**TABLE OF CONTENT:**

1. **Declaration------------------------------------------------------------2**
2. **Acknowledgement---------------------------------------------------3**
3. **Abstract---------------------------------------------------------------4**
4. **LITERATURE REVIEW…………………………………………...8**
5. **INTRODUCTION ……………………………………………..….13**
6. **INTRODUCTION TO A\*……………………………………………..…...23**
7. **OPTIMAL HIERARICHICAL PATH FINDING….………..….…….....,31**
8. **EXPERIMENTAL RESULTS AND ANALYSIS ……………..…….....…34**
9. **ALGO IMPLEMENTED…….………………………...……..........36**
10. **IMPLEMENTATION RESULT FIGURES & PROJECT SCREENS…40**
11. **REFERENCES……………………………………………………………..45**
12. **FUTURE WORK & SCOPE……………………………………………….47**

***DECLARATION***

***This is to verify that the following students ( SAKSHI RAI, ARUN SHARMA and SUSHANT KUMAR ) of 4th year (7th semester) CSE department have completed their minor project based on HPA\* successfully under the guidance of Mrs. SAKSHI HOODA of CSE department.***

***I, thereby declare the successful completion of the project.***

( )

Signature

**Mrs. Poonam Bansal** HOD CSE department

###### *ACKNOWLEDGEMENT*

###### \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Apart from the efforts of me, the success of any project depends largely on the encouragement and guidelines of many others. I take this opportunity to express my gratitude to the people who have been instrumental in the successful completion of this project. I would like to show my greatest appreciation to **Mrs. Sakshi Hooda**. I can’t say thank you enough for his tremendous support and help. I feel motivated and encouraged every time I attend his meeting. Without his encouragement and guidance this project would not have materialized.

The guidance and support received from all the members who contributed and who are contributing to this project, was vital for the success of the project. I am grateful for their constant support and help.

***ABSTRACT***

***This project is undertaken basically inorder to find the optimal path calculation. In***

***video games path finding must be done quickly and accurately. Not much***

***computational time is allowed for pathfinding , but realistic looking paths are required. One approach to path finding which attempts to satisfy both of these constraints is to perform pathfinding on abstractions of the Map. Hierarchical Pathfinding A\*(HPA\*) does this by dividing the map into square sectors and defining entrances between them.***

**Literature Review**

The first part of this section summarizes hierarchical approaches used for path-finding in commercial games. The second part reviews related work in a more general context, including applications to other grid domains such as robotics.

Path-finding using a two-level hierarchy is described in. The author provides only a high-level presentation of the approach. The problem map is abstracted into clusters such as rooms in a building or square blocks on a field. An abstract action crosses a room from the middle of an entrance to another. This method has similarities to our work. First, both approaches partition the problem map into clusters such as square blocks. Second, abstract actions are block Crossings (as opposed to going from one block center to another block center). Third, both techniques abstract a block entrance into one transition point (in fact, we allow either one or two points). This leads to fast computation but gives up the solution optimality. There are also significant differences between the two approaches. We extend our hierarchy to several abstraction levels and do this abstraction in a domain independent way. We also pre-compute and cache optimal distances for block crossing, reducing the costs of the on-line computation.

Another important hierarchical approach for path-finding in commercial games uses *points of visibility*. This method exploits the domain local topology to define an abstract graph that covers the map efficiently. The graph nodes represent the corners of convex obstacles. For each node, edges are added to all the nodes that can be seen from the current node (i.e., the can be connected with a straight line).

This method provides solutions of good quality. It is particularly useful when the number of obstacles is relatively small and they have a convex polygonal shape (i.e., building interiors). The efficiency of the method decreases when many obstacles are present and/or their shape is not a convex polygon. Consider the case of a map containing a forest, which is a dense collection of small size obstacles. Modeling such a topology with points of visibility would result in a large graph (in terms of number of nodes and edges) with short edges. Therefore, the key idea of traveling long distances in a single step wouldn't be efficiently exploited. When the problem map contains concave or curved shapes, the method either has poor performance or needs sophisticated engineering to build the graph efficiently. In fact, the need for algorithmic or designer assistance to create the graph is one of the disadvantages of the method. In contrast, our approach works for any kinds of maps and does not require complex domain analysis to perform the abstraction. The *navigation meshes* (aka. NavMesh) is a powerful abstraction technique useful for 2D and 3D maps. In a 2D environment, this approach covers the unblocked area of a map with a (minimal) set of convex polygons. A method for building a near optimal NavMesh is presented in [11]. This method relaxes the condition of the minimal set of polygons and builds a map coverage much faster.

Besides commercial computer games, path-finding has applications in many research areas. Path-finding approaches based on topological abstraction that have been explored in robotics domains are especially relevant for the work described.

*Quadtrees*  have been proposed as a way of doing hierarchical map decomposition. This method partitions a map into square blocks with different sizes so that a block contains either only walkable cells or only blocked cells. The problem map is initially partitioned into 4 blocks. If a block contains both obstacle cells and walkable cells, then it is further decomposed into 4 smaller blocks, and so on. An action in this abstracted framework is to travel between the centers of two adjacent blocks. Since the agent always goes to the middle of a box, this method produces sub-optimal solutions.

To improve the solution quality, quadtrees can be extended to *framed quadtrees* [1, 12]. In framed quadtrees, the border of a block is augmented with cells at the highest resolution. An action crosses a block between any two border cells. Since this representation permits many angles of direction, the solution quality improves significantly. On the other hand, framed quadtrees use more memory

than quadtrees. Framed quadtrees are more similar to our work than quadtrees, since we also use block crossings as abstract actions. However, we don't consider *all* the cells on the block border as entrance points. We reduce the number of block entrance points by abstracting an entrance into one or two such points. Moreover, our approach allows blocks to contain obstacles. This means that the distance between two transition points is not necessarily linear. For this reason we have to compute optimal paths between entrance points placed on the border of the same block. A multi-level hierarchy has been used to enhance the performance of multiple goal path-planning in a MDP (Markov Decision Process) framework [4]. The problem posed is to efficiently learn near optimal policies \_\_(x; y) to travel from x to y for all pairs (x; y) of map locations. The number of policies that have to be computed and stored is quadratic in the number of map cells. To improve both the

memory and time requirements (for the price of losing optimality), a multi-level structure is used. a so called *airport hierarchy*. All locations on the problem map are *airports* that are assigned to different hierarchical levels. The strategy for traveling from x to y is similar to traveling by plane in the real world. First travel to bigger and bigger airports until we reach an airport that is big enough to have a connection to the area that contains the destination. Second, go down in the hierarchy by traveling to smaller airports until the destination is reached. This approach is very similar to the strategy outlined in.

An analysis of the nature of path-finding in various frameworks is performed in . The authors classify path-finding problems based on the type of the results that are sought, the environment type, the amount of information available, etc. Challenges specific to each problem type and solving strategies such as re planning and using dynamic data structures are briefly discussed. A hierarchical approach for shortest path algorithms that has similarities with HPA\* is analyzed in [9]. This work decomposes an initial problem graph into a set of fragment sub-graphs and a global boundary sub-graph that links the fragment sub-graphs. Shortest paths are computed and cached for future use, similarly to the caching that HPA\* performs for cluster traversal routes. The authors analyze what shortest paths (i.e., from which sub-graphs) to cache, and what information to keep (i.e., either complete path or only cost) for best performance when limited memory is available.

Another technique related to HPA\* is Hierarchical A\* [2], which also uses hierarchical representations of a space with the goal of reducing the overall search effort. However, the way that hierarchical representations are used is different in these two techniques. While our approach uses abstraction to structure and enhance the representation of the search space, Hierarchical A\* is a method for automatically generating domain-independent heuristic state evaluations. In single agent search, a heuristic function that evaluates the distance from a state to the goal is used to guide the search process. The quality of such a function greatly affects the quality of the whole search algorithm. Starting from the initial space,

Hierarchical A\* builds a hierarchy of abstract spaces until an abstract one-state space is obtained. When building the next abstract space, several states of the current space are grouped to form one abstract state in the next space. In this hierarchy, an abstract space is used to compute a heuristic function for the previous space.

**INTRODUCTION**

The problem of path-finding in commercial computer games has to be solved in

real time, often under constraints of limited memory and CPU resources.

Hierarchical search is acknowledged as an effective approach to reduce the complexity of this problem. However, no detailed study of hierarchical path-finding in commercial games has been published. Part of the explanation is that game companies usually do not make their ideas and source code available.

The industry standard is to use A\* [10] or iterative-deepening A\*, IDA\* [3].

A\* is generally faster, but IDA\* uses less memory. There are numerous enhancements to these algorithms to make them run faster or explore a smaller search tree. For many applications, especially those with multiple moving NPCs (such as in real-time strategy games), these time and/or space requirements are limiting factors.

In this paper we describe HPA\*, a new method for hierarchical path-finding on grid-based maps, and present performance tests. Our technique abstracts a map Into linked local clusters. At the local level, the optimal distances for crossing the Clusters are pre-computed and cached. At the global level, an action is to cross a Cluster in a single step rather than moving to an adjacent atomic location.

Our method is simple, easy to implement, and generic, as we use no application specification knowledge and apply the technique independently of the map properties. We handle variable cost terrains and various topology types such as forests, open areas with obstacles of any shape, or building interiors. without any implementation changes.

For many real-time path-finding applications, the complete path is not needed Knowing the first few moves of a valid path often suffices, allowing a mobile unit to start moving in the right direction. Subsequent events may result in the unit having to change its plan, obviating the need for the rest of the path. A\* returns a complete path. In contrast, HPA\* returns a complete path of sub-problems. The first sub-problem can be solved, giving a unit the first few moves along the path.

As needed, subsequent sub-problems can be solved providing additional moves. The advantage here is that if the unit has to change its plan, then no effort has been wasted on computing a path to a goal node that was never needed.

The hierarchical framework is suitable for static and dynamically changing Environments. In the latter case, first assume that local changes can occur on immobile topology elements (e.g., a bomb destroys a bridge). We recompute the information extracted from the modified cluster locally and keep the rest of the framework unchanged. Second, assume that there are many mobile units on the

map and a computed path can become blocked by another unit. We compute an abstract path with reduced effort and do not spend additional effort to refine it to the low-level representation. We quickly get the character moving in a proven good direction and refine parts of the abstract path as the character needs them. If the path becomes blocked, we replan for another abstract path from the current position of the character.

The hierarchy of our method can have any number of levels, making it scalable for large problem spaces. When the problem map is large, a larger number of levels can be the answer for reducing the search effort, for the price of more storage and pre-processing time. Our technique produces sub-optimal solutions, trading optimality for improved execution performance. After applying a path-smoothing procedure, our solutions are within 1% of optimal.

**Motivation**

Consider the problem of traveling by car from Los Angeles, California, to Toronto, Ontario. Specifically, what is the minimum distance to travel by car from 1234 Santa Monica Blvd in Los Angles to 4321 Yonge Street in Toronto? Given a detailed roadmap of North America, showing *all* roads annotated with driving distances, an A\* implementation can compute the optimal (minimum distance) travel route. This might be an expensive computation, given the sheer size of the

roadmap.

Of course, a human travel planner would never work at such a low level of

detail. They would solve three problems:

1. Travel from 1234 Santa Monica Boulevard to a major highway leading out

of Los Angeles.

2. Plan Plan a route from Los Angeles to Toronto.

3. Travel from the incoming highway in Toronto to 4321 Yonge Street.

The first and third steps would require a detailed roadmap of each city. Step (2) could be done with a high-level map, with roads connecting cities, abstracting away all the detail within the city. In effect, the human travel planner uses abstraction to quickly find a route from Los Angles to Toronto. However, by treating cities as black boxes, this search is not guaranteed to find the shortest route. For example, although it may be faster to stay on a highway, for some cities where the highway goes around the city, leaving the highway and going through the city might be a shorter route. Of course, it may not be a faster route (city speeds are slower than highway speeds), but in this example we are trying to minimize travel distance.

Abstraction could be taken to a higher level: do the planning at the state/province

level. Once the path reaches a state boundary, compute the best route from state

to state. Once you know your entrances and exits from the states, then plan the

inter-state routes. Again, this will work but may result in a sub-optimal solution.

Taken to the extreme, the abstraction could be at the country level: travel from

the United States to Canada. Clearly, there comes a point where the abstraction

becomes so coarse as to be effectively useless.

We want to adopt a similar abstraction strategy for computer game path-finding.

We could use A\* on a complete 1000\_1000 map . but that represents a potentially huge search space. Abstraction can be used to reduce this dramatically. Consider each 10 \_ 10 block of the map as being a .city.. Now we can search in a map of 100 \_ 100 cities. For each city, we know the city entrances and the costs of crossing the city for all the entrance pairs. We also know how to travel between cities. The problem then reduces to three steps:

1. Start node: Within the block containing the start node, find the optimal path

to the borders of the block.

2. Search at the block level (100 \_ 100 blocks) for the optimal path from the

block containing the start node to the block containing the goal node

.

3. Goal node: Within the block containing the goal node, find the optimal path

from the border of the block to the goal.

The result is a much faster search giving nearly optimal solutions. Further, the abstraction is topology independent; there is no need for a level designer to manually break the grid into high-level features or annotate it with way-points.

**Contributions**

The contributions of this paper include:

1. HPA\*, a new hierarchical path-finding algorithm (including pseudo-code

and source code) that is domain-independent and works well for static and

dynamic terrain topology

2. Experimental results for hierarchical search on a variety of games mazes

(from BioWare's BALDUR'S GATE), showing up to a 10-fold speed improvement

in exchange for a 1% degradation in path quality.

3. Variations on the hierarchical search idea appear to be in use by several

game companies, although most of their algorithmic details are not public.

To the best of our knowledge, this is the first scientific study of using

hierarchical A\* in the domain of commercial computer games.

Section 2 contains a brief overview of the background literature. Section 3

presents our new approach to hierarchical A\*, and its performance is evaluated

in Section 4. Section 5 presents our conclusions and topics for further research.

Appendix A provides the pseudo-code for our algorithm.

**A\* Running Time.**

Assuming square sectors, let L ﾗ Lbe the sector size and E the number of entrances. Then E ≤ 4L − 4, because there are that many nodes located on the edges of the sector. There are 􀀀E 2 \_ pairs of entrances, which is O(L2). In the worst case A\* expands all L2 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(L2)) ⊆ O(log L). Therefore, the worst-case total running time of A\* to determine distances between each entrance pairs is O(E2L2 log L) ⊆ O(L4 log L).

**Running Time of Dijkstra’s Algorithm.**

Again, let Lﾗ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 L2) ⊆ O(log L). For every vertex that is removing 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(EL2 log L) O(L3 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 \_(L2) in the sector interior connecting the north .

**Introduction to A\* algorithm**

The A\* (pronounced A-star) algorithm can be complicated for beginners. While there are many articles on the web that explain A\*, most are written for people who understand the basics already. This article is for the true beginner.

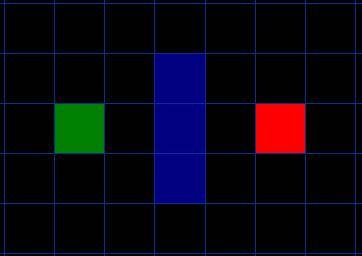
This article does not try to be the definitive work on the subject. Instead it describes the fundamentals and prepares you to go out and read all of those other materials and understand what they are talking about. Links to some of the best are provided at the end of this article, under Further Reading.

Finally, this article is not program-specific. You should be able to adapt what's here to any computer language. As you might expect, however, I have included a link to a sample program at the end of this article. The sample package contains two versions: one in C++ and one in Blitz Basic. It also contains executables if you just want to see A\* in action.

But we are getting ahead of ourselves. Let's start at the beginning ...

**Introduction: The Search Area**

Let’s assume that we have someone who wants to get from point A to point B. Let’s assume that a wall separates the two points. This is illustrated below, with green being the starting point A, and red being the ending point B, and the blue filled squares being the wall in between.

   
[Figure 1]

The first thing you should notice is that we have divided our search area into a square grid. Simplifying the search area, as we have done here, is the first step in pathfinding. This particular method reduces our search area to a simple two dimensional array. Each item in the array represents one of the squares on the grid, and its status is recorded as walkable or unwalkable. The path is found by figuring out which squares we should take to get from A to B. Once the path is found, our person moves from the center of one square to the center of the next until the target is reached.

These center points are called “nodes”. When you read about pathfinding elsewhere, you will often see people discussing nodes. Why not just call them squares? Because it is possible to divide up your pathfinding area into something other than squares. They could be rectangles, hexagons, triangles, or any shape, really. And the nodes could be placed anywhere within the shapes – in the center or along the edges, or anywhere else. We are using this system, however, because it is the simplest.

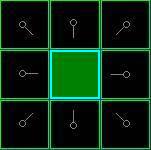
**Starting the Search**

Once we have simplified our search area into a manageable number of nodes, as we have done with the grid layout above, the next step is to conduct a search to find the shortest path. We do this by starting at point A, checking the adjacent squares, and generally searching outward until we find our target.

We begin the search by doing the following:

1. Begin at the starting point A and add it to an “open list” of squares to be considered. The open list is kind of like a shopping list. Right now there is just one item on the list, but we will have more later. It contains squares that might fall along the path you want to take, but maybe not. Basically, this is a list of squares that need to be checked out.
2. Look at all the reachable or walkable squares adjacent to the starting point, ignoring squares with walls, water, or other illegal terrain. Add them to the open list, too. For each of these squares, save point A as its “parent square”. This parent square stuff is important when we want to trace our path. It will be explained more later.
3. Drop the starting square A from your open list, and add it to a “closed list” of squares that you don’t need to look at again for now.

At this point, you should have something like the following illustration. In this illustration, the dark green square in the center is your starting square. It is outlined in light blue to indicate that the square has been added to the closed list. All of the adjacent squares are now on the open list of squares to be checked, and they are outlined in light green. Each has a gray pointer that points back to its parent, which is the starting square.

  
  [Figure 2]

Next, we choose one of the adjacent squares on the open list and more or less repeat the earlier process, as described below. But which square do we choose? The one with the lowest F cost.

**Path Scoring**

The key to determining which squares to use when figuring out the path is the following equation:

F = G + H

where

* G = the movement cost to move from the starting point A to a given square on the grid, following the path generated to get there.
* H = the estimated movement cost to move from that given square on the grid to the final destination, point B. This is often referred to as the heuristic, which can be a bit confusing. The reason why it is called that is because it is a guess. We really don’t know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). You are given one way to calculate H in this tutorial, but there are many others that you can find in other articles on the web.

Our path is generated by repeatedly going through our open list and choosing the square with the lowest F score. This process will be described in more detail a bit further in the article. First let’s look more closely at how we calculate the equation.

As described above, G is the movement cost to move from the starting point to the given square using the path generated to get there. In this example, we will assign a cost of 10 to each horizontal or vertical square moved, and a cost of 14 for a diagonal move. We use these numbers because the actual distance to move diagonally is the square root of 2 (don’t be scared), or roughly 1.414 times the cost of moving horizontally or vertically. We use 10 and 14 for simplicity’s sake. The ratio is about right, and we avoid having to calculate square roots and we avoid decimals. This isn’t just because we are dumb and don’t like math. Using whole numbers like these is a lot faster for the computer, too. As you will soon find out, pathfinding can be very slow if you don’t use short cuts like these.

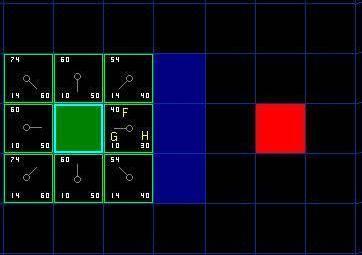
Since we are calculating the G cost along a specific path to a given square, the way to figure out the G cost of that square is to take the G cost of its parent, and then add 10 or 14 depending on whether it is diagonal or orthogonal (non-diagonal) from that parent square. The need for this method will become apparent a little further on in this example, as we get more than one square away from the starting square.

H can be estimated in a variety of ways. The method we use here is called the Manhattan method, where you calculate the total number of squares moved horizontally and vertically to reach the target square from the current square, ignoring diagonal movement, and ignoring any obstacles that may be in the way. We then multiply the total by 10, our cost for moving one square horizontally or vertically. This is (probably) called the Manhattan method because it is like calculating the number of city blocks from one place to another, where you can’t cut across the block diagonally.

Reading this description, you might guess that the heuristic is merely a rough estimate of the remaining distance between the current square and the target "as the crow flies." This isn't the case. We are actually trying to estimate the remaining distance along the path (which is usually farther). The closer our estimate is to the actual remaining distance, the faster the algorithm will be. If we overestimate this distance, however, it is not guaranteed to give us the shortest path. In such cases, we have what is called an "inadmissible heuristic."

Technically, in this example, the Manhattan method is inadmissible because it slightly overestimates the remaining distance. But we will use it anyway because it is a lot easier to understand for our purposes, and because it is only a slight overestimation. On the rare occasion when the resulting path is not the shortest possible, it will be nearly as short. Want to know more? You can find equations and additional notes on heuristics [here](http://www.policyalmanac.org/games/heuristics.htm).

F is calculated by adding G and H. The results of the first step in our search can be seen in the illustration below. The F, G, and H scores are written in each square. As is indicated in the square to the immediate right of the starting square, F is printed in the top left, G is printed in the bottom left, and H is printed in the bottom right.

  
  [Figure 3]

So let’s look at some of these squares. In the square with the letters in it, G = 10. This is because it is just one square from the starting square in a horizontal direction. The squares immediately above, below, and to the left of the starting square all have the same G score of 10. The diagonal squares have G scores of 14.

The H scores are calculated by estimating the Manhattan distance to the red target square, moving only horizontally and vertically and ignoring the wall that is in the way. Using this method, the square to the immediate right of the start is 3 squares from the red square, for a H score of 30. The square just above this square is 4 squares away (remember, only move horizontally and vertically) for an H score of 40. You can probably see how the H scores are calculated for the other squares.

The F score for each square, again, is simply calculated by adding G and H together.

# Continuing the Search

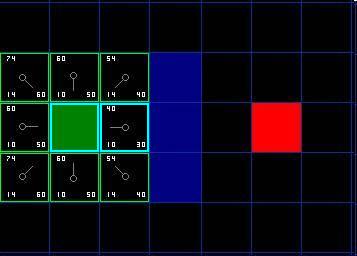
To continue the search, we simply choose the lowest F score square from all those that are on the open list. We then do the following with the selected square:

4) Drop it from the open list and add it to the closed list.

5) Check all of the adjacent squares. Ignoring those that are on the closed list or unwalkable (terrain with walls, water, or other illegal terrain), add squares to the open list if they are not on the open list already. Make the selected square the “parent” of the new squares.

6) If an adjacent square is already on the open list, check to see if this path to that square is a better one. In other words, check to see if the G score for that square is lower if we use the current square to get there. If not, don’t do anything.   
    On the other hand, if the G cost of the new path is lower, change the parent of the adjacent square to the selected square (in the diagram above, change the direction of the pointer to point at the selected square). Finally, recalculate both the F and G scores of that square. If this seems confusing, you will see it illustrated below.

Okay, so let’s see how this works. Of our initial 9 squares, we have 8 left on the open list after the starting square was switched to the closed list.  Of these, the one with the lowest F cost is the one to the immediate right of the starting square, with an F score of 40. So we select this square as our next square. It is highlight in blue in the following illustration.

   
[Figure 4]

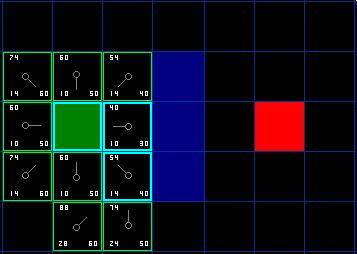
First, we drop it from our open list and add it to our closed list (that’s why it’s now highlighted in blue).  Then we check the adjacent squares. Well, the ones to the immediate right of this square are wall squares, so we ignore those. The one to the immediate left is the starting square. That’s on the closed list, so we ignore that, too.

The other four squares are already on the open list, so we need to check if the paths to those squares are any better using this square to get there, using G scores as our point of reference. Let’s look at the square right above our selected square. Its current G score is 14. If we instead went through the current square to get there, the G score would be equal to 20 (10, which is the G score to get to the current square, plus 10 more to go vertically to the one just above it). A G score of 20 is higher than 14, so this is not a better path. That should make sense if you look at the diagram. It’s more direct to get to that square from the starting square by simply moving one square diagonally to get there, rather than moving horizontally one square, and then vertically one square.

When we repeat this process for all 4 of the adjacent squares already on the open list, we find that none of the paths are improved by going through the current square, so we don’t change anything. So now that we looked at all of the adjacent squares, we are done with this square, and ready to move to the next square.

So we go through the list of squares on our open list, which is now down to 7 squares, and we pick the one with the lowest F cost. Interestingly, in this case, there are two squares with a score of 54. So which do we choose? It doesn’t really matter. For the purposes of speed, it can be faster to choose the last one you added to the open list. This biases the search in favor of squares that get found later on in the search, when you have gotten closer to the target. But it doesn’t really matter. (Differing treatment of ties is why two versions of A\* may find different paths of equal length.)

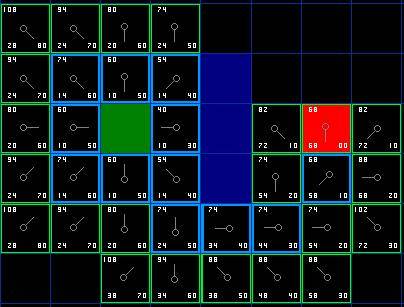
So let’s choose the one just below, and to the right of the starting square, as is shown in the following illustration.

   
[Figure 5]

This time, when we check the adjacent squares we find that the one to the immediate right is a wall square, so we ignore that. The same goes for the one just above that. We also ignore the square just below the wall. Why? Because you can’t get to that square directly from the current square without cutting across the corner of the nearby wall. You really need to go down first and then move over to that square, moving around the corner in the process. (Note: This rule on cutting corners is optional. Its use depends on how your nodes are placed.)

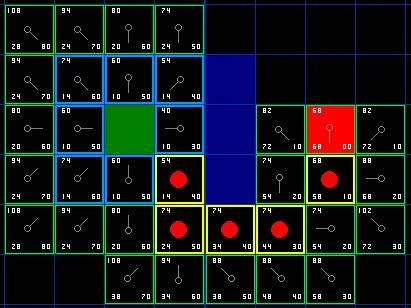
That leaves five other squares.  The other two squares below the current square aren’t already on the open list, so we add them and the current square becomes their parent. Of the other three squares, two are already on the closed list (the starting square, and the one just above the current square, both highlighted in blue in the diagram), so we ignore them. And the last square, to the immediate left of the current square, is checked to see if the G score is any lower if you go through the current square to get there. No dice. So we’re done and ready to check the next square on our open list.

We repeat this process until we add the target square to the closed list, at which point it looks something like the illustration below.

   
[Figure 6]

Note that the parent square for the square two squares below the starting square has changed from the previous illustration. Before it had a G score of 28 and pointed back to the square above it and to the right. Now it has a score of 20 and points to the square just above it. This happened somewhere along the way on our search, where the G score was checked and it turned out to be lower using a new path – so the parent was switched and the G and F scores were recalculated. While this change doesn’t seem too important in this example, there are plenty of possible situations where this constant checking will make all the difference in determining the best path to your target.

So how do we determine the path? Simple, just start at the red target square, and work backwards moving from one square to its parent, following the arrows. This will eventually take you back to the starting square, and that’s your path. It should look like the following illustration. Moving from the starting square A to the destination square B is simply a matter of moving from the center of each square (the node) to the center of the next square on the path, until you reach the target.

   
[Figure 7]

# Summary of the A\* Method

Okay, now that you have gone through the explanation, let’s lay out the step-by-step method all in one place:

1) Add the starting square (or node) to the open list.

2) Repeat the following:

a) Look for the lowest F cost square on the open list. We refer to this as the current square.

b) Switch it to the closed list.

c) For each of the 8 squares adjacent to this current square …

* If it is not walkable or if it is on the closed list, ignore it. Otherwise do the following.
* If it isn’t on the open list, add it to the open list. Make the current square the parent of this square. Record the F, G, and H costs of the square.
* If it is on the open list already, check to see if this path to that square is better, using G cost as the measure. A lower G cost means that this is a better path. If so, change the parent of the square to the current square, and recalculate the G and F scores of the square. If you are keeping your open list sorted by F score, you may need to resort the list to account for the change.

d) Stop when you:

* Add the target square to the closed list, in which case the path has been found (see note below), or
* Fail to find the target square, and the open list is empty. In this case, there is no path.

3) Save the path. Working backwards from the target square, go from each square to its parent square until you reach the starting square. That is your path.

Note: In earlier versions of this article, it was suggested that you can stop when the target square (or node) has been added to the open list, rather than the closed list.  Doing this will be faster and it will almost always give you the shortest path, but not always.  Situations where doing this could make a difference are when the movement cost to move from the second to the last node to the last (target) node can vary significantly -- as in the case of a river crossing between two nodes, for example.

**Hierarchical Path-finding**

Our hierarchical approach implements the strategy described in Section 1.1. Searching for an abstract solution in our hierarchical framework is a three step process called on-line search. First, travel to the border of the neighborhood that contains the start location. Second, search for a path from the border of the start neighbor- hood to the border of the goal neighborhood. This is done using on an abstract level, where search is simpler and faster. An action travels across a relatively large area, with no need to deal with the details of that area. Third, complete the path by traveling from the border of the goal neighborhood to the goal position.

The abstracted graph for on-line search is built using information extracted

from the problem maze. We discuss in more detail how the framework for hierarchical search is built (pre-processing) and how it is used for path-ﬁnding (on-line search). Initially we focus on building hierarchy two levels: one low level and one abstract level. Adding more hierarchical levels is discussed at the end of this section. We illustrate how our approach works on the small 40 × 40 map shown in Figure 1 (a).

**Pre-processing a Grid**

The ﬁrst step in building the framework for hierarchical search deﬁnes a topology ical abstraction of the maze. We use this maze abstraction to build an abstract graph for hierarchical search. The topological abstraction covers the maze with a set of disjunct rectangular areas called clusters. The bold lines in Figure 1 (b) show the abstract clusters used for topological abstraction. In this example, the 40 × 40 grid is grouped into 16 clusters of size 10 × 10. Note that no domain knowledge is used to do this abstraction (other than, perhaps, tuning the size of the clusters). For each border line between two adjacent clusters, we identify a (possibly empty) set of entrances connecting them. An entrance is a maximal obstacle-free segment along the common border of two adjacent clusters c 1 and c2 , formally deﬁned as below. Consider the two adjacent lines of tiles l1 and l2 , one in each cluster, that determine the border edge between c1 and c2 . For a tile t ∈ l1 ∪ l2 , we deﬁne symm(t) as being the symmetrical tile of t with respect to the border between c1 and c2 . Note that t and symm(t) are adjacent and never belong to the same cluster. An entrance e is a set of tiles that respects the following conditions:

• The border limitation condition: e ⊂ l1 ∪ l2 . This condition states that

an entrance is deﬁned along and cannot exceed the border between two

adjacent clusters.

• The symmetry condition: ∀t ∈ l1 ∪ l2 : t ∈ e ⇔ symm(t) ∈ e.

• The obstacle free condition: an entrance contains no obstacle tiles.

• The maximality condition: an entrance is extended in both directions as

long as the previous conditions remain true.

Figure 2 shows a zoomed picture of the upper-left quarter of the sample map.

The picture shows details on how we identify entrances and use them to build

the abstracted problem graph. In this example, the two clusters on the left side

are connected by two entrances of width 3 and of width 6 respectively. For each

entrance, we deﬁne one or two transitions, depending on the entrance width. If

the width of the entrance is less than a predeﬁned constant (6 in our example),

then we deﬁne one transition in the middle of the entrance. Otherwise, we deﬁne two transitions, one on each end of the entrance.

We use transitions to build the abstract problem graph. For each transition we

deﬁne two nodes in the abstract graph and an edge that links them. Since such an edge represents a transition between two clusters, we call it an inter-edge. Inter- edges always have length 1. For each pair of nodes inside a cluster, we deﬁne an edge linking them, called an intra-edge. We compute the length of an intra-edge by searching for for an optimal path inside the cluster area.

Figure 2 shows all the nodes (light grey squares), all the inter-edges (light grey

lines), and part of the intra-edges (for the top-right cluster). Figure 3 shows the

details of the abstracted internal topology of the cluster in the top-right corner of Figure 2. The data structure contains a set of nodes as well as distances between them. We deﬁne the distance as 1 for a straight transition and 1.42 1 for a diagonal 1 The generic path-ﬁnding library that we used in our experiments utilizes this value for approximating 2. A slightly more appropriate approximation would probably be 1.41 transition. We only cache distances between nodes and discard the actual optimal paths corresponding to these distances. If desired, the paths can also be stored, for the price of more memory usage. See Section 3.2.2 for a discussion. Figure 4 (a) shows the abstract graph for our running example. The picture includes the result of inserting the start and goal nodes S and G into the graph (the dotted lines), which is described in the next sub-section. The graph has 68 nodes, including S and G, which can change for each search. At this level of abstraction, there are 16 clusters with 43 inter-connections and 88 intra-connections. There are 2 additional edges that link S and G to the rest of the graph. For comparison, the low-level (non-abstracted) graph contains 1, 463 nodes, one for each unblocked tile, and 2, 714 edges. Once the abstract graph has been constructed and the intra-edge distances computed, the grid is ready to use in a hierarchical search. This information can be pre-computed (before a game ships), stored on disk, and loaded into memory at game run-time. This is sufﬁcient for static (non-changing) grids.

For dynamically changing grids, the pre-computed data has to be modiﬁed at run-time. When the grid topology changes (e.g., a bridge blows up), the intra- and inter-edges of the affected local clusters need to be re-computed.

**On-line Search**

The ﬁrst phase of the on-line search connects the starting position S to the border of the cluster containing S. This step is completed by temporarily inserting S into the abstract graph. Similarly, connecting the goal position G to its cluster border is handled by inserting G into the abstract graph.

After S and G have been added, we use A\* [10] to search for a path between S

and G in the abstract graph. This is the most important part of the on-line search. It provides an abstract path, the actual moves from S to the border of S’s cluster, the abstract path to G’s cluster, and the actual moves from the border of G’s cluster to G.

The last two steps of the on-line search are optional:

1. Path-reﬁnement can be used to convert an abstract path into a sequence of

moves on the original grid.

2. Path-smoothing can be used to improve the quality of the path-reﬁnement

solution.

The abstract path can be reﬁned in a post-processing step to obtain a detailed

path from S to G. For many real-time path-ﬁnding applications, the complete path is not needed—only the ﬁrst few moves. This information allows the character to start moving in the right direction towards the goal. In contrast, A\* must complete its search and generate the entire path from S to G before it can determine the ﬁrst steps of a character.

Consider a domain where dynamic changes occur frequently (e.g., there are

many mobile units travelling around). In such a case, after ﬁnding an abstract

path, we can reﬁne it gradually as the character navigates towards the goal. If the current abstract path becomes invalid, the agent discards it and searches for another abstract path. There is no need to refine the whole abstract path in advance.

**Searching for an Abstract Path**

To be able to search for a path in the abstract graph, S and G have to be part of

the graph. The processing is the same for both start and goal and we show it only for node S. We connect S to the border of the cluster c that contains it. We add S to the abstract graph and search locally for optimal paths between S and each of the abstract nodes of c. When such a path exists, we add an edge to the abstract graph and set its weight to the length of the path. In Figure 4 we represent these edges with dotted lines.

In our experiments we assume that S and G change for each new search.

Therefore, the cost of inserting S and G is added to the total cost of ﬁnding a

solution. After a path is found, we remove S and G from the graph. However,

in practice this computation can be done more efﬁciently. Consider a game when many units have to ﬁnd a path to the same goal. In this case, we insert G once and re-use it. The cost of inserting G is amortized over several searches. In general, a cache can be used to store connection information for popular start and goal nodes.

After inserting S and G, the abstract graph can be used to search for an abstract

path between S and G. We run a standard single-agent search algorithm such as

A\* on the abstract graph.

**Path Reﬁnement**

Path reﬁnement translates an abstract path into a low-level path. Each cluster

crossing in the abstract path is replaced by an equivalent sequence of low-level

moves.

If the cluster pre-processing cached these move sequences attached to the

intra-edges, then reﬁnement is simply a table look-up. Otherwise, we perform

small searches inside each cluster along the abstract path to re-discover the opti-

mal local paths. There are two factors that limit the complexity of the reﬁnement search. First, abstract solutions are guaranteed to be correct, provided that the environment does not change after ﬁnding an abstract path. This means that we never have to backtrack and re-plan for correcting the abstract solution. Second,

the initial search problem has been decomposed into several very small searches (one for each cluster on the abstract path), with low complexity.

**Path Smoothing**

The topological abstraction phase deﬁnes only one transition point per entrance.

While this is efﬁcient, it gives up the optimality of the computed solutions. So-

lutions are optimal in the abstract graph but not necessarily in the initial problem graph.

To improve the solution quality (i.e., length and aesthetics), we perform a post-

processing phase for path smoothing. Our technique for path smoothing is simple, but produces good results. The main idea is to replace local sub-optimal parts of the solution by straight lines. We start from one end of the solution. For each node in the solution, we check whether we can reach a subsequent node in the path in a straight line. If this happens, then the linear path between the two nodes replaces the initial sub-optimal sequence between these nodes.

**Experimental Results for Example**

The experimental results for our running example are summarized in the ﬁrst two rows of Table 1. L-0 represents running A\* on the low-level graph (we call this level 0). L-1 uses two hierarchy levels (i.e., level 0 and level 1), and L-2 uses three hierarchy levels (i.e., level 0, level 1, and level 2). The meaning of the last row, labeled L-2, is described in Section 3.5.

Low-level (original grid) search using Manhattan distance as the heuristic has

poor performance. Our example has been chosen to show a worst-case scenario.

Without abstraction, A\* will visit all the unblocked positions in the maze. The

search expands 1, 462 nodes. The only factor that limits the search complexity is the maze size. A larger map with a similar topology represents a hard problem for A\*.

The performance is greatly improved by using hierarchical search. When in-

serting S into the abstract graph, it can be linked to only one node on the border

of the starting cluster. Therefore we add one node (corresponding to S) and one

edge that links S to the only accessible node in the cluster. Finding the edge cost

uses a search that expands 8 nodes. Inserting G into the graph is identical.

A\* is used on the abstracted graph to search for a path between S and G.

Searching at level 1 also expands all the nodes of the abstract graph. The problem is also a worst-case scenario for searching at level 1. However, this time the search effort is much reduced.

The main search expands 67 nodes. In addition, inserting S and G expands 16

nodes. In total, ﬁnding an abstract path requires 83 node expansions. This effort

is enough to provide a solution for this problem—the moves from S to the edge of its cluster and the abstract path from the cluster edge to G. If desired, the abstract path can be reﬁned, partially or completely, for additional cost. The worst case is when we have to reﬁne the path completely and no actual paths for intra-edges were cached. For each intra-edge (i.e., cluster crossing) in the path, we perform a search to compute a corresponding low-level action sequence. There are 12 such small searches, which expand a total of 145 nodes.

**Adding Levels of Hierarchy**

The hierarchy can be extended to several levels, transforming the abstract graph

into a multi-level graph. In a multi-level graph, nodes and edges have labels showing their level in the abstraction hierarchy. We perform path-ﬁnding using a combination of small searches in the graph at various abstraction levels. Additional levels in the hierarchy can reduce the search effort, especially for large mazes.

See Appendix A.2.2 for details on efﬁcient searching in a multi-level graph. To

build a multi-level graph, we structure the maze abstraction on several levels. The higher the level, the larger the clusters in the maze decomposition. The clusters for level l are called l-clusters. We build each new level on top of the existing structure. Building the 1-clusters has been presented in Section 3.1. For l ≥ 2, an l-cluster is obtained by grouping together n × n adjacent (l − 1)-clusters, where n is a parameter.

Nodes on the border of a newly created l-cluster update their level to l (we call

these l-nodes). Inter-edges that make transitions between l-clusters also increase

their level to l (we call these l-inter-edges).

We add intra-edges with level l (i.e., l-intra-edges) for pairs of communicating

l-nodes placed on the border of the same l-cluster. The weight of such an edge

is the length of the shortest path that connects the two nodes within the cluster,

using only (l − 1)- nodes and edges. More details are provided in Section A.2.2.

Inserting S into the graph iteratively connects S to the nodes on the border of

the l-cluster that contains it, with l increasing from 1 to the maximal abstraction

level. Searching for a path between S and a l-node is restricted to level l − 1

and to the area of the current l-cluster that contains S. We perform an identical

processing for G too.

The way we build the abstract graph ensures that we always ﬁnd the same

solution, no matter how many abstract levels we use. In particular, adding a new

level l ≥ 2 to the graph does not diminish the solution quality. Here we provide

a brief intuitive explanation rather than a formal proof of this statement. A new

edge added at level l corresponds to an existing shortest path at level l − 1. The

weight of the new edge is set to the cost of the corresponding path. Searching at

level l ﬁnds the same solution as searching at level l − 1, only faster.

In our example, adding an extra level with n = 2 creates 4 large clusters, one

for each quarter of the map. The whole of Figure 2 is an example of a single

2-cluster. This cluster contains 2 × 2 1-clusters of size 10 × 10. Besides S, the

only other 2-node of this cluster is the one in the bottom-left corner. Compared to level 1, the total number of nodes at the second abstraction level is reduced even more. Level 2, where the main search is performed, has 14 nodes (including S and G). Figure 4 (b) shows level 2 of the abstract graph. The edges pictured as dotted lines connect S and G to the graph at level 2. Abstraction level 2 is a good illustration of how the pre-processing solves local constraints and reduces the search complexity in the abstract graph. The 2-cluster

shown in Figure 2 is large enough to contain the large dead end “room” that exists in the local topology. At level 2, we avoid any useless search in this “room” and go directly from S to the exit in the bottom-left corner.

After inserting S and G, we are ready to search for a path between S and

G. We search only at the highest abstraction level. Since start and goal have the

highest abstraction level, we will always ﬁnd a solution, assuming that one exists.

The result of this search is a sequence of nodes at the highest level of abstraction. If desired, the abstract path can repeatedly be reﬁned until the low-level solution is obtained.

**Experimental Results**

* 1. **Experimental Setup**

Experiments were performed on a set of 120 maps created by our implementation project , varying in size from 20x20 to 42x42. For each map, 3 to 4 searches were run using randomly generated S and G pairs where a valid path between the two locations existed.

The atomic map decomposition uses *octiles*. Octiles are tiles that define the adjacency relationship in 3 straight and 3 diagonal directions. The cost of vertical and horizontal transitions is 1. Diagonal transitions have the cost set to 1.41. We do not allow diagonal moves between two blocked tiles. Entrances with width less than 6 have one transition. For larger entrances we generate two transitions. We used Irrlicht graphics engine for implementing this algorithm.This library is used as a research tool for quickly implementing different search algorithms using different grid representations. Because of its generic nature, there is some overhead associated with using the library. All times reported in this report should be viewed as generous upper bounds on a custom implementation. The timings were performed on an Intel i3 core processor with 320+GB of hdd space and 3 GB of RAM. The programs were compiled using Visual Studio C++ Compiler, and were Microsoft Windows 7.

**Analysis**

Figure 5 compares low-level A\* to abstract search on hierarchies with the maximal level set to 1, 2, and 3. The left graph shows the number of expanded nodes and the right graph shows the time. For hierarchical search we display the total effort, which includes inserting S and G into the graph, searching at the highest level, and refining the path. The real effort can be smaller since the cost of inserting S or G can be amortized for many searches, and path refinement is not always necessary.

The graphs show that, when complete processing is necessary, the first abstraction level is good enough for the map sizes that we used in this experiment. We assume that, for larger maps, the benefits of more levels would be more significant. The complexity reduction can become larger than the overhead for adding the level. As we show next, more levels are also useful when path refinement is not necessary and S or G can be used for several searches. Even though the reported times are for a generic implementation, it is important to note that for any solution length the appropriate level of abstraction was able to provide answers in less than 10 milliseconds on average. Through length 400, the average time per search was less than 5 milliseconds on a 2.6 Ghz machine. A\* is slightly better than HPA\* when the solution length is very small. A small solution length usually indicates an easy search problem, which A\* solves with reduced effort. The overhead of HPA\* (e.g., for inserting S and G) is in such cases larger than the potential savings that the algorithm could achieve. A\* is also better when S and G can be connected through a .straight. line on the grid.

In this case, using the Euclidian distance as heuristic provides perfect information, and A\* expands no nodes other than those that belong to the solution.

Figure 6 shows how the total effort for hierarchical search is composed of the abstract effort, the effort for inserting S and G, and the effort for solution refinement. The cost for finding an abstract path is the sum of only the main cost and the cost for inserting S and G. When S or G are reused for many searches, only part of this cost counts for the abstract cost of a problem. Considering these,

the figure show that finding an abstract path becomes easier in hierarchies with more levels.

Figure 7 shows the solution quality. We compare the solutions obtained with hierarchical path-finding to the optimal solutions computed by low-level A\*. We plot the error before and after path-smoothing. The error measures the overhead in percents and is computed with the following formula:

e =hl – ol/100

where hl is the length of the solution found with HPA\*, and ol is the length of the optimal solution found with A\*. The error is independent of the number of hierarchical levels. The only factor that generates sub-optimality is not considering all the possible transitions for an entrance. The cluster size is a parameter that can be tuned. We ran our performance tests using 1-clusters with size 10 \_ 10. This choice at level 1 is supported by the data presented in Figure 8. This graph shows how the average number of expanded nodes for an abstract search changes with varying the cluster size. While the main search reduces with increasing cluster size, the cost for inserting S and G

increases faster. The expanded node count reaches a minimum at cluster size 10. For higher levels, an l-cluster contains 2\_2 (l􀀀1)-clusters. We used this small value since, when larger values are used, the cost for inserting S and G increases faster than the reduction of the main search. This tendency is especially true on relatively small maps, where smaller clusters achieve good performance and the increased costs for using larger clusters may not be amortized. The overhead

of inserting S and G results from having to connect S and G to many nodes placed on the border of a large cluster. The longer the cluster border, the more nodes to connect to. We ran similar tests on randomly generated maps. The main conclusions were similar but, because of lack of space, we do not discuss the the possible transitions for an entrance.

The cluster size is a parameter that can be tuned. We ran our performance tests using 1-clusters with size 10 \_ 10. This choice at level 1 is supported by the data presented in Figure 8. This graph shows how the average number of expanded nodes for an abstract search changes with varying the cluster size. While the main search reduces with increasing cluster size, the cost for inserting S and G

increases faster. The expanded node count reaches a minimum at cluster size 10.

For higher levels, an l-cluster contains 2\_2 (l􀀀1)-clusters. We used this small value since, when larger values are used, the cost for inserting S and G increases faster than the reduction of the main search. This tendency is especially true on relatively small maps, where smaller clusters achieve good performance and the increased costs for using larger clusters may not be amortized. The overhead of inserting S and G results from having to connect S and G to many nodes placed on the border of a large cluster. The longer the cluster border, the more nodes to connect to. We ran similar tests on randomly generated maps.

**Algorithm Implemented**

Pseudo - code for the implemented code is:

void abstractMaze(void) {

E = ∅;

C[1] = buildClusters(1);

for (each c1 , c2 ∈ C[1]) {

if (adjacent(c1 , c2 ))

E = E ∪ buildEntrances(c1 , c2 );

}

}

void buildGraph(void) {

for (each e ∈ E) {

c1 = getCluster1(e, 1);

c2 = getCluster2(e, 1);

n1 = newNode(e, c1 );

n2 = newNode(e, c2 );

addNode(n1 , 1);

addNode(n2 , 1);

addEdge(n1 , n2 , 1, 1, INTER);

}

for (each c ∈ C[1]) {

for (each n1 , n2 ∈ N [c], n1 = n2 ) {

d = searchForDistance(n1 , n2 , c);

if (d < ∞)

addEdge(n1 , n2 , 1, d, INTRA);

}

}

}

void addLevelToGraph(int l) {

C[l] = buildClusters(l);

for (each c1 , c2 ∈ C[l]) {

if (adjacent(c1 , c2 ) == false)

continue;

for (each e ∈ getEntrances(c1 , c2 )) {

setLevel(getNode1(e), l);

setLevel(getNode2(e), l);

setLevel(getEdge(e), l);

}

}

for (each c ∈ C[l])

for (each n1 , n2 ∈ N [c], n1 = n2 ) {

d = searchForDistance(n1 , n2 , c);

if (d < ∞)

addEdge(n1 , n2 , l, d, INTRA)

}

}

void preprocessing(int maxLevel) {

abstractMaze();

buildGraph();

for (l = 2; l ≤ maxLevel; l + +)

addLevelToGraph(l);

}

void connectToBorder(node s, cluster c) {

l = getLevel(c);

for (each n ∈ N [c])

if (getLevel(n) < l)

continue;

d = searchForDistance(s, n, c);

if (d < ∞)

addEdge(s, n, d, l, INTRA);

}

void insertNode(node s, int maxLevel) {

for (l = 1; l ≤ maxLevel; l + +) {

c = determineCluster(s, l);

connectToBorder(s, c);

}

setLevel(s, maxLevel);

}

path hierarchicalSearch(node s, g, int l) {

insertNode(s, l);

insertNode(g, l);

absP ath = searchForPath(s, g, l);

llP ath = reﬁnePath(absP ath, l);

smP ath = smoothPath(llP ath);

return smP ath;

}

**Implementation Result Figures & Project Screens**

Figure 1: (a) The 40 × 40 maze used in our example. The obstacles are painted

in black. S and G are the start and the goal nodes. (b) The bold lines show the

boundaries of the 10 × 10 clusters.

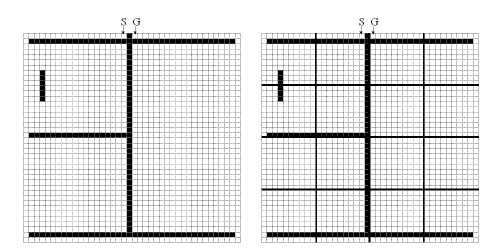
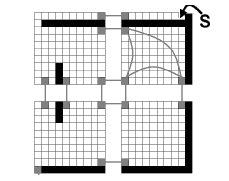
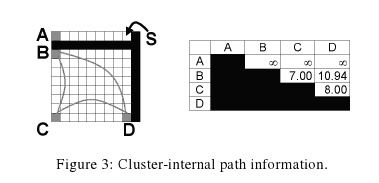


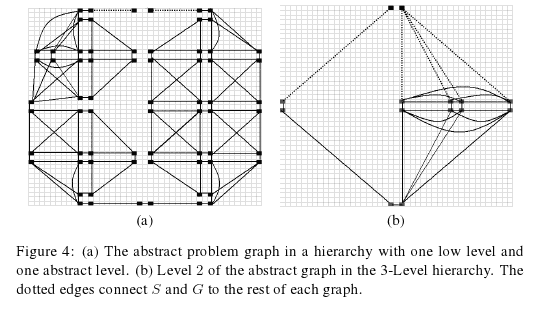
Figure 2: Abstracting the top-left corner of the maze. All abstract nodes and inter-

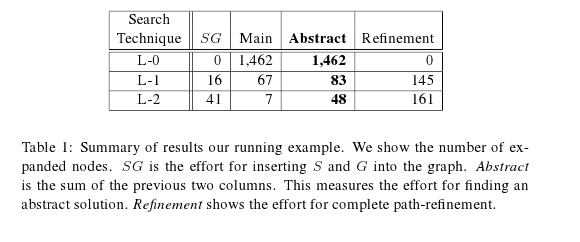
edges are shown in light grey. For simplicity, intra-edges are shown only for the

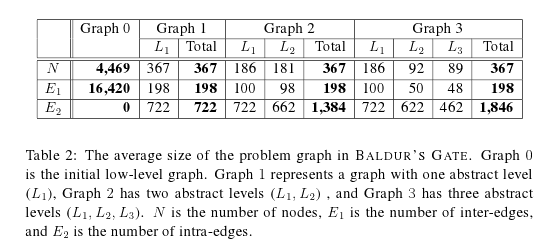
top-right cluster.

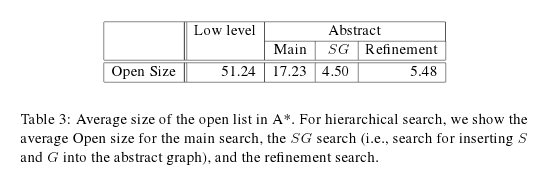


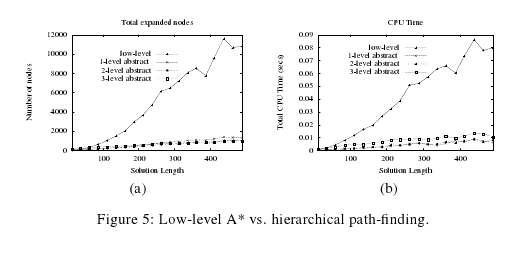


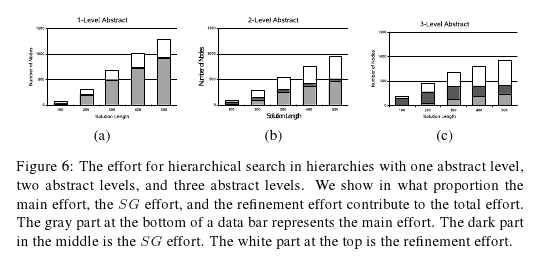


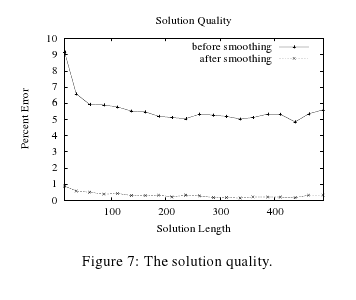






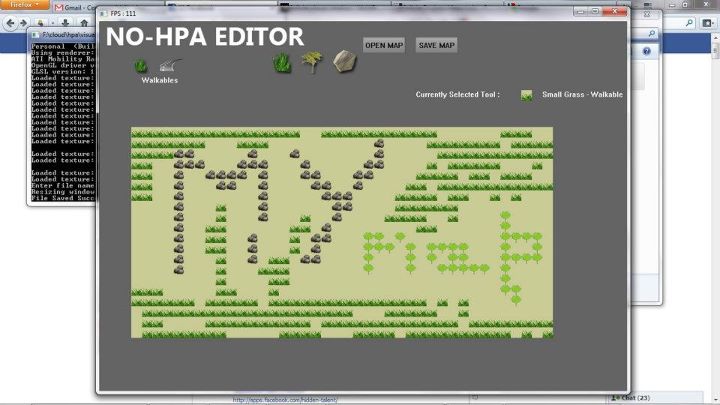




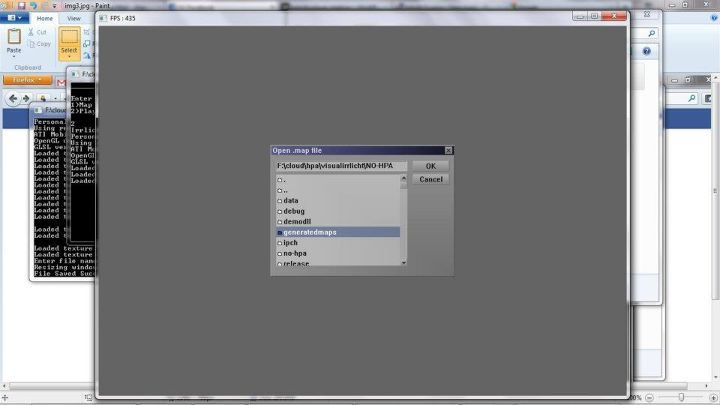


**SCREEN SHOTS OF THE PROJECT APPLICATION:**

Editor Window



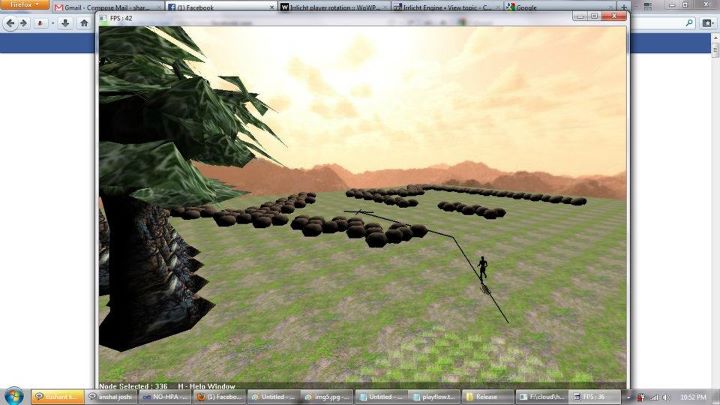
Map Loader Window

****

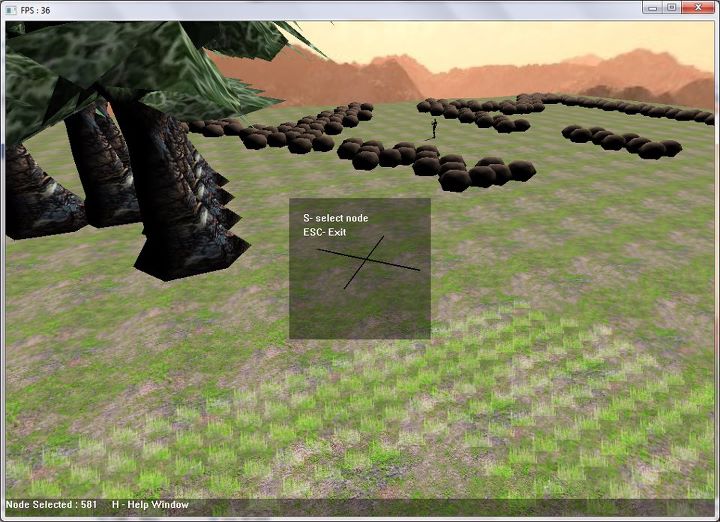
3D Environment for searching the path:

****

PATH FOUND:



Help Window



**REFERENCES**

**abotea.rsise.anu.edu.au/data/**

**www.aigamedev.com/open/review/near-optimal-hierarchical-pathfinding/**

**www.aigamedev.com/open/tutorials/clearance-based-pathfinding/  
  
www.webdocs.cs.ualberta.ca/~games/pathfind/  
  
www.aiwisdom.com/bytopic\_astar.html  
  
www.policyalmanac.org/games/aStarTutorial.htm**

**en.wikipedia.org/wiki/A\*\_search\_algorithm**

[1] D. Z. Chen, R. J. Szczerba, and J. J. Urhan Jr. Planning Conditional Shortest

Paths Through an Unknown Environment: A Framed-Quadtree Approach.

In Proceedings of the 1995 IEEE/RSJ International Conference on Intel-

ligent Robots and System Human Interaction and Cooperation, volume 3,

pages 33–38, 1995.

[2] R. Holte, M. Perez, R. Zimmer, and A. MacDonald. Hierarchical A\*:

Searching Abstraction Hierarchies Efﬁciently. In Proceedings AAAI-96,

pages 530–535, 1996.

[3] R. Korf. Depth-ﬁrst Iterative Deepening: An Optimal Admissible Tree

Search. Artiﬁcial Intelligence, 97:97–109, 1985.

[4] A. Moore, L. Baird, and L. Kaelbling. Multi-Value-Functions: Efﬁcient

Automatic Action Hierarchies for Multiple Goal MDPs. In Proceedings

of the International Joint Conference on Artiﬁcial Intelligence (IJCAI ’99),

1999.

**Future work & scope**

Despite the importance and the amount of work done in path-ﬁnding, there are not. many detailed publications about hierarchical path-ﬁnding in commercial games.

In this paper we have presented a hierarchical technique for efﬁcient near-

optimal path-ﬁnding. Our approach is domain-independent, easy to apply and

works well for different kinds of map topologies. The method adapts to dynami-

cally changing environments. The hierarchy can be extended to several abstraction

levels, making it scalable for large problem spaces. We tested our program using

maps extracted from a real game, obtaining near-optimal solutions signiﬁcantly

faster than low-level A\*.

We have many ideas for future work in hierarchical path-ﬁnding. We plan to

optimize the way that we insert S and G into the abstract graph. As Figure 6

shows, these costs increase signiﬁcantly with adding a new abstraction layer. One

strategy for improving the performance is to connect S only to a sparse subset of

the nodes on the border, maintaining the completeness of the abstract graph. For

instance, if each “unconnected” node (i.e., a node on the border to which we did

not try to connect S) is reachable in the abstract graph from a “connected” node

(i.e., a node on the border to which we have connected S), then the completeness

is preserved. Another idea is to consider for connection only border nodes that are

on the direction of G. However, this last idea does not guarantee the completeness

and it is hard to evaluate the beneﬁts beforehand. If the search fails because of

the graph incompleteness, we have to perform it again with the subset of border

nodes gradually enlarged.

The clustering method that we currently use is simple and produces good

results. However, we also want to explore more sophisticated clustering meth-

ods. An application-independent strategy is to automatically minimize some of

the clustering parameters such as number of abstract clusters, cluster interactions,

and cluster complexity (e.g., the percentage of internal obstacles).