DHPA* and SHPA*: Efficient Hierarchical Pathfinding in Dynamic and Static Game Worlds

Alex Kring, Alex J. Champandard, and Nick Samarin

Independent Researchers
awkring@gmail.com, alexjc@aigamedev.com, nicks@aigamedev.com

Abstract

In 2004, Botea et al. published the HPA* algorithm (Hierarchical Pathfinding A*), which is the first detailed study of hierarchical pathfinding in games. However, HPA* can be optimized. In this paper, we introduce the DHPA* and SHPA* hierarchical pathfinding algorithms, along with a metric for comparing the dynamic performance of pathfinding algorithms in games. We show that DHPA* searches up to an order of magnitude faster than HPA*, but consumes significantly more memory and produces slightly less optimal paths. The SHPA* algorithm searches up to five times faster than HPA* and consumes less memory, but it also produces slightly less optimal paths, and is only fit for static environments. When comparing the algorithms in dynamic environments, our results show that DHPA* often outperforms HPA*. Unlike many other hierarchical pathfinding algorithms, both solutions presented in this paper maintain a read-only terrain representation during search, which simplifies programming and debugging, and improves multithreaded performance.

1. Introduction

Modern video games require efficient pathfinding to support large numbers of agents moving through expansive and increasingly dynamic environments. Recent games *Warhammer 40000: Dawn of War 2,* and *Splinter Cell: Conviction,* both contain large worlds where the obstacles dynamically change the terrain representation (Jurney 2010; Walsh 2010). Hierarchical terrain representations improve pathfinding performance by bounding path searches to a smaller space, and restricting terrain changes to a subset of the map.

Many hierarchical pathfinding algorithms applicable to video games have been studied within academia (Holte et al. 1996; Botea et al. 2004; Sturtevant and Buro 2005; Demyen and Buro 2006; Bulitko and Björnsson 2009). Arguably, the most appropriate hierarchical pathfinding solution for video games is HPA*. The HPA* algorithm is simple to understand, easy to implement, and efficient, making it a prime candidate for video games.

Like many other hierarchical pathfinding algorithms, HPA* consists of a build algorithm and a search algorithm. The build algorithm defines the hierarchy through a series of graphs, where each graph abstracts a higher resolution graph. While HPA* works on multiple levels of abstraction, we primarily discuss it on a single level of abstraction, since multiple levels are more difficult to implement and show diminishing performance gains (Botea et al. 2004). After the hierarchy is prepared, the search algorithm finds a path at the highest level, and refines it into a series of segment paths along the lowest level. Botea et al. (Botea, Müller, and Schaeffer 2004) present HPA* for use on a grid, though it can be adapted for other terrain structures such as navigation meshes.

The HPA* build algorithm performs two offline tasks. First, it partitions the terrain into a collection of mutually disjoint clusters, where each cluster covers a constant number of low-level graph nodes, and a smaller number of abstract graph nodes. Second, the build algorithm creates a cache of edge weights. The cache stores a weight between each pair of connected abstract nodes, where the weight corresponds to the optimal low-level path length between those two nodes. The weights are then used to determine the movement cost in the abstract search.

At runtime, the build algorithm is responsible for updating the graphs when the terrain changes. In such cases, the build algorithm runs locally on the damaged clusters, and reconstructs the abstract nodes and portions of the cache corresponding to the damaged clusters.

HPA* is well fit for video games, but it is very generic. In practice, it can be both optimized significantly for static worlds where the graph hierarchy can be better prepared, and in typical dynamic worlds it can also be improved to handle frequent terrain changes. This paper first introduces the SHPA* and DHPA* algorithms for efficient hierarchical pathfinding in static and dynamic terrain representations, and then introduces a metric for comparing the dynamic performance of pathfinding algorithms in games. In section 2, we describe the SHPA* and DHPA* algorithms in detail. In section 3, we present our dynamic performance metric. In section 4, we provide an empirical analysis comparing SHPA*, DHPA*, and HPA*. And finally, in section 5, we summarize our conclusions, and present possible future research directions.

2. Algorithms

DHPA* and SHPA* both consist of a build algorithm and an improved search algorithm. Broadly speaking, DHPA* reduces the run-time cost of HPA* by spending more time and memory in the build algorithm, and less time in the search algorithm. In contrast, SHPA* improves performance and reduces the memory requirements of HPA* on static terrain. Both algorithms avoid changing the terrain representation during search, in order to simplify programming and improve multithreaded performance.

DHPA* Build Algorithm

The DHPA* build algorithm creates clusters in the same manner as HPA*, but caches additional information to improve the speed of both the abstract search and the low-level search. We run Dijkstra once for each abstract node within a cluster, creating a separate cache for each abstract node (Figure 1). The cache contains an entry for every low-level node within the cluster, representing information about the optimal path from the low-level node to the abstract node, found by Dijkstra. A single cache entry consists of the optimal path length between the pair of nodes, and a path index pointing to a neighbor node within the same cluster. Effectively, the cache represents the shortest path tree for each abstract node. The abstract search utilizes the cached path length, and the low-level search utilizes the cached path index.

When the terrain dynamically changes within a cluster, we run the build algorithm within that cluster and recompute the corresponding portion of the cache, the intra-edges[1] within that cluster, and the inter-edges connected to that cluster. We further optimize this re-computation by only recreating a cluster's intra-edges and inter-edges when necessary. It is only necessary to recreate these edges if any of the border cells dynamically change. Otherwise, the abstract graph remains the same, and only the cache is re-computed.

The DHPA* cache requires more memory than the HPA* cache. The cache stores $O(bn^2c^2)$ entries, for b is the total number of abstract nodes per cluster, $n \times n$ is the size of each cluster, and $c \times c$ is the total number of clusters. In comparison, the HPA* cache is the same size as the number of intra-edges. This size, in the worst case, is denoted by the following equation (Botea et. al. 2004).

$$n(c-2)^2(2n-1) + 2(n-1) + 3(c-2)(1.5n-1)$$

[1] The intra-edges connect the abstract nodes within each cluster. The inter-edges connect two separate clusters.

Which can be expressed as $O(n^2c^2)$. Also, DHPA* creates the same number of abstract nodes as HPA*, because both build algorithms use the same process for creating clusters. In the worst case, there are $2n$ abstract nodes per cluster (Botea et. al. 2004). Thus, the HPA* cache stores $O(n^2c^2)$ entries, and the DHPA* cache stores $O(n^3c^2)$ entries, and therefore DHPA* stores a factor of n more cache entries in the worst case. Next, we describe how we use the cached path lengths and indices to improve search speed.

DHPA* Search Algorithm

The DHPA* search algorithm uses the cache that was created in the build algorithm to generate the abstract path, and refine it into a low-level path. We increase the speed of the abstract search by eliminating the time consuming "SG effort" that is present in HPA*. The SG effort is the performance-wise effort of inserting the start and goal nodes into the abstract graph before each search (Botea et al. 2004). This involves searching for the edge weights that will connect the start and goal nodes into the abstract graph, where each edge weight represents an optimal path length found by an A* search. For example, connecting the start node into a cluster containing four other abstract nodes will require four A* searches.

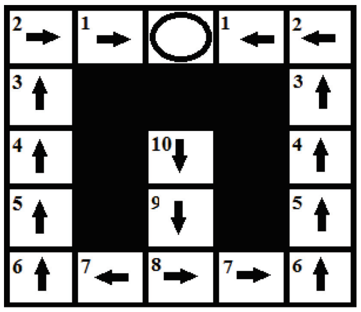

Figure 1: DHPA cache for a single abstract node in a cluster of size 5. In this figure, we do not consider diagonal distance, for simplicity.*

In contrast, DHPA* has no SG effort. At the beginning of the abstract search, we identify the abstract graph nodes that belong to the start cluster. These nodes are then pushed onto the open list as the initial search space, in place of the abstract node that HPA* adds to the graph. The costs for each of these nodes are retrieved from the cache that we previously built, rather than searching for the costs as HPA* does. Then, we search the abstract graph.

Note that the termination condition is also modified to avoid changing the terrain representation (Figures 2a and 2b). We do not add the goal node to the abstract graph, and we do not search for the goal node. Instead, we search for any abstract node that belongs to the cluster containing the goal position. Searching in this manner contributes to the path suboptimality, because it is possible that the first abstract node encountered inside the goal cluster is not part of the optimal path. After the search is complete, we append the start and goal positions to the beginning and end of the solution path, respectively.

Next, we refine the path by running the low-level search. Searches that start and end in the same cluster run a traditional A* search with a Euclidean distance heuristic. Searches that span two or more clusters use the previously cached path indices to find the segment paths within each cluster.

A path index refers to the following node that lies on the optimal path from the current node to the abstract node. It specifies one of the eight adjacent move directions surrounding the current grid cell, similar to a Movement Vector (Jansen and Sturtevant 2008).

The path index enables us to recover any path between any pair of nodes `(s,g)`, where either `s` or `g` is an abstract node. Planning any such path involves only `l` operations, where `l` is the length of the path, and the basic operation is to retrieve the path index from the cache. Thus, the DHPA* refinement search performs exactly `l` operations for a single cluster, since every refinement search either starts or ends with an abstract node (except for refinement searches that start and end in the same cluster).

```
HasHPASearchSuccessfullyCompleted(currNode)
1   if (currNode.ID == goalNode.ID) then
2       return true
3   end if
4   return false
HasDHPASearchSuccessfullyCompleted(currNode)
1   if (currNode.clusterID == goalClusterID) then
2       return true
3   end if
4   return false
```

Figure 2 (a, b): HPA and DHPA* termination conditions. "currNode" denotes the node most recently removed from the open list and added to the closed list. SHPA* uses the same termination condition as DHPA*.*

In comparison, HPA* uses A* for its refinement search. A* is $O(n^2)$ in the worst case, and $O(l)$ in the best case, for a cluster of size $n \times n$ and a path of length l. The basic operation of A* expands a single node, which requires a constant number of priority queue operations, and is much more expensive than the constant-time cache lookup performed by the basic operation of the DHPA* refinement search. The priority queue operations can be done in time logarithmic in the number of nodes in the queue, where a single operation consists of either removing a node from the queue or updating one of the eight node neighbors (Jansen and Buro 2007). If we assume an average path length of n, which is often the approximate segment path length according to our experimental results[2], then the HPA* refinement search is $O(n^2 \log(n^2))$ in the worst case, and $O(n \log(n))$ in the best case. Therefore, for paths of length n that either start or end in a cell with an abstract node, the DHPA* refinement search requires a factor of $\log(n)$ fewer operations than HPA*'s best case search, and a factor of $n \log(n^2)$ fewer operations than HPA*'s worst case search.

[2] After running DHPA* in our second experiment, we found that the segment path lengths differ from *n* by approximately 5%, on average.

SHPA* Build Algorithm

The SHPA* build algorithm directly addresses the concern with using a simple clustering method, and the desire to explore more sophisticated clustering methods (Botea et al. 2004). It runs only once to setup the clusters, and does not repair the clusters dynamically. The build algorithm is the same as described by van der Sterren (2008), and decomposes the map into many variable-sized fully connected clusters based upon a greedy heuristic, rather than decomposing the map into many same-sized clusters as HPA* does. The heuristic attempts to merge clusters based on two different properties. It merges clusters that have a large number of abstract nodes relative to their size, and avoids merging clusters that exceed a maximum combined size. By building fully connected clusters with similar properties, we are able to achieve better spatial decomposition and improve the search algorithm, as we describe next.

SHPA* Search Algorithm

The SHPA* search is similar to the DHPA* search, but it does not use a cache. Instead, it uses a Euclidean distance heuristic to compute edge weights, and it searches for low-level paths using A*. The exact details of the search are described by Kring (2010).

Shared Properties

There are three noteworthy properties shared by both DHPA* and SHPA*. First, we are able to deterministically predict the state of the path planner after the first A* cycle, without searching (Kring 2010). The *InitializeHierarchicalSearchStart* function in Figures 3 and 4 demonstrates how we make use of the cached path length to avoid searching. Eliminating the SG effort clearly improves performance, but it also allows us to provide an initial search space without modifying the abstract graph.

Second, DHPA* and SHPA* are both simpler to implement than HPA*, and more suited for multithreading. SHPA* does not require dynamically rebuilding clusters, but more significantly, both algorithms do not modify the terrain during search. HPA* modifies the terrain by inserting and removing the start and goal nodes at the beginning and end of each search. The programmer must manage these terrain modifications for each search, further complicating programming and debugging. Such a system is more tightly coupled, since it requires the state of the terrain to be dependent upon the state of the path searches. Furthermore, modifying the terrain is worse for multithreading, as it increases the chance of performance bottlenecks if multiple searches on separate threads need to access the terrain data. By minimizing terrain changes, we have less memory contention for the terrain representation.

Finally, both of our algorithms are well poised for incremental pathfinding, because we do not modify the terrain representation. It is common practice to incrementally solve paths in games. Higgins (2002) was one of the first to illustrate the importance of architecting a navigation system capable of incremental pathfinding. More recently, Sturtevant (2010) reported working on an incremental pathfinding system for *Dragon Age: Origins*. Since HPA* modifies the terrain representation at the beginning and end of each search, it is unclear as to how the algorithm would maintain the temporary terrain state when several agents are simultaneously solving paths across multiple frames. We do not modify the terrain representation during search, so maintaining a temporary terrain state is not a concern for either of our algorithms.

```
HierarchicalSearch(startPos, goalPos)
1   InitializeHierarchicalSearchStart(startPos,goalPos)
2   goalCluster = GetCluster(goalPos)
3   StoreGoalClusterID(goalCluster.ID)
4   abstractPath = SearchForPath(startPos, goalPos)
5   lowLevelPath = RefinePath(abstractPath)
6   return lowLevelPath
```

Figure 3: DHPA hierarchical search*

```
InitializeHierarchicalSearchStart(startPos, goalPos)
1   startCluster = GetCluster(startPos)
2   for each abstract node b in startCluster
3       b.g = GetPathLengthFromCache(startPos, b)
4       b.h = ComputeHeuristicCost(b.pos, goalPos)
5       AddToOpenList(b)
6   end for
```

Figure 4: Initialize DHPA hierarchical search start. SHPA* uses this same algorithm, only it uses a Euclidean distance to compute the cost in line 3.*

3. Dynamic Performance Metric

Often research papers on hierarchical pathfinding in games make conjectures about dynamic performance, but rarely are they backed by experimental data. Here, we define a dynamic performance metric for hierarchical pathfinding in games. The metric is a measure of the total time spent on path requests per dynamic change.

Since hierarchical pathfinding usually consists of building and searching, we propose the following experimental steps for measuring the dynamic performance of a hierarchical pathfinding algorithm.

1. Build any data needed for the path searches.
2. Run x path searches.
3. Randomly move all dynamic obstacles on the map.

A single data point amounts to the total time spent building and searching, after completing the steps above. All of these steps take place online, and any offline computations do not factor in to the dynamic performance. Furthermore, the dynamic obstacles must occupy a constant percentage of each map, in order to compare the dynamic performance across multiple maps. In our third experiment in the following section, we use this metric to compare the dynamic performance of DHPA* to HPA*.

4. Empirical Evaluation

We conducted our experiments on 120 maps from *Baldur's Gate* (BioWare Corp. 1998), using the same maps used in the experiments by Botea et al. (2004). The maps range in size from 50 x 50 to 320 x 320. For all search results, we randomly pick start and goal nodes. The timings were performed on a 2.53 GHz Intel Core2 Duo CPU E7200 with 2 GB of memory. The experiments were implemented in The AI Sandbox (Champandard 2010), compiled using Visual Studio 2008, and run under Windows XP Pro.

The first experiment compares the build performance of DHPA* to HPA* (Figure 5). We accumulated the build times of both algorithms for each cluster on all 120 *Baldur's Gate* maps. We conducted the experiment for clusters of size five, ten, and twenty. In all three experiments, the HPA* build algorithm is marginally faster than the DHPA* build algorithm, which agrees with our hypothesis and the results found by Jansen and Buro[3] (2007).

Cluster size	Total clusters	Total HPA* build time (sec)	Total DHPA* build time (sec)
5	32044	33.826	37.223
10	10121	30.318	31.525
20	3338	19.779	21.087

Figure 5: Comparison of total build times for HPA and DHPA*.*

In the largest map that we tested, when using a cluster size of 10, DHPA* created 809 clusters and 547441 cache entries. Each cache entry occupies eight bytes (one float and one integer), resulting in a total cache size of approximately 4.2 MB. On the same map, HPA* created a cache with 36488 entries, occupying 142 KB when storing the entries as floats.

Storage utilization can be optimized by keeping only the necessary parts of the cache in memory, and by reducing the size of each cache entry. Each search requires only the parts of the cache corresponding to the clusters covered by that search, which is also the case for HPA* (Botea et al. 2004). In a cluster of size 10, each cache entry only needs three bytes—two for the path length and one for the path index. The maximum path length is 100, and we only use two digits of precision, because we approximate diagonally connected grid cells with a weight of 1.42. This requires seven bits for the digits to the left of the decimal point, and seven bits for the digits to the right. The path index requires three bits, because it can specify eight directions. If we had used three bytes per cache entry, we would have

[3] Jansen and Buro's experiment measured the build time taken to create the HPA* edge weight cache, comparing A* to Dijkstra's algorithm.

only needed 1.6 MB for the cache. If greater memory efficiency is desired, one can consider a minimal-memory abstraction (Sturtevant 2007), though build time will increase and dynamic performance will likely decrease as a result.

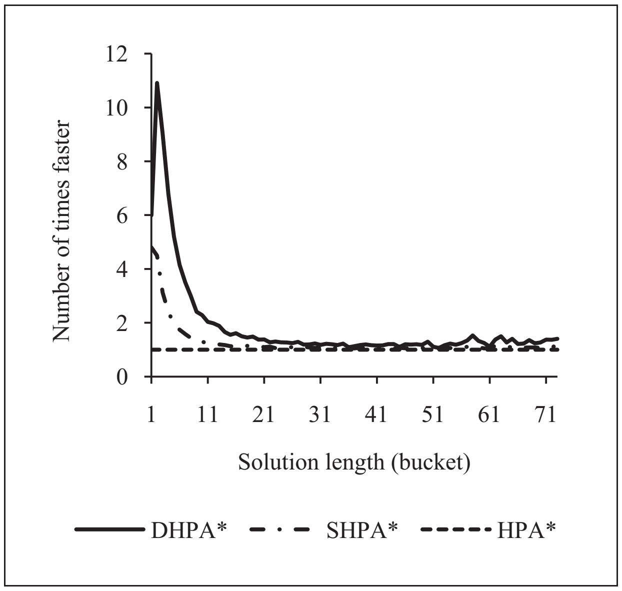

Figure 6: Relative search speed increase over HPA.*

The second experiment compares the search performance of HPA*, DHPA*, and SHPA* (Figure 6). Each data point on the graph represents a single bucket of searches categorized by A* path length. A single bucket contains the average results of running many searches with the same A* path length, and each bucket stores all paths within a range of length ten. For example, the first bucket stores all paths of length [2,9], the second bucket stores all paths of length [10,19], and so on. As a result, the experiment represents a total of ~24,000 searches spread across seventy-three buckets. For this experiment, we used a cluster size of ten.

The best case for the DHPA* search occurs when searching across two clusters, where DHPA* is ~11 times faster than HPA*. On average, the DHPA* search is 1.9 times faster. DHPA* is less than 1% suboptimal to HPA* in all of our results.

The best case for the SHPA* search also occurs when searching two clusters, where SHPA* is ~5 times faster than HPA*. On average, the SHPA* search is 1.3 times faster. SHPA* is at most 6.7% suboptimal, or 4.7% less optimal than HPA*. On average, SHPA* is 1.8% suboptimal, or 0.5% less optimal than HPA*.

There are two interesting points made apparent by comparing the three search algorithms. First, the DHPA* search outperforms both of the other search algorithms because it avoids the SG effort, and it does not have to search for a low-level path. Both SHPA* and HPA*, on the other hand, must search for their low levels paths. Second, both DHPA* and SHPA* see the greatest performance gains when searching for shorter paths. In many games, agents more often search for shorter paths, because most of the action engages in the space local to the player.

The third experiment measures the dynamic performance of DHPA* versus HPA* (Figure 7). We ran our experiment on both DHPA* and HPA*, using a cluster size of ten. A single run of the experiment consists of completing the same steps listed for the dynamic performance metric, ten times for each of the 120 maps. We ran the experiment for `x={10i|i=[1,10]}` searches between each build. Each dynamic obstacle is size `2 x 2`, and 2% of each map is occupied by dynamic obstacles.

HPA* demonstrates superior dynamic performance when there are twenty or fewer searches per build, and DHPA* is superior when there are more than twenty searches per build. Thus, each algorithm is equipped to handle different levels of dynamic change in the environment, and DHPA* demonstrates superior dynamic performance in most cases.

In Figure 7a, we show that the accumulated DHPA* build time is marginally slower. However, the HPA* search time is increasingly slower than the DHPA* search time, as the number of searches per build increases (Figure 7b). In Figure 7c, we see the point where DHPA* begins to outperform HPA*, when there are twenty searches between builds. The graph also shows that the overall performance of DHPA* improves as the environment becomes less dynamic.

5. Conclusions and Future Work

In this paper, we showed how to improve hierarchical pathfinding in games by providing algorithms that are efficient and simple to implement. Both DHPA* and SHPA* are useful in different circumstances. DHPA* searches at best eleven times faster than HPA*, and provides improved performance in dynamic environments when there are more than twenty searches between builds. Though, DHPA* costs more memory and produces slightly less optimal paths. SHPA* is at best five times faster than HPA* for static environments, and costs less memory, but it also produces slightly less optimal paths. Furthermore, both DHPA* and SHPA* simplify coding and enhance multithreaded performance by minimizing changes to the terrain representation.

In the future, we believe that we can reduce the memory cost and increase the performance of DHPA*. We have identified memory consumption as the biggest drawback to DHPA*, and we would like to further investigate efficient ways to encode the cache. We also would like to empirically test our theoretical claims about increasing speed in multithreaded environments, to further demonstrate the importance of maintaining a read-only terrain representation during search.

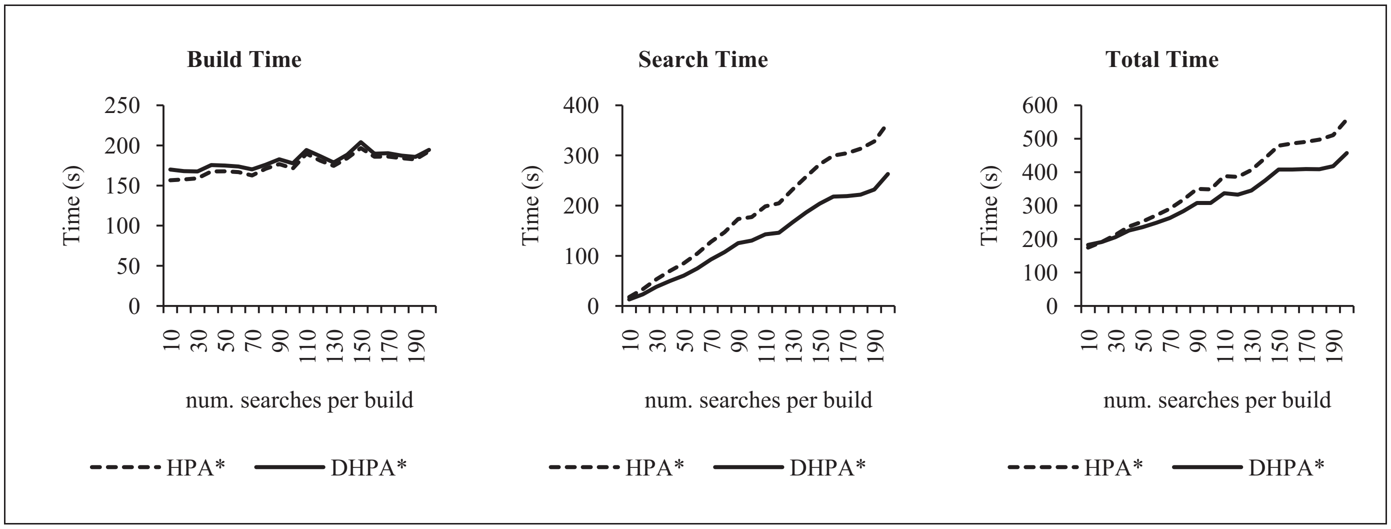

Figure 7 (a,b,c): Dynamic performance of HPA vs. DHPA*.*

References

BioWare Corp. 1998. Baldur's Gate. http://www.bioware.com/bgate/.

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

Bulitko, V., and Björnsson, Y. 2009. kNN LRTA*: Simple Subgoaling for Real-Time Search. In *Proceedings of the Fifth Artificial Intelligence and Interactive Digital Entertainment Conference,* 2-7. Menlo Park, Calif.: AAAI Press.

Champandard, A. J. 2010. The AI Sandbox. http://AiSandbox.com/.

Demyen, D., and Buro, M. 2006. Efficient triangulation-based pathfinding. In *Proceedings of the Twenty-First AAAI Conference on Artificial Intelligence*, 942–947. Menlo Park, Calif.: AAAI Press.

Higgins, D. 2002. Pathfinding Design Architecture. In *Steve Rabin, editor, AI Game Programming Wisdom*, 128-130. Charles River Media, Inc.

Holte, R.; Perez, M.; Zimmer, R.; and MacDonald, A. 1996. Hierarchical A*: Searching Abstraction Hierarchies Efficiently. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence,* 530-535. Menlo Park, Calif.: AAAI Press.

Jansen, M. R., and Buro, M. 2007. HPA* Enhancements. In *Proceedings of the Third Artificial Intelligence and Interactive Digital Entertainment Conference,* 84-87. Menlo Park, Calif.: AAAI Press.

Jansen, M. R., and Sturtevant, N. 2008. Direction Maps for Cooperative Pathfinding. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference,* 185-190. Menlo Park, Calif.: AAAI Press.

Kring, A. 2010. SHPA*: Improving Hierarchical Pathfinding Performance by Maintaining A Static Hierarchy with HPA*. http://aigamedev.com/premium/articles/static-hierarchical-pathfinding/.

Sturtevant, N. 2007. Memory-Efficient Abstractions for Pathfinding. In *Proceedings of the Third Artificial Intelligence and Interactive Digital Entertainment Conference,* 31-36. Menlo Park, Calif.: AAAI Press.

Sturtevant, N., and Buro, M. 2005. Partial Pathfinding Using Map Abstraction and Refinement. In *Proceedings of the Twentieth National Conference on Artificial Intelligence,* 1392-1397. Menlo Park, Calif.: AAAI Press.

Sturtevant, N., and Champandard, A. J. 2010. High-Performance and Memory-Efficient Pathfinding in Dragon Age with Nathan Sturtevant. http://aigamedev.com/premium/interviews/pathfinding-dragon-age/.

Teich, T.; Jurney, C.; and Champandard, A. J. 2010. Case Studies: AI in Recent Games. *Game Developers Conference.*

van der Sterren, W., and Champandard, A. J. 2008. Automated Terrain Analysis with William van der Sterren. http://aigamedev.com/premium/masterclass/automated-terrain-analysis/.

Walsh, M. Dynamic Navmesh - AI in the Dynamic Environment of Splinter Cell: Conviction. 2010. *Game Developers Conference.*