HPA* Enhancements

M. Renee Jansen and Michael Buro
Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{maaike,mburo}@cs.ualberta.ca

Abstract

In video games, pathfinding must be done quickly and accurately. Not much computational time is allowed for pathfinding, but realistic looking paths are required. One approach to pathfinding which attempts to satisfy both of these constraints is to perform pathfinding on abstractions of the map. Botea *et al.*'s Hierarchical Pathfinding A* (HPA*) does this by dividing the map into square sectors and defining entrances between them. Although HPA* performs quick pathfinding which produces near-optimal paths, some improvements can be introduced. Here we discuss a faster path smoothing method, an alternative way to compute the weights of abstract edges, and lazy edge weight computations.

Introduction

Pathfinding algorithms are required for many commercial video games, since most games include characters who need to find their way around the world. To make the AI agents' behaviour look realistic, the agents must start moving quickly; they do not have unlimited time to plan. In addition, the agents should choose a path that quickly leads to the goal. Extremely long paths will make the agent look unintelligent.

The standard approach to pathfinding is running the A* algorithm (Hart, Nilsson, & Raphael 1968) on a tile-based representation of the map. Although A* generates optimal paths, it is computationally expensive, especially for large state spaces. One way to speed up pathfinding is to construct spatial abstractions which retain the topological structure of planar map graphs but contain less nodes. A* often runs considerably faster on such abstractions of the map, while the quality of the generated paths is very good. Recently developed algorithms based on this idea are Path-Refinement A* (PRA*, (Sturtevant & Buro 2005)), which abstracts cliques to single nodes, and Triangulation-Reduction A*(TRA*, (Demyen & Buro 2006)), which builds a triangulation of the map and builds a compact topological description of the map by collapsing corridors and tree components. An earlier hierarchical pathfinding algorithm is HPA* (Botea, Müller, & Schaeffer 2004). This method builds abstractions of the search space by segmenting the map into sectors. It assigns entrances along the borders between the sectors, which become nodes in the abstract graph. The example in Figure 1 shows a small map. Edges are added between corresponding entrance nodes in adjacent sectors as well as between entrance nodes within the same sector if there is a path between them which lies entirely within the cluster. The weight for an edge is set to be the cost of a shortest path in a lower level of the abstraction. The path corresponding to the edge weight can either be cached or recomputed when it is needed.

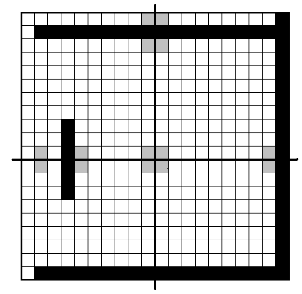


Figure 1: Example of how abstract nodes are chosen in HPA* (Adapted from (Botea, Müller, & Schaeffer 2004)). The sector size is 10×10. Black squares are obstacles and the thick black lines indicate sector boundaries. Gray squares are entrance nodes.

It is possible to build more than one level of abstraction for HPA*. This is done incrementally, by combining multiple sectors at level $n-1$ into a single sector at level n. All entrance nodes on level $n-1$ which lie along the borders of sectors on level n will become entrance nodes in level n.

To perform a pathfinding operation, HPA* first inserts the start and goal nodes into the abstract graphs. Next, an A* search is done on some abstract level and the resulting path is refined level by level by replacing each edge by a lower-level path. This is done until the map level is reached. The paths corresponding to each edge are either retrieved from the cache or recomputed. When the path at the map level is found, smoothing can be done to shorten the path.

In the rest of this paper, we study improvements to HPA*. The reasons for this are that HPA* can find near-optimal paths in very little time, and that it has the property that changes in the world can be handled efficiently because changes will only have a local effect. Next, we propose three extensions to this algorithm, namely a faster smoothing algorithm, an alternative way of computing edge weights, and lazy edge weight computations. We then show experimental results and we end with a discussion of future work and some concluding remarks.

HPA* Enhancements

In this section we explain the our HPA* extensions in detail.

Faster Path Smoothing

Although HPA* can find an initial path very quickly, this path is not usually optimal. To reduce the path length, *smoothing* can be done, which replaces parts of the path with straight lines. The smoothing method used by Botea *et al.* does this by shooting rays in all eight directions from each node n on the path. When a ray reaches a node m further along the path, the original part of the path between n and m is replaced by a straight line and the smoothing process continues with the node two positions before m on the improved path. The resulting paths have lengths close to the optimal length, but the computation is expensive. A straightforward way of addressing this is by placing a *bounding box* around the entire path, *i.e.*, a box spanning from the lowest x-and y-coordinates of any node on the path to the largest ones, and only tracing rays inside this bounding box. However, because paths may span a large portion of the map, smoothing is slow even when we only trace rays inside this box. Therefore, we propose placing a smaller bounding box around the current position in the path, which will reduce the time spent on smoothing but could potentially sacrifice some optimality. In the experimental results section we will show that the time reduction is significant, while the paths are only slightly longer.

Dijkstra's Algorithm

To compute the weights of edges inside a sector, HPA* performs an A* search between each pair of entrances. We propose using Dijkstra's single-source shortest path algorithm for each entrance node to compute weights of all the edges adjacent to it because the worst-case running time for A* is worse than that for Dijkstra's single-source shortest path, as outlined below.

A* Running Time. Assuming square sectors, let $L \times L$ be the sector size and E the number of entrances. Then $E \leq 4L - 4$, because there are that many nodes located on the edges of the sector. There are $\binom{E}{2}$ pairs of entrances, which is $O(L^2)$. In the worst case A* expands all L^2 sector nodes in each run. For each node that is expanded, a constant number of priority queue operations must be performed, namely removing the node from the queue, and updating the neighbours of this node (at most 8 in an 8-connected grid world). These operations can be done in time logarithmic in the number of nodes in the queue, namely $O(\log(L^2)) \subseteq O(\log L)$. Therefore, the worst-case total running time of A* to determine distances between each entrance pairs is $O(E^2 L^2 \log L) \subseteq O(L^4 \log L)$.

Running Time of Dijkstra's Algorithm. Again, let $L \times L$ be the sector size, and E the number of entrances. The priority queue is implemented with a binary min-heap, so the cost of extracting the vertex with minimum weight is $O(\log L^2) \subseteq O(\log L)$. For every vertex that is removed from the queue, we apply at most 8 decrease-key operations. Each of these also has cost $O(\log L)$. Therefore, the total worst-case running time for Dijkstra's algorithm computing all distances between entrances is $O(EL^2 \log L) \subseteq O(L^3 \log L)$.

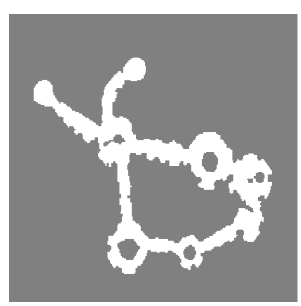
Relationship between running times. The established worst-case runtime bounds for both algorithms are tight. This can be seen by considering sectors with $E/2$ entrances at the north and south edges and a single zig-zag path of length $\Theta(L^2)$ in the sector interior connecting the north and south edge entrances. In this case using Dijkstra's algorithm leads to an asymptotic improvement over A*'s worst case running time by a factor of $1/E$, hinting at an advantage of Dijkstra's algorithm for maps with sectors with many entrances and long or complex paths between them. In the best case, the number of nodes A* expands is linear in the total path length between all entrance pairs, amounting to $\Theta(E^2 L \log L)$ operations. In this situation, Dijkstra's algorithm will likely explore all interior nodes if the entrances are scattered along the sector perimeter, and thus take $\Theta(EL^2 \log L)$ operations. Consequently, when considering large empty sectors with only a few entrances, the entrance distance computation based on A* will be asymptotically faster.

Lazy Edge Weight Computation

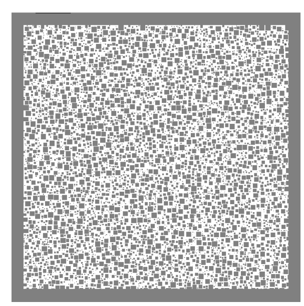
Because HPA*'s abstraction is based on independent sectors, changes in the map will only affect the sectors which are local to the changes. This makes HPA* particularly suitable for use in dynamic environments.

One way to deal with these dynamic environments, as proposed by Botea *et al.*(Botea, Müller, & Schaeffer 2004), is to recompute the entrances and edge weights for affected sectors when changes happen. However, in dynamic environments it is possible that the sector is changed again before this recomputed information is needed. In such cases it is more efficient to compute edge weights on demand, thereby amortizing the cost of computing edge weights.

We therefore propose using a lazy approach to computing edge weights. Instead of recomputing the weights of edges when a sector changes, we insert edges between all pairs of entrances in the sector and we mark them *invalid*. When we perform a search, we find the weight of an edge whenever a node adjacent to it is expanded. This is done by recursively searching in each of the levels until the map level is reached, similar to how it would be done in the precomputation phase. Edge weights are then percolated up the abstraction until the edge at the topmost level has been assigned its proper weight. The advantage of using this approach is that if some edges are never needed for a search, their weights will not be computed. In the worst case, the lazy approach performs the same amount of work as the eager approach, but if any edge weights are never needed, the lazy approach will perform less computation.



(a) Game map



(b) Artificial map

Figure 2: Map Samples. Only white areas are traversable.

Experiments

The impact of the proposed improvements was evaluated by finding paths on two sets of maps. The first set is comprised of 116 commercial game maps: 75 maps from Bioware's *Baldur's Gate* and 41 from Blizzard Entertainment's *Warcraft III*. The maps were scaled up to 512 $\times$ 512. An example is shown in Figure 2(a). The second set consists of 80 artificial maps, also of size 512 $\times$ 512. For these maps, square obstacles were placed on an empty map such that they do not touch one another. The percentage of the map that is blocked is varied from 10 to 80, and the maximum size of an obstacle is varied from 1 to 10 (*i.e.*, if the maximum size is 10, the map can have obstacles of size 1, 2,...,10). An example of an artificial map is shown in Figure 2(b). For each of these maps, random pairs of points were chosen such that the optimal distance between them was at most 512. The paths were divided into 128 *buckets*: a path is in bucket i iff $i = \lfloor l/4 \rfloor$, where l is the optimal path length. For the game maps, over 144,000 pairs of points were used for each experiment, with the number of paths in each bucket ranging between approximately 800 (for very long paths) and 1150 (for shorter paths). For each of the 80 artificial maps, 10 paths in each bucket were used, giving a total of 102,400 pairs of points, equally distributed over the buckets.

In our implementation of HPA*, we introduce a single abstract entrance node for an entrance of at most length 5, and two abstract entrance nodes for larger entrances. In addition, at higher levels of abstraction we combine 2$\times$2 lower-level sectors into a single abstract sector. Lastly, in the eager case all paths are cached — they do not need to be recomputed when the path is refined. We will present graphs for two types of results: relative path length and time to find a path. In both cases, these will be graphed against the A*, or optimal, path length. For each of these, graphs contain percentiles, *i.e.*, 5% of paths lie above the 95th percentile line, and so on. All timing experiments were run on a machine with two 1 GHz Pentium III processors and 512 MB RAM.

Smoothing

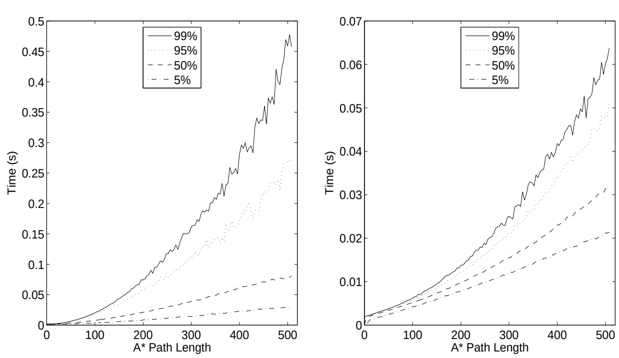
To evaluate the proposed smoothing method, we performed experiments with varying sizes of bounding boxes. In one case, we build a bounding box around the entire path. In other words, we allow the x-values to range from the smallest to the largest x-values of any node on the path, and similar for the y-values. In the other cases we restrict the bounding box to some small square centered at the current position. In particular, the dimension of the bounding box, centered at (x, y), was varied between (x±5, y±5) and (x±20,y±20), *i.e.*,the length of the box was varied from roughly one to four times the sector size.

Intuitively, we expect the pathfinding time to increase as the size of the bounding box increases because longer rays must be traced during the smoothing process. We also expect the sub-optimality to decrease as the size of the bounding box increases since the algorithm traces longer rays, increasing the chances of reaching another part of the path. The experimental data confirms both these intuitions. Figures 3 and 4 show the results for a representative experiments on the set of game maps. A single level of abstraction with sector size 10 was used and edge weights were computed using Dijkstra's algorithm. The left-hand figures show the time spent to find paths, which is significantly higher for the case with the large bounding box, as expected. The right-hand figures show the path quality. Although the algorithm performs worse with the smaller bounding box, the difference — especially for longer paths — is only approximately 1% in the worst case. The usefulness of adjusting the size of the bounding box lies in the fact that a balance can be struck between shorter paths and faster pathfinding. The parameter can be adjusted according to time or path quality requirements. For example, up to distance 100 — where smaller bounding boxes lead to relative errors larger than 1.03 in 5% of the cases — one could use larger bounding boxes to improve path quality and suffer only a minor runtime penalty.

Dijkstra's Algorithm

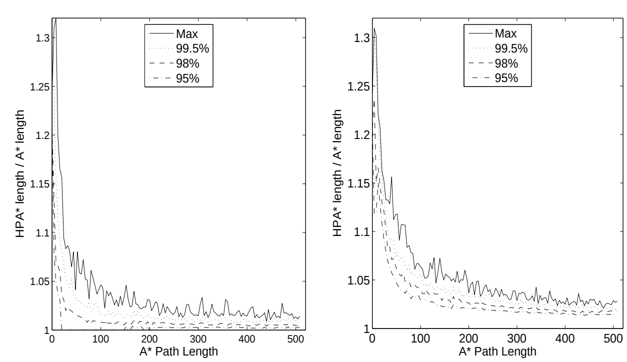
To compare the effect of using Dijkstra's algorithm versus using A* to compute edge weights, we performed experiments on both sets of maps for three different sector sizes and between one and three levels of abstraction.

Game Maps. The first set of experiments was performed on the game maps. Tables 1 and 2 show the time spent on



(a) Large bounding box (b) Bounding box ranging from (x-5,y-5) to (x+5,y+5)

Figure 3: Time spent on pathfinding when smoothing with different bounding box sizes



(a) Large bounding box (b) Bounding box ranging from (x-5,y-5) to (x+5,y+5)

Figure 4: Path quality when smoothing with different bounding box sizes

Table 1: HPA* using A* to compute edge weights. No smoothing. Game maps. *Build* indicates the total computation time in seconds to build the HPA* representation for all 114 game maps. *Path* indicates the total time to find all paths on all maps which includes adding the start and goal nodes to the abstraction graph. The shortest pathfinding time is emphasized.

	Sector Size 5			Sector Size 10			Sector Size 20		
Lvl.	Build	Path	Total	Build	Path	Total	Build	Path	Total
1	597.48	3118.37	3715.85	651.45	**1057.02**	1708.47	903.12	1219.82	2122.94
2	762.90	2093.52	2856.41	832.87	1213.56	2046.43	960.75	1461.38	2422.12
3	1027.77	1659.27	2687.04	1395.55	2273.28	3668.83	1140.23	2804.71	3444.94

Table 2: HPA* using Dijkstra's algorithm to compute edge weights. No smoothing. Game maps.

	Sector Size 5			Sector Size 10			Sector Size 20		
Lvl.	Build	Path	Total	Build	Path	Total	Build	Path	Total
1	675.61	3163.10	3838.71	743.40	**1096.83**	1840.23	1008.87	1196.24	2205.11
2	785.63	2120.97	2906.60	797.36	1121.18	1918.54	1025.68	1235.57	2261.25
3	899.77	1472.71	2372.48	878.66	1142.58	2021.24	1057.81	1364.04	2421.85

Table 3: HPA* using A* to compute edge weights. No smoothing. Artificial maps.

	Sector Size 5			Sector Size 10			Sector Size 20		
Lvl.	Build	Path	Total	Build	Path	Total	Build	Path	Total
1	722.47	5343.01	6065.49	1308.43	2791.80	4100.24	2403.61	2126.79	4530.40
2	1213.79	3061.99	4275.78	2231.80	**1926.67**	4158.47	4053.77	2901.60	6955.38
3	2985.08	2905.47	5890.55	5230.43	3470.52	8700.96	9035.56	8337.21	17372.78

Table 4: HPA* using Dijkstra's algorithm to compute edge weights. No smoothing. Artificial maps.

	Sector Size 5			Sector Size 10			Sector Size 20		
Lvl.	Build	Path	Total	Build	Path	Total	Build	Path	Total
1	672.22	5501.56	6173.78	954.87	2857.86	3812.73	1312.89	1879.64	3192.52
2	954.59	2109.24	4063.83	1248.94	**1670.43**	2919.37	1735.63	1632.04	3367.66
3	1803.92	2490.78	4294.70	2228.62	1713.25	3941.87	3202.39	2465.34	5667.73

building the abstractions and finding the paths for A* and Dijkstra's algorithm, respectively. Both algorithms perform fastest with a single level of abstraction and sector size 10, but A* performs a bit better. In fact, A* performs slightly better for any sector size with a single level of abstraction, but as the number of abstract levels increases, Dijkstra performs better, at least for larger cluster sizes. This can be explained by the fact that A* needs to expand larger numbers of nodes when the (abstract) sectors are larger.

Artificial Maps. The results for the second set of experiments, which use the artificial maps, are shown in Tables 3 and 4. The maps were designed to increase the number of entrances in any sector. The expectation was that this would increase the amount of work done by A* relative to Dijkstra's algorithm, and the experimental results confirm that this is the case. In particular, the A* approach spends more time building the abstractions except in the case of a single level of abstraction with cluster size 5. Dijkstra's algorithm is slightly slower for pathfinding in some cases, but 14% better overall in the best setting (sector size 10 and two levels of abstraction). In this case Dijkstra's algorithm also precomputes weights almost twice as fast as A*.

Table 5: HPA* using Dijkstra's algorithm with sector size 10 and 1 level of abstraction, no smoothing.

	Time to build (s)		Time to find paths		Total(s)
	Total(s)	Avg(s)	Total(s)	Avg(ms)	
Eager	743.40	6.41	1096.83	7.61	1840.23
Lazy	350.46	3.02	1409.14	9.78	1759.60

Lazy Edge Weight Computation

As an investigation into the benefits of performing lazy weight computation as opposed to precomputing edge weights, experiments were run with Dijkstra's algorithm, a single level of abstraction, sector size 10, and no smoothing. Table 5 shows some cumulative results for these experiments. First of all, it is clear that the initial time to build the abstraction is much smaller when paths are not precomputed, as expected. The fact that the total time for the lazy approach is smaller than the total time for the eager approach indicates that for our experiments, the edge weights are never computed for some sectors. It seems likely, for example, that sectors near the borders of a map are less likely to be used. It is unclear whether this is often the case in practice, but if it is, this method could be particularly useful since the speedup in the building of the map is significant.

Conclusion and Future Work

In this paper we have presented three extensions to HPA*. We have shown that smoothing can be sped up at a cost of slightly longer paths. We have also shown that using Dijkstra's algorithm instead of A* can be beneficial for computing the edge weights. Lastly, we have proposed a novel approach for using HPA* in dynamic environments. Instead of pre-computing edge weights it may be beneficial to compute them on demand. Preliminary results are promising, but further investigation is needed to accurately assess the merits of this approach in dynamic environments.

Looking beyond fixed game maps for which pathfinding data in form of abstraction graphs and edge weights can be precomputed, interesting questions are how HPA*, PRA*, and TRA* can be extended to work efficiently in large dynamic environments featuring many moving objects, changing topologies, and incremental terrain discovery.

References

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

Demyen, D., and Buro, M. 2006. Efficient triangulation-based pathfinding. In *Proceedings of AAAI*, 942–947.

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.

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