Memory-Efficient Abstractions for Pathfinding

Nathan R. Sturtevant
Department of Computing Science, University of Alberta,
Edmonton, Alberta, Canada T6G 2E8
nathanst@cs.ualberta.ca

Abstract

From an academic perspective there has been a lot of work on using state abstraction to speed path planning. But, this work often does not directly address the needs of the game development community, specifically for mechanisms that will fit the limited memory budget of most commercial games. In this paper we bring together several related pieces of work on using abstraction for pathfinding, showing how the ideas can be implemented using a minimal amount of memory. Our techniques use about 3% additional storage to compute complete paths up to 100 times faster than A*.

Introduction and Overview

Pathfinding is a key task in many domains, including video games. In games in particular, computers must compute a path between two points as efficiently as possible, as there many are many other demands on the CPU, such as physics, graphics, and even additional pathfinding tasks from other units (agents). We present a new method to build automated, minimal-memory state abstractions to speed pathfinding. With just 3% additional storage, we show large reductions in computational costs.

State abstractions for pathfinding have been explored by in a variety of different settings using a variety of methods for abstraction (Holte *et al.* 1996a; 1996b; Tozour 2003; Sturtevant & Buro 2005; Botea, Müller, & Schaeffer 2004). This paper describes in detail how state abstraction techniques can be optimized for use in games which have tight memory constraints. Specifically, this work is the product of a successful collaboration between BioWare Corp® and our university. According to Mark Brockington, Programming Fellow for BioWare Corp®, "Our collaboration with the University of Alberta on this project has been successful, and we are pleased with the performance and implementation of their pathfinding research within Dragon Age™."

Abstraction For Pathfinding

Automatic state abstractions are able to take a high-resolution map of the world and automatically transform it into a smaller, more abstract map, which can be used to speed pathfinding in the actual world environment.

The concept of using some sort of abstraction for pathfinding is a natural one, as humans often use abstractions to reason about the world. A long car trip does not being with minute-by-minute planning of the expected route. Instead, a general plan is composed which includes the start and goal cities and a high-level route between these cities. One may, for instance, decide whether to travel from Los Angeles to San Francisco by taking the faster but less scenic interstate or to take the scenic drive up the coast at the cost of additional time. Once this high-level decision has been made, it is up to the driver of the car to make low-level decisions such as which lane to drive in and how and when to pass other cars on the road.

Once a high-level route is known, pathfinding becomes a local operation. It is sufficient to plan the next few steps leading along the high-level abstract route as they guarantee that the goal will be reached. This is an ideal feature when running in environments which tightly limit computation costs.

The approach of first planning an abstract path before refining it has been considered in a variety of forms (Holte *et al.* 1996b; Sturtevant & Buro 2005; Botea, Müller, & Schaeffer 2004) and in a variety of domains (G.R. Jagadeesh 2002; Yang, Tenenberg, & Woods 1996). In this paper we address the issue of automatically building a state abstraction from an underlying map representation in a memory efficient manner. We then demonstrate the performance of our implementation in terms of memory used, pathfinding speed, and resulting path optimality.

Efficient Abstraction Implementation

We first describe how we can represent a map in a memory efficient manner, and then demonstrate how the representation can be used for pathfinding.

Computing Abstract Map

One common representation of a map is a simple grid, where a grid cell is the smallest unit of space that can be occupied by a single unit within the world. This is the underlying representation that BioWare Corp® decided to use for their maps in Dragon Age™. In one respect this representation is efficient, as there is a simple mapping between x/y coordinates and grid cells, avoiding the memory cost of using pointers. But for large maps, the total memory usage and

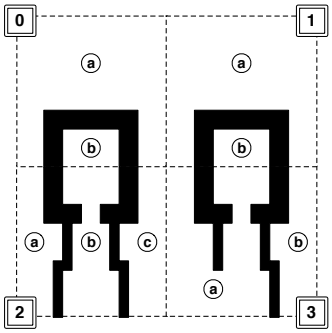

Figure 1: Computing sectors.

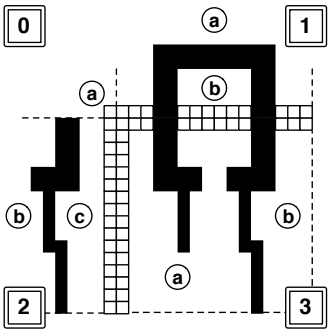

Figure 2: Computing edges.

cost of planning on such a grid can still be expensive. This is particularly true because we must store information for every grid cell, whether or not it can ever be occupied or traversed. The overhead of the high-resolution map means that our abstraction mechanism must use a minimum amount of memory. We propose an abstraction mechanism which maintains memory efficiency by using a large-scale grid. It is also effective in reducing planning costs. We call this a minimal-memory (MM) abstraction. We distinguish the abstraction representation from the techniques which use the abstraction for pathfinding. Although much of a MM abstraction can be built quickly, we expect that the abstraction will be built offline and stored with the map.

The first step in building a MM abstraction is to place a lower-resolution grid over the map. This grid implicitly breaks the world up into sectors; the sector of any unit in the world can quickly be calculated given its actual x/y coordinate. However, sectors are not guaranteed to be contiguous, so we need to do more than just divide the map into sectors.

Sectors are further divided into regions. There is one region for each connected component within a sector, that is, we require all nodes within a region to be reachable without leaving the sector. The nodes in a region are identified by performing a breadth-first search within a sector. We demonstrate sectors and regions in Figure 1, which shows a simple map. Black areas are considered blocked. In this figure there are 4 sectors, $0 \ldots 3$. Sector 2 has three regions (2.a, 2.b, 2.c), while sector 3 only has two regions (3.a, 3.b).

Each region within a sector is represented by a single point or node within the sector. Initially we place this point at the weighted center of the region, although we will later discuss how these points can be optimized.

Computing Edges

Once the sectors and regions have been established, we must compute edges between regions. During this process we assume that all cells in the original map are marked with their respective regions. Once the MM abstraction is built, we lift this assumption.

Edges are computed by iterating along the border of each sector, comparing the region for states on either side of the border. We add an edge to the abstraction for each unique pair of values between adjacent sectors and regions. This is illustrated in Figure 2, which is a closer view of sector 3 and its borders. We must iterate through the marked cells to add edges between sector 3 and other sectors. Region 3.a, for instance, has four edges: to region 2.c, region 0.a, and regions 1.a and 1.b.

In addition to the cardinal directions, we add diagonal edges to the abstraction. This increases the size of the abstraction, but if diagonal moves are possible in the world, having them in the abstraction results in higher quality paths.

The final abstract graph is shown in Figure 3. While the original map had approximately 900 states (four 16×16 sectors) and thousands of edges (not stored explicitly), the abstract map has 9 states, one for each region, and 10 edges.

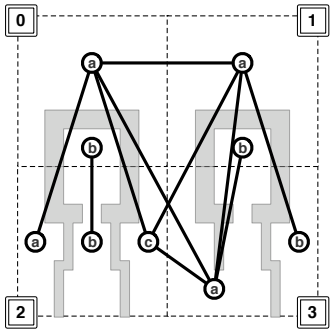

Figure 3: The final abstract graph.

Memory Allocation

Memory for the abstraction is allocated in two portions. The first portion is fixed-size, based on the size of the map being abstracted. This memory references the number of regions in each sector and the memory address in which the regions and edges are stored. The second piece of memory is variable sized and depends on the complexity of the map being abstracted. This memory contains the abstract regions and the edges between them. To store an abstract region we need the location of at least one node within the abstract region and the number of edges.

To make this presentation more concrete we make a few assumptions about sector sizes and density, and then compute the exact memory needed to store such an abstraction. First, we will assume that the maximum sector size is 16×16 or 256 grid squares. We will also assume that the number of regions inside a sector is limited. It is fairly simple to construct worst-case examples where there could be 32 or more

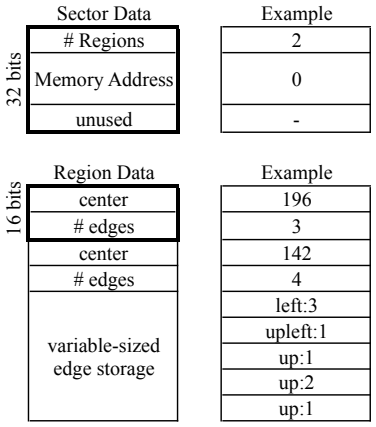

Figure 4: Memory layout for storing abstraction and example data.

regions inside a sector with as many edges. But these examples should not occur in real-world game maps.

The description of our data structures as well as an example are found in Figure 4. Sectors contain the number of regions (8 bits) and their memory address (16 bits). For memory efficiency, we pad them to 32 bits. Using a small secondary lookup table it is possible to squeeze this data into 16 bits, although we haven't done so for this implementation. We leave these bits free for other sector information that we might want to store (eg. occupancy information).

Each region is defined by a representative node (8 bits) and edge information (8 bits). The node is stored as the offset from the top-left corner of the sector, and so this limits the sector size to at most 16×16. This also limits this implementation of the abstraction to 256 edges per sector. Instead of storing the number of edges explicitly, we store the cumulative number of edges thus far in the abstraction. This optimizes the time needed to extract the edges for a region.

After the region center and edge information for each sector, we store edge data. The start sector and region of an edge are implicitly known and do not need to be stored. We just store the edge direction, which implicitly specifies the target sector. There are eight possible edge directions, which can be stored in 3 bits. The remaining 5 bits store the region which can be reached by the edge. This limits the number of regions in a sector to 32.

The right portion of Figure 4 shows the actual data layout for sector 3 from Figure 1. This sector is 16×16 and has 2 regions. Because we haven't stored other sector information here, the memory address in our table is 0. In the region data, the center of region 3.a is offset 196 cells (12 rows, 4 columns) from the top left of the sector, and this region has 3 edges. The center of region 3.b is offset 142 cells (8 rows, 14 columns) from the top left of the sector and has only one edge. The last edge for this region is offset four bytes from the first edge. The first edge for each region can be computed from the previous region.

In some maps there may be large regions of open space for which there will always be 8 edges, one in each direction. As a further optimization, we can mark these sectors with a special value which indicates that they are 'default' sectors, and then avoid storing these sectors explicitly to save memory. Note also that edges are directed, so we store them twice. It is possible to eliminate half of the edges, however this would increase the computation costs of finding edges as well.

Pathfinding Using Abstraction

A pathfinding problem is typically defined by start and goal locations. These locations are in the actual world, so the first step is to transform them into abstract space. In sectors with only one region this is trivial. But, in sectors with more than one region, the process is a bit more complicated. If extra memory is available, cells in the world can be annotated with their abstract region. Otherwise, a small search will find the current region center. The abstract location of a unit in the world can be cached so that this process does not have to be repeatedly applied.

The next step of any pathfinding process is to use A* (Hart, Nilsson, & Raphael 1968) to find a path through abstract space. We use the octile-distance between region centers both as our heuristic value, and as the g-cost of abstract edges. Given a complete abstract path, there are many different ways to use this information for computing paths in the actual world. We will describe a strategy for refining an abstract path as R[n, t, c, s].

The most simple way to use the abstract path is to follow the abstract region centers exactly. That is, plan from the start location to the first region location. Then, successively plan between each region along the abstract path, and finally plan between the final region center and the goal location. This can be generalized so that we refine longer portions in each step by skipping some segments of the path. The first parameter of R[n, t, c, s], n, is the number of abstract edges we refine in each step. So, the above approach is R[1, -, -, -]. If a parameter is unused, we will replace it with a dash '-'.

R[1, -, -, -] has several drawbacks. If we always plan paths between region centers along the abstract path, we often travel much further than required. We demonstrate this in Figure 5. If we start at the node labeled s and first proceed to the region centers along the abstract path before proceeding to the goal, the distance travelled will be much further than the optimal distance, which is a single step. The first and last steps of basic refinement can be optimized by not planning a path through the region centers, but just traveling directly to the goal. This same problem manifests itself, to a lesser extent, when traveling to each region center along a path. One partial solution is to implement a trimming policy. After reaching each region center, trim the returned path using some policy t. For instance, we might always remove the last 5 nodes from our path. Or, we can trim off the last 10% of the computed path. After trimming, plan from the end of the current path to the next region center. So, the actual path executed will not be forced to travel through all region centers. Refining 1 step at a time and then subsequently trimming the path by 10% would be R[1, 10%, -, -]. We use both of these techniques together whenever we trim.

The next option is to limit the refinement process to some corridor, c defined by the abstract path. That is, do not allow A* to expand nodes which fall outside the sectors on the

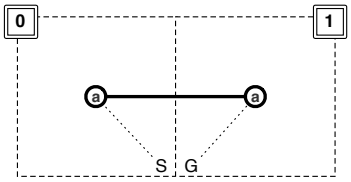

Figure 5: Paths through region centers are suboptimal.

abstract path. This can prevent the cost of any single step from getting too large, but it may result in strange paths, as units are forced to stay within bounds which aren't visible to the user. A better approach seems to be to optimize the region locations, which we will discuss in the next section.

Finally, after a path is complete, we can optionally apply a smoothing algorithm s to smooth the final path. This is particularly useful if our units walk on real-valued coordinates instead of on our pathfinding grid. So, in our experimental results and description of related work we will refer to refinement algorithms as R[n, t, c, s].

The refinement process can be performed in several independent operations, which means it is ideal for the tight time bounds of computer games. Given a time bound for path planning in each frame, successive portions of the abstract path can be refined until that time bound is exceeded. If one refinement step is not finished, the partial work can be discarded and re-computed on the next frame, as each refinement step is small. Only a minimal amount of state must be carried over between frames, just the current and abstract path, making this approach well-suited for time and memory constrained environments. Additionally, it is possible to begin executing the path before planning is finished.

Pathfinding Cost

In this section we analyze the total cost of pathfinding using the abstraction and the R[1, -, -, -] refinement policy. That is, what is the cost of computing an entire path when we refine one abstract step at a time without trimming, corridors or smoothing. Assume that we are trying to find a path length ℓ, with sector size z. The length of the abstract path, a, will be approximately ℓ/z. Assuming that it is simple to plan paths between region centers, the cost of refining one portion of the actual path will be z (the distance between sector centers), and the cost of refining the complete path (all abstract steps) will be $za = \ell$. This means that the cost of *refinement* using this policy is independent of the sector size. Thus, because larger sector sizes decrease the cost of finding an abstract path, the *total* work required to find a complete path will decrease as the sector sizes increase. We verify this prediction in the experimental results.

Abstraction Enhancements

We consider two possible enhancements for the abstraction.

Dynamic Maps

In many games, designers would like to see changes made to the map during play which may affect pathfinding. We present two possible approaches for handling this in the MM abstraction. First, if the changes are localized, the abstraction can just be dynamically rebuilt after these changes occur. This only has to occur locally in the sectors which were changed. A second approach, which can handle changes even more quickly, is to apply the changes to the map during preprocessing, and then build the abstraction a second time. This abstraction can be compared to the original abstraction, and any sectors which do not change can be discarded. Then, the region data for the sectors that do change can be appended to the end of the region data for the original abstraction. When the changes occur in-game, we only need to change the memory address of the region data for these regions, and the abstraction will effectively be updated.

Optimizing Region Centers

One potential optimization is to choose appropriate locations for region centers to minimize the computational cost of pathfinding. This computation can be performed differently depending on which refinement approach we are using. For simplicity, we assume we are using R[1, -, -, -]. Note that while we want to minimize the cost of traveling between region centers, we actually want to minimize the maximum cost, that is the worst-case performance.

We first place region centers as close to the center of the region as possible. Then, for each possible location where the region center can be placed, we compute the maximum number of nodes expanded by A* to reach each of the neighboring region centers. The region center of the current region is placed in the location which minimizes this cost.

We demonstrate the usefulness of this approach in the experimental results. In fact, any in-game parameters can be used to optimize the region centers in this way. For instance, we may also want to optimize the distance from walls, etc.

Related Work

The approach we describe in this paper can be seen as a combination of HPA* (Botea, Müller, & Schaeffer 2004) and PRA* (Sturtevant & Buro 2005). We make tradeoffs between these two approaches to decrease the memory used for abstraction. These algorithms both use A* at an abstract level to find an initial, abstract path, and then refine it, however, they use multiple levels of refinement and different types of abstraction.

HPA* uses a sector-based abstraction similar to our MM abstraction, however it has multiple entrance points in each sector, and it computes and uses optimal paths between these points. It then uses a simple smoothing mechanism once the paths are complete, so HPA* can be described as R[1, -, -, *simple*]. When HPA* caches optimal paths within the world it will use much more memory than the MM abstraction. The HPA* abstraction also stores many more points in the map, and so must use more memory for storage.

PRA* uses a more fine-grained abstraction based on merging groups of connected cells, or cliques, in the original map. PRA* avoids smoothing the final path by doing multiple refinements using a small refinement window around the abstract path, p. The length of refinement, k is a parameter for PRA*, so PRA* can be described as repeated applications of R[k, -, p^+, -]. The clique abstraction which PRA*

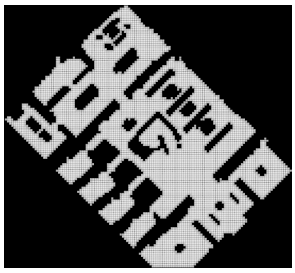

Figure 6: An example of a map used for our experiments.

uses explicitly stores parent and child information and is not memory-efficient at all.

An overview of other pathfinding methods can be found in, eg (Botea, Müller, & Schaeffer 2004). An longer discussion of many types of abstraction and their use for search and robotics can be found in (Fernandez & Gonzalez 2001).

Experimental Results

In our experiments we used a set of 120 maps taken from a popular role-playing game. We scaled these maps to 512×512 grid cells and then selected 93,235 total paths over these maps ranging in length from 1 to 512. An example map is shown in Figure 6. When experiments depend on path length, the problems are divided into 128 buckets, each bucket size 4.

Memory

We first look at the amount of memory needed to store abstract maps as the sector size increases. Note that if we use 8 bits of storage for each grid cell, a 512×512 map will take 256k of memory. We present the average memory used for the abstraction in Figure 7. With 16×16 sectors, the abstraction uses 6-7k, less than 3% of the cost of the full map. The top line is the amount of memory used when the abstraction stores all nodes and edges in the map. The bottom line is the amount of memory used when we mark the default sectors and do not store them. Because the maps in our experiments are scaled from smaller maps, this may be an over-estimate of the amount of the memory savings, but for larger sector sizes the gain from compression is minimal.

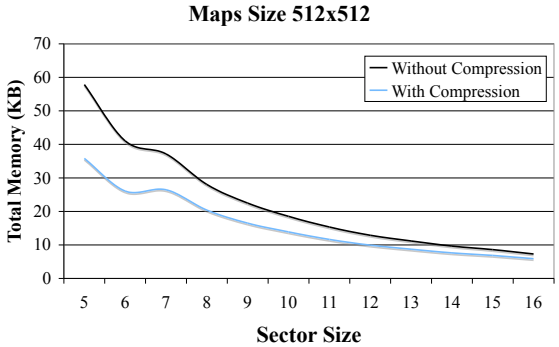

Figure 7: Memory used to store the abstraction.

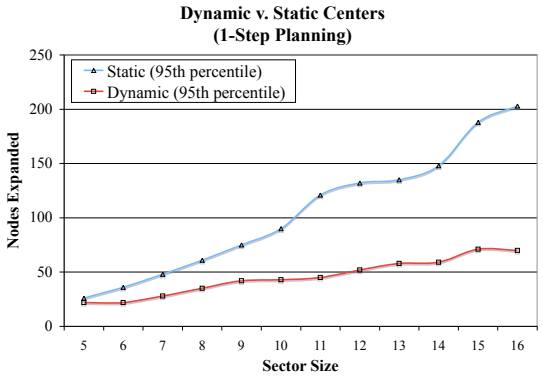

Figure 8: Dynamic or static region centers.

Region Center Optimization

Next we analyze the gains from optimizing regions centers in the abstraction. We run the same experiment on the MM abstraction with and without optimized region centers and compare the work that is done. For this experiment, we compare the work done on the largest bucket of problems in our data set (length 508-512) by R[1, 10%, -, -] as the sector size increases. We measured the maximum number of nodes expanded for any single refinement operation, and then graph the 95*th* percentile over all paths in Figure 8. This graph shows that dynamic centers can reduce the planning cost of an individual step by a factor of two or more.

Total Work

Third, we look at the total work performed when pathfinding using abstraction. We again use R[1, 10%, -, -] to refine abstract paths. In Figure 9 we show the total work in nodes expanded, needed to compute an entire path, including the abstract path and all portions of the actual path. We graph the 95*th* percentile of the number of nodes expanded in this figure. As expected in our theoretical analysis, increasing the sector size decreases the total number of nodes expanded, although there are diminishing returns. In fact, if we plot the same curves without optimizing region centers, we do the least work at sector size 14, and slightly more work with larger sector sizes. With larger sector sizes it is no longer easy to travel from one region center to the next, unless the region centers are optimized.

In Figure 10 we compare the total nodes expanded when planning an entire path using the MM abstraction and R[1, 10%, -, -] to the nodes expanded by A*. Note that we are plotting on a log-axis. The maximum number of nodes expanded using the MM abstraction even on the longest paths is just over 1,000. But, A* averages over 10,000 nodes for long paths and may expand nearly 100,000 nodes in the worst case. The minimum number of nodes expanded by each algorithms is similar, so we only plot one curve, although A* expands up to 100 fewer nodes.

Our implementation of A*, which is entirely generic and not optimized for any domain specific features, takes an average of 12μs to expand a node (83,000 nodes per second) on a 1.5Ghz PPC with 1GB of memory.

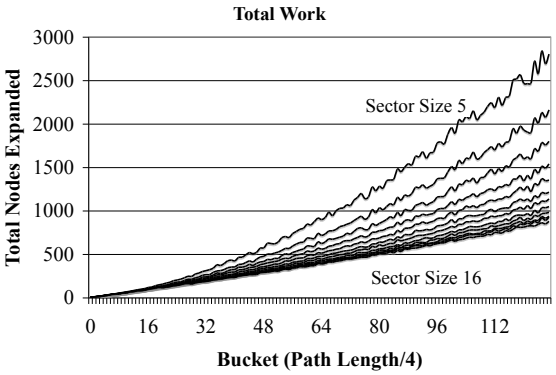

Figure 9: Total work required to compute paths.

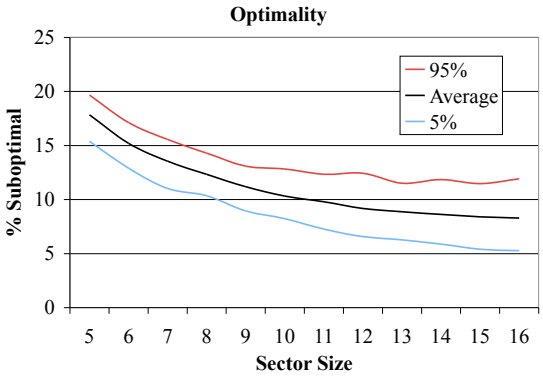

Figure 11: Suboptimality over sector sizes.

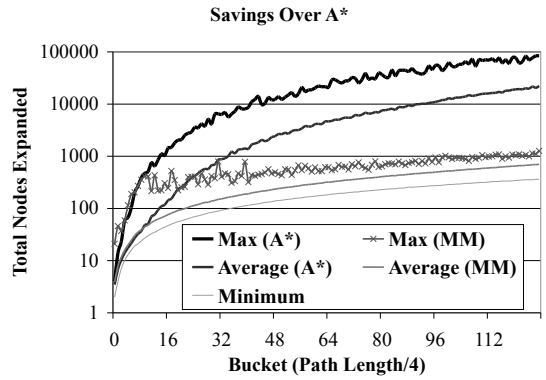

Figure 10: A* nodes expanded v R[1, 10%, -, -].

Optimality

The speed that comes with using an abstraction technique comes at the cost of optimality. We show the suboptimality introduced by the MM abstraction and R[1, 10%, -, -] refinement in Figure 11. These are the results only for the largest bucket, with paths of length 508-512. This illustrates that using larger sectors results in better optimality. For a sector size of 16, 90% of paths are between 5% and 12% suboptimal. All paths are between 4% and 18% suboptimal.

Suboptimality can be decreased in several ways. For instance, using R[2, 15%, -, -] refinement with sectors size 16×16, 90% of the paths will be just 2-6% longer than optimal, although this will increase the total pathfinding cost. Note that these results are on the longest paths from our experiments. Paths which only span one or two sectors will just use A* and be optimal. Paths that span a few sectors can have a higher percentage of suboptimality because they are so short. In practice, smoothing is applied as a post-processing step to further reduce suboptimality and to enhance the visual quality of the paths.

Conclusions

In this paper we have shown how known abstraction techniques can be adapted to domains with tight memory bounds. With roughly 3% more memory, pathfinding and refinement using the MM abstraction is up to 100 times faster than using A*. The MM abstraction is similar to other methods used in computer games, such as navigation meshes (Tozour 2002). This can be seen as a specialized type of navigation mesh, optimized for grid-based maps.

The MM abstraction has been implemented in BioWare's® upcoming title Dragon Age™ (due out in Winter 2007/2008), and has thus been proven to meet the needs of the games industry.

References

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Near optimal hierarchical path-finding. *J. of Game Develop.* 1(1):7–28.

Fernandez, A., and Gonzalez, J. 2001. *Multi-Hierarchical Representation of Large-Scale Space*. Kluwer.

G.R. Jagadeesh, T. Srikanthan, K. Q. 2002. Heuristic techniques for accelerating hierarchical routing on road networks. *IEEE Transactions on Intelligent Transportation Systems* 3:301–309.

Hart, P.; Nilsson, N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. on Systems Science and Cybern.* 4:100–107.

Holte, R.; Perez, M.; Zimmer, R.; and MacDonald, A. 1996a. Hierarchical A*: Searching abstraction hierarchies efficiently. In *AAAI/IAAI Vol. 1*, 530–535.

Holte, R. C.; Mkadmi, T.; Zimmer, R. M.; and MacDonald, A. J. 1996b. Speeding up problem solving by abstraction: A graph oriented approach. *Artificial Intelligence* 85(1-2):321–361.

Sturtevant, N., and Buro, M. 2005. Partial pathfinding using map abstraction and refinement. In *Proceedings of AAAI*, 47–52.

Tozour, P. 2002. Building a near-optimal navigation mesh. In *Steve Rabin, editor, AI Game Programming Wisdom*, 171–185. Charles River Media, Inc.

Tozour, P. 2003. Search space representations. In *AI Game Programming Wisdom 2*, 85–102. Charles River Media.

Yang, Q.; Tenenberg, J.; and Woods, S. 1996. On the implementation and evaluation of ABTweak. *Computational Intelligence Journal* 12(2):295–318.