Bachelor Thesis

Hierarchical Path-Finding Theta*

Combining HPA* and Theta*

21st August 2013

Author:
Linus van Elswijk
s0710261
linusvanelswijk@student.ru.nl

Supervisor:
dr. I.G. Sprinkhuizen-Kuyper

Second Reader:
dr. F. Wiedijk

Abstract

Pathfinding is a common problem in computer games. Pathfinding problems are often solved by running A* on a grid representation of the terrain. This leads to problems if the terrain is large and/or continuous. Although A* is able to produce the optimal path on a grid, this path is often unrealistically angular and noticeably suboptimal on the continuous terrain. Since the number of expansions A* needs to do are exponential in the length of the returned path, the run time and memory usage becomes an issue for large terrains. In this paper we present a new algorithm, Hierarchical Path-Finding Theta*, that combines techniques of exisiting algorithms Theta* and HPA*. We show that this new algorithm is able to solve the problems we identified with A*.

Radboud University Nijmegen

Contents

1 Introduction

In this paper we will be investigating pathfinding in computer games, with a focus on large, two-dimensional continuous terrains, represented as maps of square cells. These cells are either blocked or walkable.

Pathfinding problems have to be solved in many computer games. In games, there are a lot of demands for the pathfinding algorithms:

- The algorithms, must be able to return a path fast, because you do not want the system to slow down when a path is being computed.
- The paths returned by the algorithm should be near optimal. Agents in the system should not take obvious detours.
- The algorithm must be mild with memory usage, because this memory could very well be required elsewhere in the system.

A common solution to pathfinding is to represent the game maps with grids and compute paths on these grids using the A* algorithm (Hart et al., 1968). This approach leads to problems[1] when applied large, continuous terrain:

1. Since A* runs on grid representation of the terrain, all returned paths will be bound to the nodes and edges of the grid. In many cases this lead to unrealistically angular paths, which are noticeably suboptimal on the continuous terrain. An example of this is shown on figure 1.
2. In practice, the number of expansions of A* will almost always be exponential in the length of the resulting path(Russel and Norvig, 2003). This will pose a problem for pathfinding on large maps where long paths are required to traverse the map.
3. For the same reason, the memory usage of A* will start to be a problem on large maps.

From now on we will refer to problem 1 as the *grid constraint problem*, problem 2 as the *runtime problem*, and problem 3 as the *memory problem*.

A recent variant of A*, Theta* (Nash et al., 2007), is specifically designed to solve the grid constraint problem, but still suffers from both the runtime and memory problem. Theta* is able to produce the dotted path on figure 1.

The paper that introduced Theta*, describes two flavours of Theta*. Basic Theta*, which uses line of sight checks, and Angle Propagation Theta*, which adds extra data over the visited nodes during propagation.

[1] The triad of problems (path length, computation time, memory usage) will be a recurring theme throughout this paper.

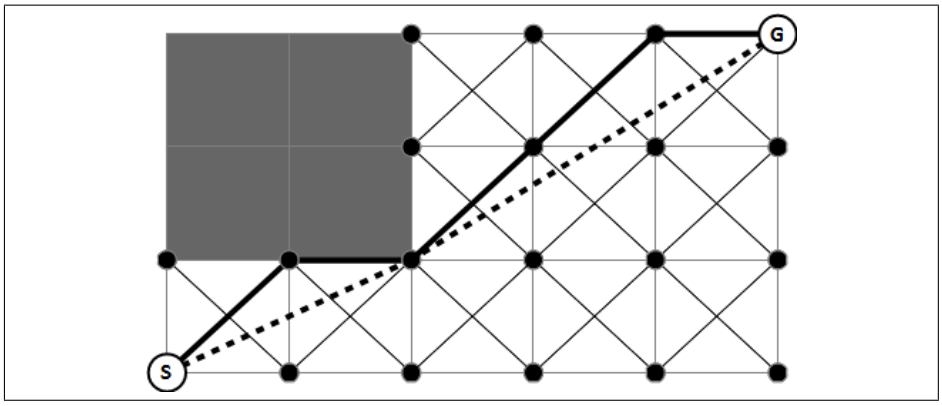


Figure 1: The grid constraint of A*. The thick black line shows a typical A* path from red to blue. The dotted line shows the optimal path on the continuous terrain.

Another variant, Hierarchical Path-Planning A*(Botea et al., 2004), solves runtime problem and the memory problem, but still suffers from the grid constraint problem.

We developed a new algorithm, *Hierarchical Path-Planning Theta** (HPT*), that combines Theta* with HPA*. Our hypothesis is that HPT* solves all 3 problems and the purpose of our research is to test this hypothesis.

2 Description of the Algorithms

In this section we will discussing the different algorithms. The appendix of this paper contains pseudocode for all algorithms described here.

2.1 A*

The classic A*, as described by Hart et al. (1968), is a form of best-first search(Russel and Norvig, 2003). Best-first-search algorithms try to expand the nodes that are closest to the goal first. A heuristic function is used to make an *estimation* of how close a node is to it's goal *before* the node is expanded. A* also keeps track of the cost of the path to reach a node. Instead of just expanding the node that is closest to the goal, A* tries to expand the node for which the sum of the estimated distance to the goal and the cost to reach that node is the smallest. You could say that A* tries to expand the nodes with the smallest estimated solution cost first. Hart et al. (1968) proved that A* always finds the optimal path when the heuristic function never *over*estimates the distance of a node to the goal. The thick black line in figure 1 shows a typical A* path.

2.2 Theta*

Theta*(Nash et al., 2007) is variant of A* that makes use of the fact that the grid on which the search is performed is superimposes a continuous space. Theta* follows the exact same steps as A*, except when expanding. During an expansion, it will check if new node has line of sight with it's grandparent. If this is true, than the parent can be skipped, and a straight path can be drawn to the grandparent. Because the path from the new node to it's grandparent is a straight line, it can never be longer than the path via the parent. The dotted line in figure 1 shows a typical Theta* path. The paper that first described Theta*(Nash et al., 2007), actually describes 2 variants of it: a basic variant that uses line of sight checks and an Angle-Propagation variant, that propagates additional data along the nodes it visits. In our research we will only be looking at the basic variant. Whenever we talk about Theta*, we will implicitly be refering to the basic variant.

2.3 HPA*

Hierarchical Path-Finding A* was first described by Botea et al. (2004). HPA* can provide a large performance boost on large search spaces/-graphs when compared to classic A*. HPA* uses one or more levels of abstraction to reduce the complexity of a problem. In our research we will limit ourselves to a single abstraction level. The abstraction of the search-space is done in three steps:

1. First the graphs is divided in square clusters of user specified size.
2. Then for all neighbouring clusters, the maximal sized entrances are determined.
3. The entrances of neighbouring clusters are connected with an edge cost of 1. Diagonal neighbors are not connected.
4. All nodes that are within the same cluster are connected to eachother, if an A* path exists between them within the cluster. The cost of these edges is set to the cost of the A* path.

In order to calculate a path using the abstract graph, first the goal and start node need to be inserted into the graph. After the start and goal have been inserted on the graph, a normal A* run can be used to compute an abstract solution on the abstract graph. This abstract solution can then be refined into a concrete solution, by connecting the nodes in the abstract solution with each other. The connections are made by doing A* searches on the concrete graph. Since the abstract graph is a lot smaller than the original graph, search problems can be greatly simplified by using the abstract-graph instead of the original graph.

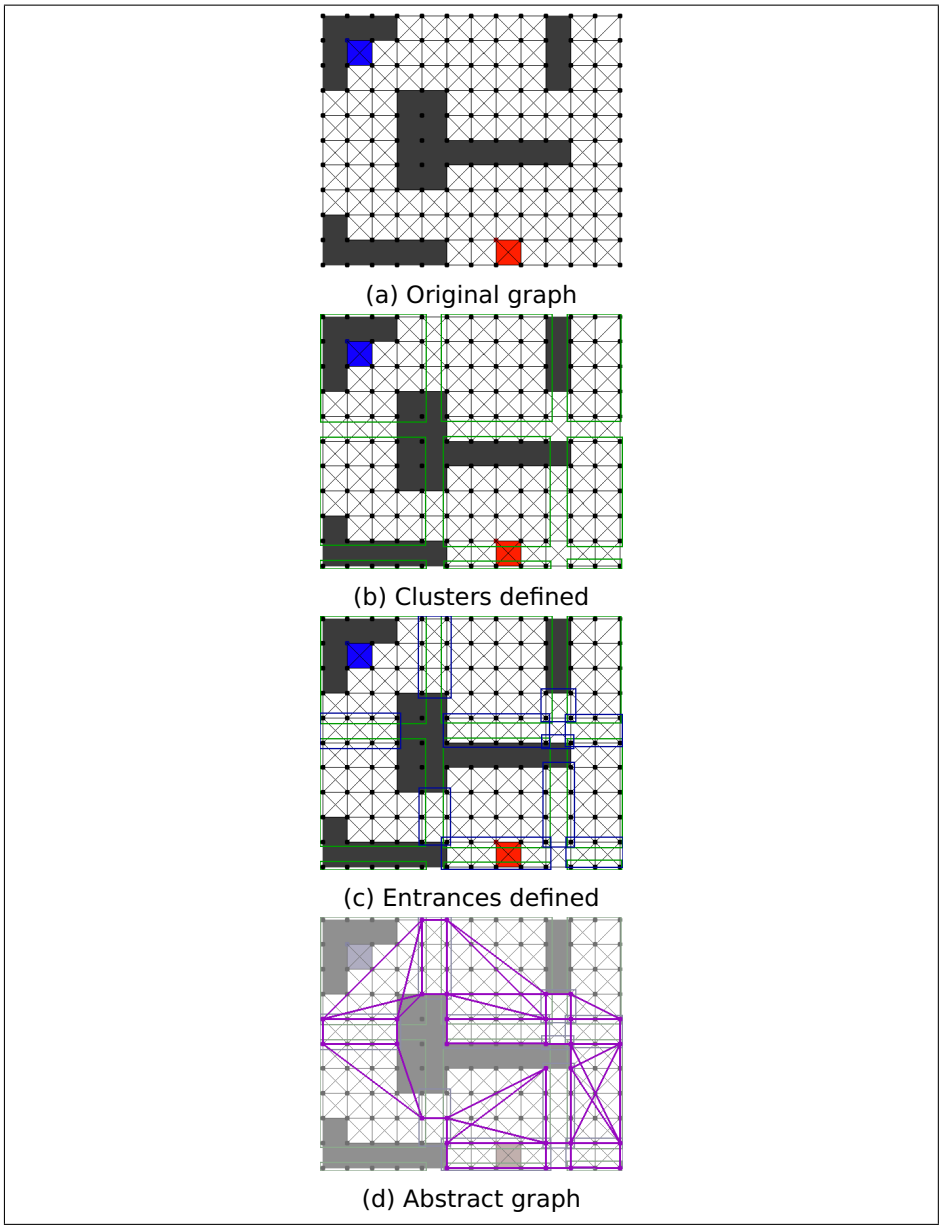


Figure 2: Process of building an abstract graph. Applies to both HPA* and HPT*.

2.4 HPT*

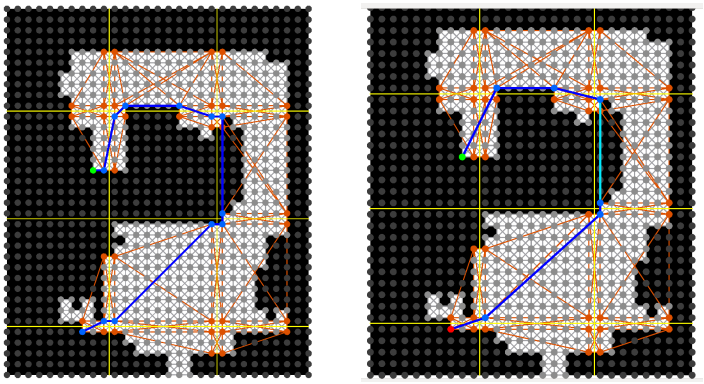
Hierarchical Path-Finding Theta* is created by combining HPA* with Theta*. The abstraction of the search-space is done in three steps, similar to those of HPA*. However, instead of using A* as a subroutine, HPT* uses Theta* as a subroutine:

1. First the graphs is divided in square clusters of user specified

size.

2. Then for all neighbouring clusters, the maximal sized entrances are determined.
3. The entrances of neighbouring clusters are connected with an edge cost of 1. Diagonal neighbors are not connected.
4. All nodes that are within the same cluster are connected to eachother, if an Theta* path exists between them within the cluster. The cost of these edges is set to the cost of the Theta* path.

In order to calculate a path using the abstract graph, first the goal and start node need to be inserted into the graph. After the start and goal have been inserted on the graph, a normal Theta* run can be used to compute an abstract solution on the abstract graph. This abstract solution can then be refined into a concrete solution, by connecting the nodes in the abstract solution with eachother. The connections are made by doing Theta* searches on the concrete graph only if the nodes do not have line of sight with eachother. If the nodes have line of sight with eachoter, the concrete path between the nodes is just a straight line. This last observation could allow HPT* to skip a lot of work on the path refinement phase.



(a) HPA* path (b) HPT* path

Figure 3: HPT* versus HPA*. During path refinement, HPA* needs to connect all nodes on the blue path by doing A* searches. HPT* only needs to do a Theta* search on the light blue line, because this is the only time the two subsequent nodes do not have line of sight.

3 Research Questions

We want HPT* to be a solution to the triad of problems we identified with A* (on page 2). HPT* will only be succesfull in this if it inherits the correct properties from it's parent algorithms, Theta* and HPA*.

What this means is that, on average, for large ammounts of realistic pathfinding problems:

1. HPT* paths should be of roughly the same or better quality as Theta* paths.
2. The computation times of HPT* should be roughly the same or less than the computation times of HPA*.
3. The memory requirements of HPT* should be roughly the same or less than the memory requirements of HPA*.

Notice that on all these points we demand HPT* to be only *roughly* as good as one of it's parent algorithms, never better. So even if HPT* performs a little bit worse than the corresponding parent on all 3 points, we will still consider HPT* to be a success. This might seem strange, but it can be explained.

Our goal with HPT* is not to outperform Theta* on path quality and neither to outperform HPA* on computation time or memory requirements. Instead, the goal of HPT* is to be the middle ground between it's parent algorithms. In comparison to Theta*, HPT* should be able to produce paths of near equal quality, in significantly less time, while using significantly less memory. In comparison to HPA*, HPT* should be able to produce paths, of a significantly better quality, but use a bit more time and memory.

3.1 Research Question

This leads us to the following research question:

Does *Hierarchical Path-Finding Theta* succesfully combine* the *benefits of Theta** with the *benefits of HPA**?

Before we can answer this question, we first need to be clear about what it's meaning:

- "The benefits of Theta*" refers to the average solution path quality of Theta*.
- "The benefits of HPA*" refers to both the average computation time of HPA* and the average memory requirements of HPA*.
- A succesfull combination of the benefits requires the following to be true:
    1. HPT* produces paths that are not more than 10% longer, on average, when compared to Theta*.
    2. HPT* requires not more than 10% more node visits, on average, to compute paths when compared to HPA*.
    3. HPT* requires not more than 10% more memory, on average to, compute paths when compared to HPA*.

In our research we will use the length of the returned path as the measure of quality. We will use the number of node visits to measure the computation time of the algorithms. By measuring the number of visits, instead of the real time in seconds, we can rule out any outside influence affecting the data. The memory usage is measured by counting the maximum number of nodes in the open and closed list combined. Again, to rule out any outside influence affecting the data.

3.2 Subquestions

The research question can be answered after we know the answer to the following subquestions:

1. What is the average path length of HPT* in comparison relative to Theta*?
2. What is the average number of visits required by HPT* relative to HPA*?
3. What is the average of the maximum number of nodes in the open and closed list combined, relative to HPA*?

4 Strategy

To answer our questions, we will let Theta*, HPA* and HPT* algorithms compute a large number of paths on a set of maps. For each algorithm we will record for all problems

1. the length of the path produced by the algorithm,
2. the number of node visits the algorithm had to do to produce the path and
3. the maximum number of nodes in the open and closed list combined during the computation.

For each algorithm a we calculate

1. l_a the average path length over all problems
2. v_a the average number of node visits over all problems
3. m_a the average, over all problems, of the maximum number of nodes in the open and closed list combined

This absolute data can then be used to calculate the required relative data:

$$L = \frac{l_{\text{HPT*}}}{l_{\text{Theta*}}} \tag{1}$$

$$V = \frac{v_{\text{HPT*}}}{v_{\text{HPA*}}} \tag{2}$$

$$M = \frac{m_{\text{HPT*}}}{m_{\text{HPA*}}} \tag{3}$$

L, V and M answer subquestion 1,2 and 3 respectively. To answer the research question and determine if HPT* is a succesful combination of Theta* and HPA*, we need to check if either L, V or M is larger then 1.1. Only if neither L, V nor M are larger than 1.1, then we may conclude that HPT* is a succesful combination.

4.1 Pathfinding Problems Used

It is important that the maps and problems are realistic and provide a proper representation of pathfinding problems the algorithms might face in practice. We will therefore use the maps from "Dragon Age: Origins", which can found on http://www.aiide.org/benchmarks/maps (Sturtevant, 2011) and corresponding pathfinding problems of these maps. All maps in this repository are maps from real games and made discrete by using square tiles. The tile map file format describes multiple types of tiles, but we only distinguish blocked tiles from walkable tiles. Table 1 shows the conversion we used.

ASCII in map file	description	converted to
.	passable terrain	walkable
G	passable terrain	walkable
@	out of bounds	blocked
O	out of bounds	blocked
T	trees	blocked
S	swamp	walkable
W	water	blocked

Table 1: Conversion table from .map file format to binary tile maps

From this repository we chose the maps from "Dragon Age: Origins", because the corresponding problems often require the algorithms to compute long paths (length of 250 or more). Instead of computing the paths for all problems on the "Dragon Age: Origins" maps, we chose to randomly pick 10 maps out of the repository and then compute the paths for these particular maps. This to speed up the process of benchmarking the algorithms.

To pick 10 random maps at random, we generated a random set of 10 (unique) numbers, between 1 and the number of maps available. Each number *n* would then correspond to the *n*th map after the maps where ordered alphabetically. We ended up with the following 10 maps: brc505d, den408d, den600d, lak100c, lak105d, lak110d, lak503d, lak511d, lgt604d and orz300d. We also ommited all problems for which no solution exists, to keep the computation of the averages straightforward. Considering all this, we end up with 9937 path planning problems spread over 10 maps.

4.2 Algorithm and Map Parameters

We interpreted the maps as octile grids, where agents can move to any of the 8 neighboring tiles in a single step.(Yap, 2002). The cost of a horizontal or vertical step is 1, any diagonal step has a cost of $\sqrt{2}$.

Both A* and HPA* will use the octile distance heuristic, since this heuristic correctly predicts the path length when no obstacles are encountered. Theta* and HPT* use the euclidean distance heuristic, because for these algorithms, the euclidean distance correctly predicts the path cost when no obstacles are encountered.

The ammount of visits needed by HPA* and HPT* are measured as the sum of the number of visits on the abstract graph, the visits on the concrete graph during path refinement and the number of visits needed to connect the initial and goal node to the abstract graph. The ammount of nodes in memory needed by HPA* and HPT* are measured as the maximum of the nodes in memory during the abstract search, during the searches on the concrete graph and during the process of connecting the start and goal node.

5 Results

In this section we will show data we recorded with the benchmark test. All graphs in this section were made by plotting the recorded values of individual pathfinding problems. The x-axis is always the A* path length, which gives a good indication of the difficulty of the problem. The y-axis of the graph is always the measured data for the same problem.

The tables present the averages over pathfinding problems, both absolute and relative to the goal algorithm.

5.1 Path Lengths

The graphs on figure 4 show that HPT* often returns better resulting paths than both HPA* and A*. The difference between the algorithms seems to be more pronounced for problems with an A* path length longer than 600. When we compare the average path lengths of HPT* with the average path lengths of Theta* (table 2), we see that HPT* produces path that are only 4.34% longer on average than Theta*

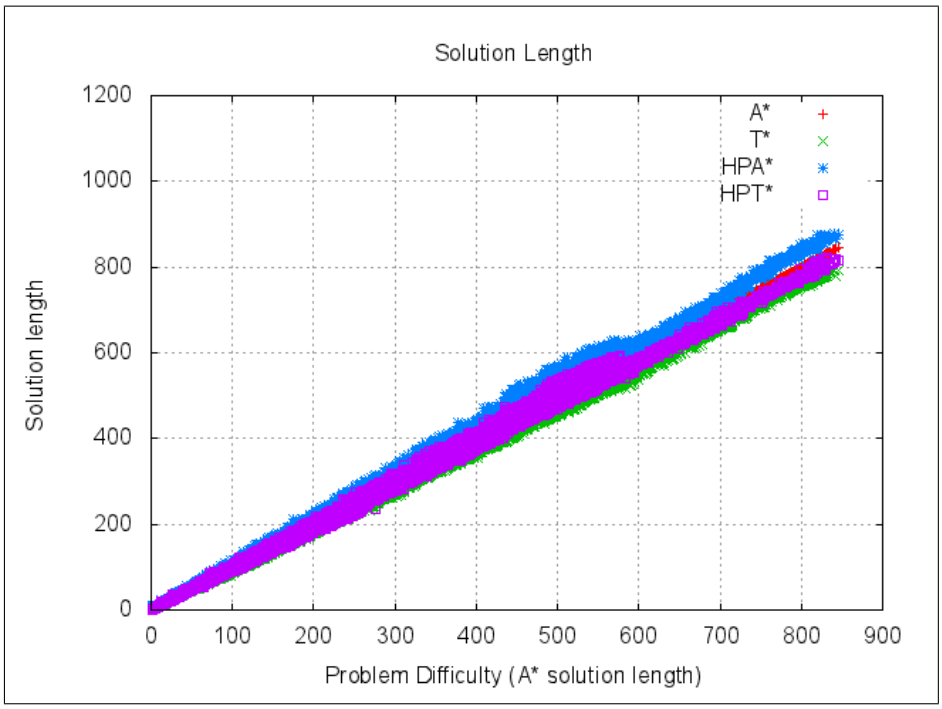


Figure 4: Length of the paths returned by each algorithm.

Algorithm	Average Path Length	Relative to Theta*
A*	282.81	1.0632
Theta*	265.99	1
HPA*	296.16	1.1134
HPT*	277.54	1.0434

Table 2: Average lengths of paths returned by each algorithm.

paths. HPA* produces paths that are 11.34% longer, so we gained 7% in this regard. HPT* also manages to outperform A* on path lengths.

Note: In some cases, our results showed HPA* to outperform A* on the path. HPA* should never be able to outperform A*, since A* produces the optimal paths on the grid. This can only be explained as being the result of a fault in the code we used to do our benchmarks. This is further expanded on in the discussion.

5.2 Node Visits

The graph makes clear why the hierarchical algorithms HPA* and HPT* are needed. As the problems become more complex, the number of node visits for both A* and Theta* start to rise quickly. At about an A* path length of 200, both A* and Theta* begin to surpass 10000 node visits. HPA* only just starts to reach this point at A* lengths of 800. HPT* never reached 10000 node visits during the benchmarks.

Table 3 shows us, that on average, HPT* only requires about 53%

the ammount of node visits of HPA*. This while A* and Theta* require more than 2.5 times the ammount of node visits of HPA*.

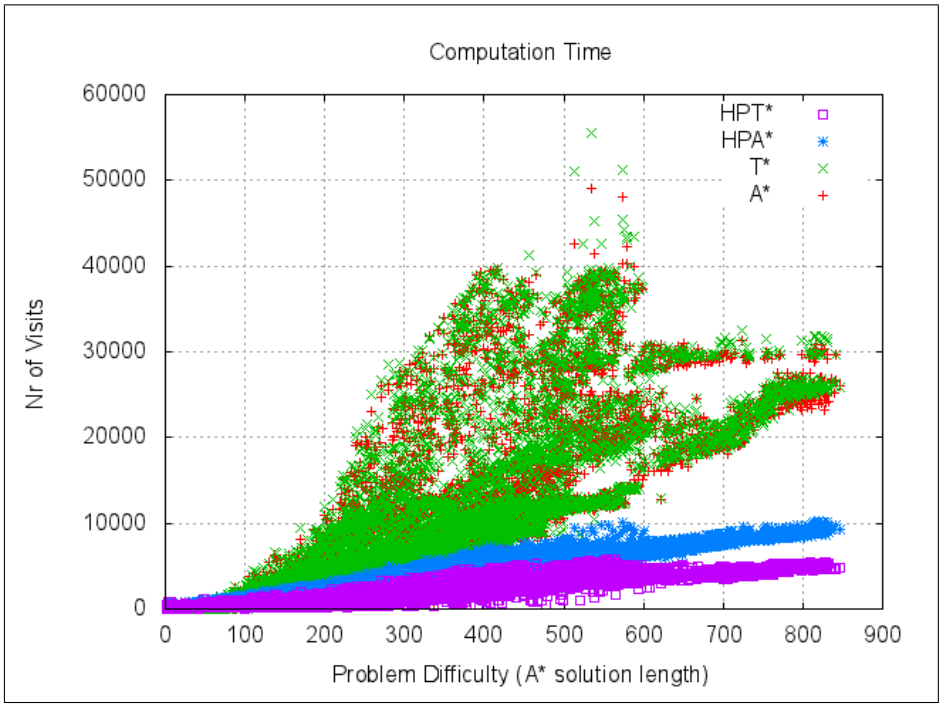


Figure 5: Number of visits done by each algorithm.

Algorithm	Nr of visits	Relative to HPA*
A*	8638.2	2.6091
Theta*	8825.1	2.6655
HPA*	3310.8	1
HPT*	1752.9	0.52944

Table 3: Average number of visits done by each algorithm.

5.3 Nodes in Memory

As can be expected, the memory usage shows a strong relation to the number of nodes visited. A high number of nodes visited, results in a high number of nodes in memory. The shapes of the graphs in figure 6 look very similar to those in figure 5. Again, A* and Theta* break the 10000 node line at around an A* path length of 200. HPT* is more similar in to HPA* with memory usage, than it is in node visits. As table 4 shows, HPT* only needs to store about 7% less nodes in memory than HPA*. The difference between the hierarchical algorithms and the non-hierarchical algorithms is greater with memory usage, than with node visits.

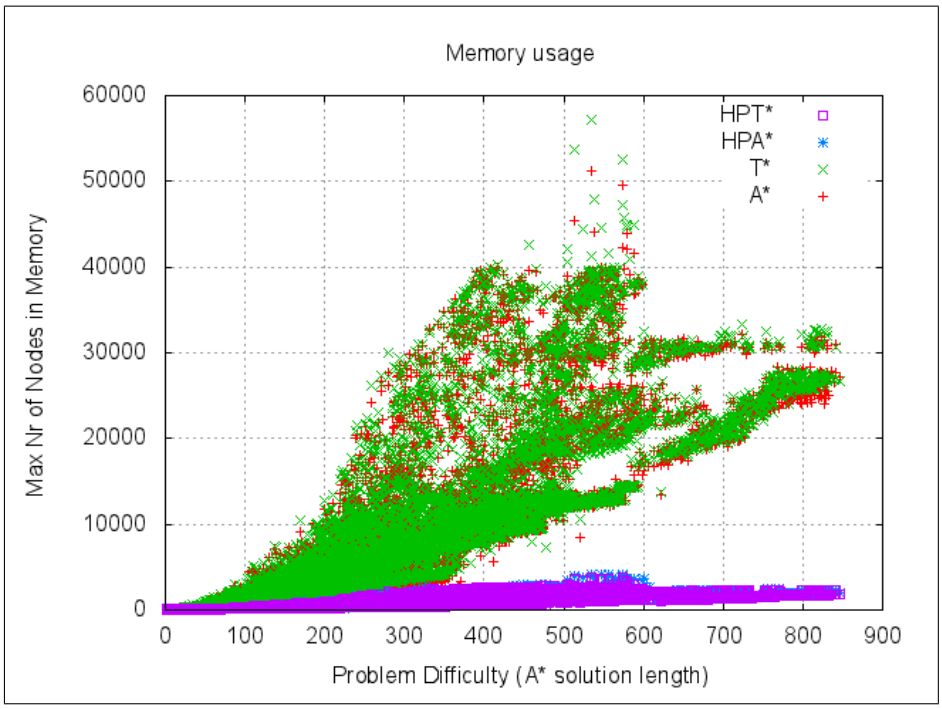


Figure 6: Maximum of nodes in memory with each algorithm.

Algorithm	Nodes in memory	Relative to HPA*
A*	9378.3	11.470
Theta*	9462.9	11.573
HPA*	817.64	1
HPT*	760.18	0.92972

Table 4: Average maximum of nodes in memory for each algorithm.

6 Conclusion

In section 3, we established our research question:

Does *Hierarchical Path-Finding Theta* succesfully combine* the *benefits of Theta** with the *benefits of HPA**?

We also imposed the following requirements on the HPT*, to call it a successful combination:

1. HPT* produces paths that are not more than 10% longer, on average, when compared to Theta*.
2. HPT* requires not more than 10% more node visits, on average, to compute paths when compared to HPA*.
3. HPT* requires not more than 10% more memory, on average to, compute paths when compared to HPA*.

Test results showed us that HPT* paths are only 4.34% longer than Theta* paths on average. The number of visits required by HPT* turned out to be 52.9% of the visits required by HPA*. This can be explained by the path refinement process of HPT* in comparison to the path refinement process of HPA*. HPA* needs to run A* for every subsequent point in the abstract solution. HPT* skips the refinement process for points in the abstract solution that have line of sight with eachother. On large open maps, where distant nodes often have line of sight with eachother, HPT* will be able to skip a large ammount of visits, that would otherwise be required. Since the memory required by HPT* is also 7% less than with HPA*, we can conclude that HPT* indeed successfully combines the benefits of both Theta* and HPA*.

6.1 Discussion

As noticed in the results section. We noticed HPA* outperforming A* in some benchmarked problems. This should be impossible since A* produces the optimal results on the grid. This can only be explained as being the result of a fault in the code we used to do our benchmarks. We have not been able to find the cause for these strange results, partly because of the late stage in which the data errors were discovered. This does ofcourse impact the reliability of our test results and thus conclussions.

However, we can argue that it is unlikely that the fault has had a drastic effect on our results. If the fault only produced incorrect path lengths, while HPA* still produced correct paths, then our analysis is still valid, since we didn't need to compare to HPA* path lengths in our conclusion. Even with the fault, HPA* still seems to behave as expected on most cases and produce a path that is on average 4.7% longer than the equivalent A* path. In the original HPA* paper (Botea et al., 2004), HPA* produces paths that are 4% to 6% longer than A*, for paths with a length larger than 100, before they apply path smoothing. So it seems likely that, with the large ammount of paths benchmarked, the effect of the programming error diminished.

6.2 Future Work

It would be interesting to change the HPT* algorithm, to use a variant of Theta* and see how this effects the search results. Two candidates for this are AP-Theta* (instead of Basic Theta*)(Nash et al., 2007) or the newer variant, Lazy Theta*(Nash et al., 2010).

It would also be interesting to see how well HPT* would work in highly dynamic maps, where paths can become invallid over time. HPT* like HPA*, is able to postpone the path refinement of sections of the abstract solution. However, since HPT* simplifies the abstract path by removing nodes from it, HPT* would have less fine grained control over which parts of the solution to refine.

7 Bibliography

A. Botea, M. Müller, and J. Schaeffer. Near optimal hierarchical path-finding. *Journal of game development*, 2004.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions of System Science and Cybernetics*, SCC-4(2):100–107, 1968.

A. Nash, K. Daniel, S. Koenig, and A. Felner. Theta*: Any-angle path planning on grids. *the AAAI Conference on Artificial Intelligence*, pages 1177–1183, 2007.

Alex Nash, Sven Koenig, and Craig Tovey. Lazy theta*: Any-angle path planning and path length analysis in 3d. In *Third Annual Symposium on Combinatorial Search*, 2010.

Stuart Russel and Peter Norvig. *Artificial intelligence : a modern approach*. Prentice Hall/Pearson Education, Upper Saddle River, N.J, second edition, 2003. ISBN 0130803022.

Nathan Sturtevant. Nathan sturtevant's moving ai lab: Pathfinding benchmarks, April 2011. URL `http://www.aiide.org/benchmarks/`.

Peter Yap. Grid-based path-finding. In Robin Cohen and Bruce Spencer, editors, *Advances in Artificial Intelligence*, volume 2338 of *Lecture Notes in Computer Science*, pages 44–55. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43724-6. doi: 10.1007/3-540-47922-8_4. URL `http://dx.doi.org/10.1007/3-540-47922-8_4`.

A A* and Theta* pseudocode

The following pseudocode describes the A* and the Theta* algorithm.

```
1  // A* algorithm:
2  // Returns a solution (path) to the problem if a solution exists,
3  // returns null otherwise.
4  Solution aStar(Problem problem)
5  {
6    fringe     = empty priority queue;
7    closedList = empty list;
8
9    solution = null;
10   initialStateNode = createNode(problem.initialState());
11
12   fringe.push(initialStateNode);
13
14   while( solution == null && fringe.size() > 0 )
15   {
16     currentNode = fringe.pop();
17     if( problem.isGoal(currentNode.state) )
18     {
19       solution = toSolution(currentNode);
20     }
21     else
22     {
23       closedList.push(currentNode);
24       successors = expand(currentNode, problem);
25
26       foreach node in successors
27       {
28         //For A*: this line of code is skipped.
29         //For Theta*: do a line of sight check with grandparent
30         //to see if it is possible to skip the parent.
31         skipParentIfPossible(node, problem);
32
33         //check open and closed list to see if node can be placed into fringe
34         replaceIfPossible(node);
35       }
36     }
37   }
38
39   return solution;
40 }
```

B HPA* and HPT* pseudocode

The following pseudocode describes both HPA* and HPT*. For A* the searchAlgorithm parameter would always be an instance of A*. For Theta* the searchAlgorithm parameter would always be an instance of Theta*.

Note: Trivial paths are straight line paths that are possible when nodes have line of sight.

```
AbstractGraph buildAbstractGraph( GridGraph concreteGraph , int clusterSize,
                                  SearchAlgorithm searchAlgorithm
                                )
{
  AbstractGraph abstractGraph = empty graph;

  //divide the concrete graph into square clusters
  int clusterArrayWidth  = ceiling(concreteGraph.width / clusterSize);
  int clusterArrayHeight = ceiling(concreteGraph.width / clusterSize);
  abstractGraph.clusters =
    Cluster[clusterArrayWidth][clusterArrayHeight];

  //connect neighboring clusters via entrances
  buildNodes(abstractGraph, searchAlgorithm);

  //connect nodes in same cluster
  buildIntraEdges(abstractGraph, searchAlgorithm);

  return hMap;
}
```

```
void buildNodes(AbstractGraph abstractGraph, SearchAlgorithm searchAlgorithm )
{
  foreach (A,B) in (abstractGraph.clusters x abstractGraph.clusters)
  {
    if(A neighbors B) {
      //Maximal obstacle free entrances on border
      Entrance[] entrances = findEntrances(A,B);

      for each entrance in entrances
      {
        //either two nodes or a square of nodes
        //connecting A to B.
        Node[] nodes = toNodes(entrance);

        //Connect all nodes to eachother,
        /with path cost as edge costs.
        //For Theta*, also mark trivial paths.
        connectAll(nodes, searchAlgorithm);

        abstractGraph.add(nodes);
      }
    }
  }
}
```

```
1  void buildIntraEdges( AbstractGraph abstractGraph,
2                        SearchAlgorithm searchAlgorithm
3                      )
4  {
5    foreach Cluster cluster in abstractGraph.clusters
6    {
7      foreach (A,B) in (abstractGraph.clusters.nodes x abstractGraph.clusters.nodes)
8      {
9        //If a path from A to B exists within the boundaries of the cluster
10       //add an edge with the path cost as edge cost.
11       //For Theta* also mark trivial paths.
12       connect(A,B, searchAlgorithm);
13     }
14   }
15 }
```

```
1  void hierarchicalSearch( Node start, node goal,
2                           AbstractGraph abstractGraph,
3                           SearchAlgorithm searchAlgorithm
4                         )
5  {
6    //insert start and goal into correct cluster and connect to all other nodes in
         same cluster
7    abstractGraph.insertAndConnect(start, searchAlgorithm);
8    abstractGraph.insertAndConnect(goal, searchAlgorithm);
9
10   Solution abstractSolution = searchAlgorithm(start, goal, abstractGraph);
11
12   //clean up
13   abstractGraph.remove(start);
14   abstractGraph.remove(goal);
15 }
```

```
1   void refinePath( Solution abstractSolution,
2                    int fromIndex, int toIndex,
3                    SearchAlgorithm searchAlgorithm
4                  )
5  {
6    Node start = abstractSolution.at(fromIndex);
7
8    Solution refinement = start;
9
10   for(int previous = fromIndex, next = fromIndex + 1; next <= toIndex; next++,
         previous++)
11   {
12     //Only abstract solutions from Theta* have trivial paths
13     if( trivial(abstractSolution.at(next) ) )
14     {
15       refinement.addToPath(abstractSolution.next());
16     }
17     else {
18       Solution subRefinement = searchAlgorithm(abstractSolution.at(previous),
             abstractSolution.at(next));
19       refinement.addToPath(subRefinement);
20     }
21   }
22 }
```