Project 1 - Any Angle Path

Qandeel Sajid and Tina Nye

March 1, 2012

Contents

0.1 Project Description

Path Planning is an important aspect of artificial intelligence that can be used for multiple applications especially in the field of robotics and computer gaming. In gaming, good path planning makes the difference in the quality of games; it is likely that the player has to navigate on a grid while avoiding obstacles and looking for the shortest path to its desired destination. To find the shortest path, path planning depends heavily on search algorithms that can find the best path under time and space constraints. Some of these search algorithms include A*, Theta* or using the Visibility Graphs to search the grid.

In this assignment, the focus is to implement A* and Theta*. This paper discusses the implementation of these algorithms and tested them on 50 different randomly generated grid samples. A* on a grid, Theta*, and A* on a visibility graph are tested on the same 50 grids. This paper discusses the results and compares the algorithms based on the path costs and runtimes. After the discussion, solutions to the problems given for this project are shown.

0.2 Implementation

The program for this project is done in Python, and the visualization is implemented with PyGame. For the program, the standard size of the grid is 100 columns by 50 rows. The grid size can be altered by changing the values of the variables *colLength* and *rowLength* found above in the files functions.py and main.py. The graph and visualization are called in main.py. A* and Theta* are abstracted into a class which, when called by the main, return the path found. More details on the implementation are found below.

0.2.1 Graph Setup

The graph for the program is implemented using SparseGraph from the Python standard graph library. Each end of a cell on the grid represents a vertex in the graph. The blocked cells are visualized by the gray cells. These vertices are connected to neighboring vertices through an edge if that edge does not go through a blocked cell. Additional functions found in functions.py are used to convert between the vertex ID, and the pixel location of the vertex. Figure 1 shows the connected edges highlighted in red. The surrounding gray cells represent a border around the grid.

For the implementation of the graph, only the top-left (x, y) value is remembered for the obstacles. This will be important later in the report for the implementation of Theta*.

0.2.2 A*

Figure 2 shows the pseudo-code for the A* algorithm used in the implementation. The actual implementation of the code follows the pseudo-code closely. Originally, the implementation of A* used the standard binary heap that comes with Python as the fringe. The fringe ordered the nodes so the one with the smallest f(n) value is on top. It broke ties using the total path cost (g(n)). After the first demo of the project, we implemented our own binary heap using Python lists.

Optimization was our goal for the first implementation of A*. For optimization there is no specific list for the "closed" vertices. Instead, flags for whether the vertex is in the fringe or has been expanded are stored in the vertex of the graph. By doing this, the algorithm did not not require looping through the set of vertices in the closed list; it would know if the node has been expanded as soon as it checks the boolean value for that vertex from the graph. Something done to optimize the first implementation of the algorithm is that the algorithm does not have to check for blocked cells as it is planning. Only valid edges have been inserted into the graph which means A* never has to test whether an edge between two vertices is through a blocked

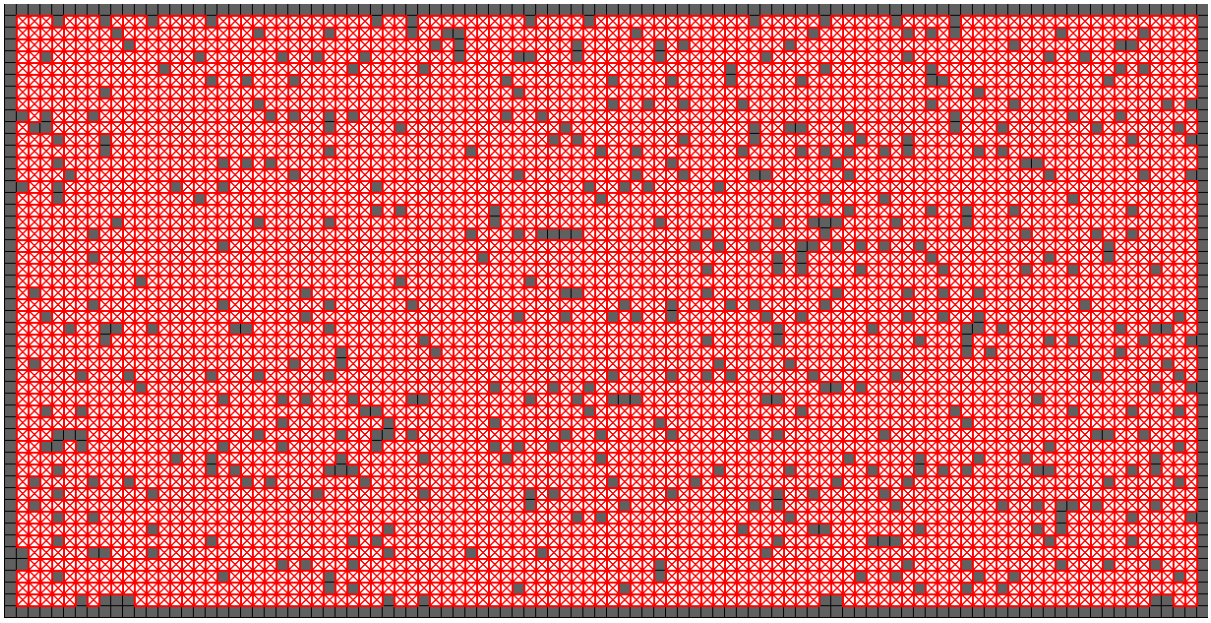


Figure 1: Visualization of the graph used by the program

cell. The results show this gives a faster solution. After the demo, another optimization was done through the binary heap. For the standard binary heap there was no way to remove a vertex so the authors implemented their own. The remove function required copying all the elements, except the one that has to be removed, into a new heap. Obviously, this is wasteful since it requires a hard copy of the of the heap every time a node needed to be removed. The new implementation uses the remove function for Python lists.

0.2.3 Theta*

Theta* allows paths to go through cells rather than just moving from corner to corner on the cell as well as connect to vertices that are not neighbors. This gives better-looking, shorter paths for games. The main parts of the Theta* algorithm are similar to A*. The difference lies with the UpdateVertex function shown in Fig. 3. As shown, this algorithm relies on the use of the LineOfSight function. The project was provided with an algorithm for implementing the LineOfSight which just checks if the straight line connection between two vertices does not go through any blocked cells. Unfortunately, with the way the authors implemented the graph (remembering only the top-left (x, y) values of the obstacles cells), the algorithm given for LineOfSight cannot be used in this project. For this, the authors made a customized function that does the same thing as LineOfSight. This function just finds a linear equation to represent the straight line connecting between the two points. Then, in small intervals, the functions checks the (x, y) value from the equation and rounds the value down to check if the (x, y) values matches top-left x,y value of an obstacle.

After the first demo, the authors discovered two bugs in the program. One of the bugs occurred very rarely but has to do with how the function check of the floored (x, y) value matched that of an obstacle. The second bug had to do with the interval at which the function is checked. On a grid with larger cells, the interval can be large as well. For smaller cell the interval also needs to be small otherwise it mistakenly gives a wrong solution in which paths go through blocked cells. By having a smaller interval, the function takes longer to check if a path is free. Smaller intervals are not done in original Theta* causing the time taken by the optimized to be similar to the original. Theta* was optimized the same way as A*.

```
Main()
    g(s_start) := 0;
    parent(s_start) := s_start;
    fringe := ∅;
    fringe.Insert(s_start, g(s_start) + h(s_start));
    closed := ∅;
    while fringe ≠ ∅ do
        s := fringe.Pop();
        if s = s_goal then
            return "path found";
        end
        closed := closed ∪ {s};
        foreach s' ∈ succ(s) do
            if s' ∉ closed then
                if s' ∉ fringe then
                    g(s') := ∞;
                    parent(s') := NULL;
                end
                UpdateVertex(s, s');
            end
        end
    end
    return "no path found";
end
UpdateVertex(s,s')
    if g(s) + c(s, s') < g(s') then
        g(s') := g(s) + c(s, s');
        parent(s') := s;
        if s' ∈ fringe then
            fringe.Remove(s');
        end
        fringe.Insert(s', g(s') + h(s'));
    end
end
```

Figure 2: Shows the pseudo-code for the A* algorithm used in the implementation

```
UpdateVertex(s,s')
    if LineOfSight(parent(s), s') then
        /* Path 2 */
        if g(parent(s)) + c(parent(s), s') < g(s') then
            g(s') := g(parent(s)) + c(parent(s), s');
            parent(s') := parent(s);
            if s' ∈ open then
                open.Remove(s');
            end
            open.Insert(s', g(s') + h(s'));
        end
    else
        /* Path 1 */
        if g(s) + c(s, s') < g(s') then
            g(s') := g(s) + c(s, s');
            parent(s') := s;
            if s' ∈ open then
                open.Remove(s');
            end
            open.Insert(s', g(s') + h(s'));
        end
    end
end
```

Figure 3: Shows the pseudo-code for the Theta* algorithm used in the implementation

0.2.4 Visibility Graph

In the Visibility Graph, only edges connecting the corner of blocked cells are allowed. These edges cannot go through any blocked vertices. The Visibility Graph is made by iterating through

the corners of the obstacles and checking if the edge is free of obstacles using LineOfSight. Then A* is just applied to the Visibility Graph to find the shortest path from the start to goal. The algorithm for the Visibility Graph is not shown.

0.3 Results and Discussion

Both the original A* and Theta* and their optimized versions were run on the same set of 50 randomly generated grids with only 10 percent of the cells as obstacles. Lastly, the Visibility Graph was run on that same set of 50 grids. The results are shown and discussed below. These tests were conducted on a machine with an Intel core i7 CPU and an nVidia GeForce GTX 560M.

0.3.1 Original A* and Theta*

Tables 1 and 2 show the average path lengths and runtimes found by running the set of 50 grids mentioned above when 10 percent and 50 percent of the cells as obstacles respectively. As shown by the path lengths results, Theta* gives the shorter path. This is definitely because Theta* can move to vertices that are not its direct neighbors.

100x50 Grid	**A* on Grid**	**Theta***
Random Obstacles, 10%	43.5766	41.394
Random Obstacles, 50%	45.2565	42.8329

Table 1: Path length

Although the Theta* algorithm has to traverse each edge it finds using the LineOfSight function, it still takes less time than the A* algorithms. This is likely because the path it finds, although shorter than A*, might not be as short as it could have been. Theta* also does not have to waste time looking at just its closest neighboring vertices, it can jump to an vertex that it can reach with a valid edge. In addition, this implementation of Theta* uses larger intervals to check the linear equation in the LineOfSight which gives it a lot smaller runtime than expected.

As shown by the Tables, the algorithms clearly give a long path and take more time for their solution when the percentage of obstacles is at 50.

100x50 Grid	**A* on Grid**	**Theta***
Random Obstacles, 10%	31.187	8.93683
Random Obstacles, 50%	34.4797	9.74801

Table 2: Runtime

0.3.2 Optimized A* and Theta*

The optimized versions of A* and Theta* were tested on the same set of 50 randomly generated grid as their original counterparts. The results are shown in Tables 3 and 4. Due to a lack of time, these algorithm were not tested on grids with 50 percent algorithms. In the optimized, the paths found by the Theta* are not noticeably better than those found by A*, but they still are shorter. The time taken by Theta* to find the solution is more than the A*. As mentioned above, the optimized Theta* uses smaller intervals in its LineOfSight equation which causes it to take more time than the optimized A* compared to how well the originals did.

100x50 Grid	A* on Grid	Theta*
Random Obstacles, 10%	40.9819	39.2298
Random Obstacles, 50%	N/A	N/A

Table 3: Path length

100x50 Grid	A* on Grid	Theta*
Random Obstacles, 10%	27.3734	34.7868
Random Obstacles, 50%	N/A	N/A

Table 4: Runtime

0.3.3 Comparison between old and new implementation results

The averaged path length found by the original A* compared to the optimized is about 5 percent longer and the same can be said for the Theta* implementations. The focus of the optimization is actually not the path length since nothing that can radically affect the path lengths was implemented in the optimized version.

The optimization was for improving the runtime found by both algorithms. As shown by the tables above, the averaged runtime for optimized A* is about 20 percent less than that taken by the original. This is likely because of the efficient implementation of the binary heap remove. By not hard copying the heap, the optimized decreased in the amount of time it takes. The results given Theta* cannot be objectively compared. The optimized version of Theta* uses smaller intervals while the original uses larger to check edges.

0.3.4 Visibility Graph

Tables 5 and 6 show the path length and runtime taken for the same set of 50 grids. The average path found by applying A* on a Visibility Graph is better those found by either implementation of Theta*. This is because the Visibility Graph connects only the useful edges (those around obstacles) compared to Theta* which just connect to vertices with low f(n) values.

100x50 Grid	A* on Visibility Graph
Random Obstacles, 10%	39.1003
Random Obstacles, 50%	N/A

Table 5: Path length

The time taken by the Visibility Graph is horrible though because iterating through edges and applying LineOfSight takes a long time in Python. To reduce the time taken by the algorithm, only nodes being expanded connect to the vet ices in the graph.

100x50 Grid	A* on Visibility Graph
Random Obstacles, 10%	1376.18
Random Obstacles, 50%	N/A

Table 6: Runtime

0.4 A* heuristic vs Theta* heuristic

From the results in Table 1 and Table 3, Theta* always has shorter path lenths; however, A* and Theta* switched between which one was faster depending on the implementation as seen in Table 2 and Table 4. This discrepancy in runtimes may be due to the non-monotonic property

of Theta*; it is able to find a shorter path at the expense of runtime. Otherwise, the discrepancy may be due to something that might have happened/changed in the optimized implementation.

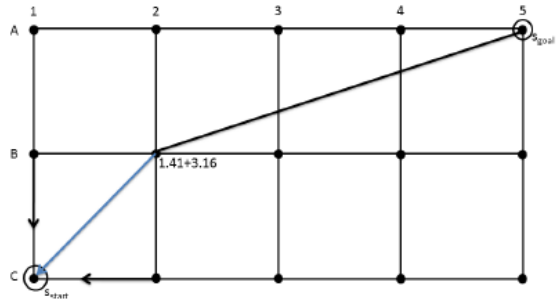
On a grid, it is fair that A* and Theta* use different h-values. A* can only traverse along edges, which in the case of a grid, is eight discrete directions, but Theta* can traverse a path in a direction along any angle. It makes sense that Theta* uses the striaght-line (linear) distance formula between two points to calculate the its heuristic because it can move in any angle, so since A* does not have this capability it needs a heuristic that gives a appropriate distance from the goal while considering its movement capabilities. According to A. Nash et al. in the paper "Theta*: Any-Angle Path Planning on Grids", "A* with consistent h-values finds paths of the same length no matter how small or large the h-values are." In conclusion, it is a legitimate arguement that A* and Theta* use different h-values.

0.5 Explain why A* with the h-values from Equation 1 is guaranteed to find shortest grid paths

Equation 1 calculates the Manhattan distance, which is the sum of all hoizontal and vertical distances from the current vertex to the goal. By using the Manhattan distance as a heuristic function, A* never overestimates the distance to the goal, thus making it admissible. This does not consider if there are obstacles in the path that could make the path longer; therefore, it assumes the shortest path along eight discrete directions. A* is also consistent (or monotonic) because a node selected for expansion was reached because it had the lowest possible cost. This is similar to Dijkstra's best-first search algorithm to find the shortes path in a graph with no negative edges. The monotonic property of A* is that the f-value of an expanded vertex can not be any larger than one of its unexpanded visible neighbors after updating the nodes. Which means that any expanded vertex that traverses in reverse following the parents will yeild the shortest path from the start vertex to the expanded vertex in reverse. As a result, A* is guaranteed to find the shortest grid paths with the h-values from Equation 1 because it is admissible and consistent (monotonic).

0.6 Admissibility vs Monotonicity

Unlike the monotonic property of A*, Theta* can have an expanded vertex with an f-value larger than an f-value of one or more of its unexpanded visible neighbors. Consequently, any expanded vertex that traverses in reverse following the parents is not guaranteed to yeild the shortest path from the start vertex to the expanded vertex in reverse. This implies that Theta* has the ability find shorter paths at the expense of the runtime by expanding verticies more than once, whereas A* cannot. Figure 4 shows an example of the non-monotonicity of Theta*: Start vertex C1 expands to vertex B2, sets $f(B2) = 1.41 + 3.16 = 4.57$, and sets the parent of B1 to C1. Then, Theta* expands vertex B2 to vertex B3, sets $f(B3) = 2.24 + 2.24 = 4.48$, and sets the parent of B3 to B2. Under these circumstances, the expanded vertex B2 has a larger f-value than its unexpanded visible neighbor B3.



Start @ C1; Fringe: B1, B2, C2

C1:

$g(C1) = 0$

$h(C1) = \sqrt{4^2 + 2^2} = 4.47$

$f(B1) = 4.47$

B1:

$g(B1) = 1$

$h(B1) = \sqrt{4^2 + 1^2} = 4.12$

$f(B1) = 5.12$

B2:

$g(B2) = \sqrt{2}$

$h(B1) = \sqrt{3^2 + 1^2} = 3.16$

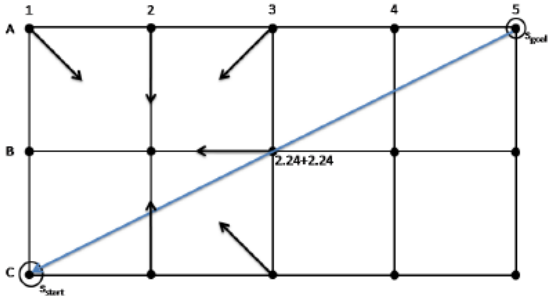
f(B1) = 4.57

C2:

$g(C2) = 1$

$h(C2) = \sqrt{3^2 + 2^2} = 3.61$

$f(C2) = 4.61$



Start @ B2; Fringe: A1, A2, A3, B3, C3; $g = \sqrt{2}$

A1:

$g(A1) = \sqrt{2} + \sqrt{2} = 2.83$

$h(A1) = \sqrt{4^2 + 0^2} = 4$

$f(A1) = 6.83$

A2:

$g(A2) = \sqrt{2} + 1 = 2.41$

$h(A2) = \sqrt{3^2 + 0^2} = 3$

$f(A2) = 5.41$

A3:

$g(A3) = \sqrt{2} + \sqrt{2} = 2.83$

$h(A3) = \sqrt{2^2 + 0^2} = 2$

$f(A3) = 4.83$

B3:

$g(B3) = \sqrt{2} + 1 = 2.41$

$h(B3) = \sqrt{2^2 + 1^2} = 2.24$

f(B3) = 4.65

C3:

$g(C3) = \sqrt{2} + \sqrt{2} = 2.83$

$h(C3) = \sqrt{2^2 + 2^2} = 2.83$

$f(C3) = 5.66$

Figure 4: Theta* non-monotonic f-values