bulk of `QVI` performance gains stem from data reference locality improvements at the page level and to a lesser degree at the cache line level.

When in isolation, the technique of vertex clustering proved to be the most effective, followed by image mapping and queue embedding. Our techniques appear to compliment each other since when combined they produced better performance gains than when on their own. Applying image mapping and/or queue embedding increases the memory footprint of the abstraction hierarchy (between

[4] Because every graph is connected our experiments never include dead-end searches.
[5] Reconstruction of paths took between 1% and 17% of the measured execution times.
[6] When generating G_{i+1} we sequentially iterated through all vertices in G_i using yet to be classified vertices as starting points for new subgraphs.

Table 1. Systems and compilers used in our experiments (All systems had 1GB of main memory and a UNIX based OS).

System	Processor	Compilers	Caches
SGI	R12K 350Mhz	MIPSPro (7.2.1) GCC (2.7.2)	L1 32KB L2 4MB
IBM	Power3 450Mhz	IBM XLC (6.0) GCC (2.9)	L1 64KB L2 8MB
AMD	XP 1667Mhz	Intel (6.0) GCC (2.96)	L1 64KB L2 256KB
INTEL	P4 2260Mhz	Intel (6.0) GCC (2.96)	L1 8KB L2 256KB

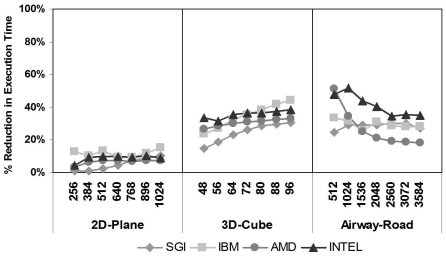


Fig. 7. Execution time reductions produced by `QVI` over `Baseline`.

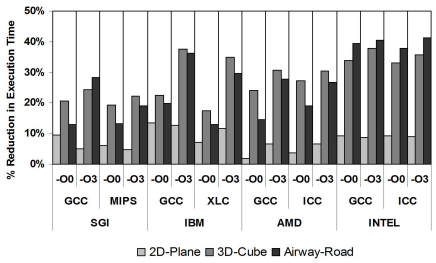


Fig. 8. Average execution time reductions produced by `QVI` over `Baseline` using various compilers with `-O0` and `-O3`.

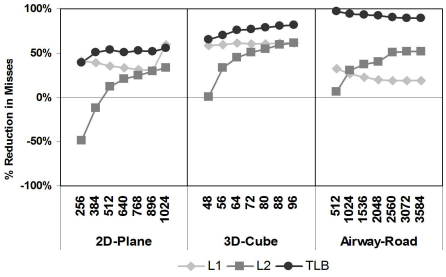


Fig. 9. L1, L2, and TLB miss-total reductions by `QVI` over `Baseline` on the `IBM` system.

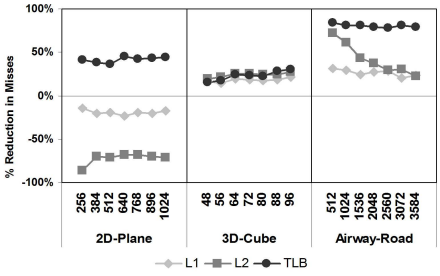


Fig. 10. L1, L2, and TLB miss-total reductions by `QVI` over `Baseline` on the `INTEL` system.

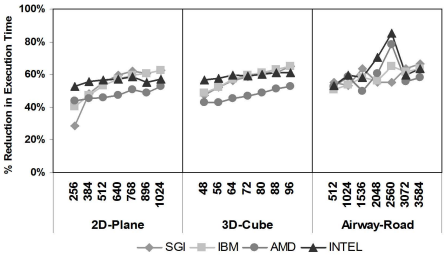


Fig. 11. Execution time reductions produced by `QVI` over `Baseline` with scrambled input graph vertex order.

8.7% and 65% compared to `Baseline`). However, the increase often translated into a decrease in the overall amount of heap memory touched during search, especially at page sized granularities. We note that our experiments generated virtually no page faults, which is indicative of memory traffic being exclusive to internal memory.

In the absence of vertex clustering our implementations place vertices in memory in G_0 in the order in which they appear in the input graph. Vertices in our input graphs are ordered in a manner that reflects the topology of the graph. For instance, vertices in our `2D-Plane` input graphs are ordered as left-to-right

rows stacked in a top-down fashion. When we scrambled our orderings we observed dramatic increases in performance gains. Figure 11 shows `QVI` producing improvements ranging from 28.4% to 85.2% under these circumstances. Scrambling reduced the likelihood of vertices in the same pre-image appearing in close proximity of each other in memory, thus making vertex clustering and image mapping more viable. Although such a scenario may seem artificial, consider a graph representing a national road system, where vertices correspond to cities and junctions. Here, it is possible for vertices to end-up in memory in an order based on an alphabetical sort of their labels (ie. city names), thereby having a similar effect on pre-image data proximity that vertex order scrambling had on our graphs.

5 Related Work

Edelkamp and Schrödl address the problem of thrashing of pages at the virtual memory level [7]. They apply their localized A* to improve the page level locality of a route planning system. In the field of *external memory* algorithms we find various techniques aimed at improving the I/O efficiency of graph search [12, 13]. Typically these methods use techniques akin to 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. Variants of vertex grouping are also used to increase the performance of sparse matrix multiplication [14]. Graph partitioning, needed for abstraction generation, is a well studied problem [5, 8, 11]. Better partitioning could yield improvements in page level locality.

The Artificial Intelligence community focuses on reducing the search space (for example, [16]), 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 [2, 3]. 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

Research in the Artificial Intelligence 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 oriented techniques can consistently improve the performance of CR pathfinding for sparse graphs. In our experiments these techniques improved data reference locality resulting in performance gains ranging from 1.0% to 51.2%. In addition, these techniques appear to be orthogonal to compiler optimizations and robust with respect to hardware architecture.

References

1. Personal correspondence with David C. Pottinger of Ensemble Studios.
2. B. Calder, K. Chandra, S. John, and T. Austin. Cache-conscious data placement. In *In Proceedings of the Eighth International Conference on Architectural Support for ProgrammingLanguages and Operating Systems*, pages 139–149, 1998.
3. 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.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*, chapter Chapter 24, Dijkstra's algorithm, pages 598–599. MIT Press, September 2001.
5. Josep Daz, Jordi Petit, and Maria Serna. A survey of graph layout problems. *ACM Computing Surveys (CSUR)*, 34(3):313–356, 2002.
6. Mark DeLoura. *Game Programming Gems Vol 1.* Charles River Media, 2000.
7. S. Edelkamp and S. Schrödl. Localizing A*. In *17th National Conference on Artificial Intelligence (AAAI-2000)*, pages 885–890, 2000.
8. 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.
9. 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.
10. 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 Conf. on Artificial Intelligence (AI'94)*, pages 263–270, 1994.
11. G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: application in vlsi domain. In *Proc. 34th annual conference on Design automation conference*, pages 526–529. ACM Press, 1997.
12. U. Meyer, P. Sanders, and J. F. Sibeyn, editors. *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar]*, volume 2625 of *Lecture Notes in Computer Science*. Springer, 2003.
13. M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.
14. 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.
15. David C. Pottinger. Terrain analysis in realtime strategy games. In *Game Developers Conference Proceedings*, 2000.
16. Stuart J. Russell. Efficient memory-bounded search methods. In *10th European Conference on Artificial Intelligence Proceedings (ECAI 92)*, pages 1–5, 3–7 August" 1992.