

Final project report on Path Finding Problem

CS 7750

Zolbayar Magsar
Chanmann Lim
Yihan Xu

December 15, 2014

Electrical & Computer Engineering
University of Missouri, Columbia

Instructor: Dr. Yi Shang

Abstract

In many occasions, we face the problem of finding a path from one place to another place. In such situations, were we not just trying to find the shortest distance; but the traveling time also an important factor to be taken into consideration. In artificial intelligence, search algorithms are mainly used for such problems. On the other hand, when it comes to differentiating between such algorithms, it is often useful to implement them on a path finding problem and compare their features. We have attempted to create a game map which can be used in a simple path finding problem. On the map, we have two points, source and destination. We are also free to create an arbitrary shape of walls between them to create a problem. We then have myriad of ways to examine the precise difference between search algorithms and compare them in terms of execution speed, accuracy, and overall performance. To solve the path finding problem, we have implemented six major search algorithms representing 3 main types of search algorithm families, namely Breadth-First Search, Depth-First Search, Greedy Best-First Search, A* Search, Hill-Climbing, and Simulated Annealing. The map is created using a grid structure, so as to have a precise depiction of the 'search movements' of each of the algorithms visually in terms of characteristics and performance.

1 Introduction

Pathfinding or *pathing* is the plotting, by a computer application, of the shortest route between two points. Two primary problems of pathfinding are to find a path between two nodes in a graph and to find the optimal shortest path. Basic algorithms such as Breadth-first and Depth-first search address the first problem by exhausting all possibilities; starting from the given node, they iterate over all potential paths until they reach the destination node. The more complicated problem is finding the optimal path. However, it is not necessary to examine all possible paths to find the optimal one. Algorithms such as A* strategically eliminate paths, through heuristics while local search such as Hill-climbing further reduce search space to a constant number of choices to make search rocket fast.

At its core, a pathfinding method searches a graph by starting at one vertex and exploring adjacent nodes until the destination node is reached, generally with the intent of finding the shortest route. Although graph searching methods such as a Breadth-first search would find a route if given enough time, other methods, which "explore" the graph, would tend to reach the destination sooner. An analogy would be a person walking across a room; rather than examining every possible route in advance, the person would generally walk in the direction of the destination and only deviate from the path to avoid an obstruction, and make deviations as minor as possible.

We have implemented six different algorithms to traverse and find the pathway on a grid from a given square A to destination B using 3 main strategies of search algorithms:

1. Uninformed search
 - Breath-first search
 - Depth-first search
2. Heuristic search
 - Greedy best-first search
 - A* search
3. Local search
 - Hill-climbing
 - Simulated annealing

2 Algorithms and Implementations

There will be obstacles between A and B, and our implementation would show graphically how the algorithm manage to reach the destination, which in turn would visually demonstrate the actual performance of algorithms in real world problems. We made use of Manhattan distance as the heuristics for the informed search algorithms and as the scoring function for local searches.

2.1 Breadth-first search algorithm:

BFS is a strategy for searching in a graph when search is limited to essentially two operations: (a) visit and inspect a node of a graph; (b) gain access to visit the nodes that neighbor the currently visited node. The BFS begins at a root node and inspects all the neighboring nodes. Then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on.

```

while (true) {
    if (frontier.isEmpty()) return new Failure();
    node = removeFrontier(frontier.get(0));

    if (problem.isGoal(node)) return new Solution(node);
    addExplored(node);

    for (Action action : problem.getActions(node)) {
        Node child = problem.getResult(node, action);
        if (!explored.contains(child)) {
            if (!frontier.contains(child)) {
                addFrontier(child);
            }
        }
    }
}

```

The algorithm uses a queue data structure to store intermediate results as it traverses the graph, as follows:

1. Enqueue the root node
2. Dequeue a node in first-in-first-out manner and examine it
 - If the element sought is found in this node, quit the search and return a result.
 - Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.
3. If the queue is empty, every node on the graph has been examined quit the search and return "not found".
4. If the queue is not empty, repeat from Step 2.

2.2 Depth-first search algorithm:

DFS starts at the root(selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

```
while (true) {
    if (frontier.isEmpty()) return new Failure();
    node = removeFrontier(frontier.get(frontier.size() - 1));

    if (problem.isGoal(node)) return new Solution(node);
    addExplored(node);

    for (Action action : problem.getActions(node)) {
        Node child = problem.getResult(node, action);
        if (!explored.contains(child)) {
            if (!frontier.contains(child)) {
                addFrontier(child);
            }
        }
    }
}
```

DFS always expands the deepest node in the current frontier of the search tree. As seen above, the search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.

2.3 Best-first search algorithm:

It is a search algorithm which explores a graph by expanding the most promising node chosen according to a specified rule.

We used "best-first search" to refer specifically to a search with a heuristic function that attempts to predict how close the end of a path is to a solution, so that paths which are judged to be closer to a solution are extended first. This specific type of search is called **greedy best-first search**.

```
while (true) {
    if (frontier.isEmpty()) return new Failure();
    node = removeFrontier(getClosestNode(frontier));
    if (problem.isGoal(node)) return new Solution(node);
    addExplored(node);

    for (Action action : problem.getActions(node)) {
        Node child = problem.getResult(node, action);
        if (!explored.contains(child)) {
            if (!frontier.contains(child)) {
                addFrontier(child);
            }
        }
    }
}
```

By using a greedy algorithm, we expand the first successor of the parent. After a successor is generated:

1. If the successor's heuristic is better than its parent, the successor is set at the front of the queue (with the parent reinserted directly behind it), and the loop restarts.
2. Else, the successor is inserted into the queue (in a location determined by its heuristic value). The procedure will evaluate the remaining successors (if any) of the parent.

2.4 A* search algorithm:

A* assigns a weight to each open node equal to the weight of the edge to that node plus the approximate distance between that node and the finish. This approximate distance is found by the heuristic, and represents a minimum possible distance between that node and the end. This allows it to eliminate longer paths once an initial path is found. If there is a path of length x between the start and finish, and the minimum distance between a node and the finish is greater than x , that node need not be examined.

```
while (true) {
    if (frontier.isEmpty()) return new Failure();
    node = removeFrontier(getLowestCostNode());
    if (problem.isGoal(node)) return new Solution(node);
    addExplored(node);

    for (Action action : problem.getActions(node)) {
        Node child = problem.getResult(node, action);
        if (!explored.contains(child)) {
            if (!frontier.contains(child)) {
                addFrontier(child);
            } else {
                Node existing = frontier.get(frontier.indexOf(child));
                if (getCost(existing) > getCost(child)) {
                    removeFrontier(existing);
                    addFrontier(child);
                }
            }
        }
    }
}
```

When the value of the heuristic is exactly the true distance, A* examines the fewest nodes. As the value of the heuristic increases, A* examines fewer nodes but no longer guarantees an optimal path. In many applications (such as video games) this is acceptable and even desirable, in order to keep the algorithm running quickly.

2.5 Hill-climbing algorithm:

A local search algorithm starts from a candidate solution and then iteratively moves to a neighbors solution. When no improving configurations are present in the neighborhood, local search is stuck at a locally optimal point.

```
while (true) {
    Node neighbor = getHighestValueNode(current);
    addExplored(current);
    if (getValue(neighbor) <= getValue(current)) {
        break;
    }
    current = neighbor;
}
if (problem.isGoal(current)) return new Solution(current);
return new Failure();
```

It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a

better solution by incrementally changing a single element of the solution. If the change produces a better solution, an incremental change is made to the new solution, repeating until no further improvements can be found. Hill climbing is good for finding a local optimum (a solution that cannot be improved by considering a neighboring configuration) but it is not necessarily guaranteed to find the best possible solution (the global optimum) out of all possible solutions (the search space).

2.6 Simulated annealing algorithm:

At each step, the SA heuristic considers some neighboring state of the current state, and probabilistically decides between moving the system to that state or staying in the current state. These probabilities ultimately lead the system to move to states of lower energy. Typically this step is repeated until the system reaches a state that is good enough for the application, or until a given computation budget has been exhausted.

```
for (int k=0; ;k++) {
    temperature = coolingSchedule(k);
    if (temperature == 0) return new Failure();
    if (problem.isGoal(current)) return new Solution(current);

    Node nextNode = getRandomChildNode(current);
    addExplored(current);

    int deltaE = getValue(nextNode) - getValue(current);
    if (deltaE > 0) {
        current = nextNode;
    } else {
        double probability = Math.pow(Math.E, deltaE / temperature);
        if (probability > Math.random()) {
            current = nextNode;
        }
    }
}
```

The innermost loop of the simulated-annealing algorithm is quite similar to hill climbing. Instead of picking the best move, however, it picks a random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1.

Use of grids: Grids are commonly used in games for representing playing areas such as maps (in games like Civilization and Warcraft), playing surfaces (in games like pool, table tennis, and poker), playing fields (in games like baseball and football), boards (in games like Chess, Monopoly, and Connect Four), and abstract spaces (in games like Tetris).

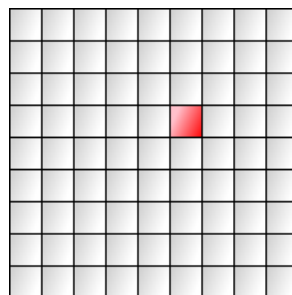


Figure 1: Square grids

Grids are built from a repetition of simple shapes. The most common grid is a square grid. Its simple, easy to work with, and maps nicely onto a computer screen. Square grids are the most common grids used in games, primarily because they are easy to use. Locations can use the familiar cartesian coordinates (x, y) and the axes are orthogonal. The square coordinate system is the same even if your map squares are angled on screen in an isometric or axonometric projection.

3 Results and Analysis

The results we have obtained, after running these algorithms individually both with and without a block between the points A and B, are pretty astonishing and is a pleasure for anyone to watch how these algorithms get the job done through such distinctive methodologies.

Let us compare the six algorithms in an exactly same problem case with a single square block in between the two points.

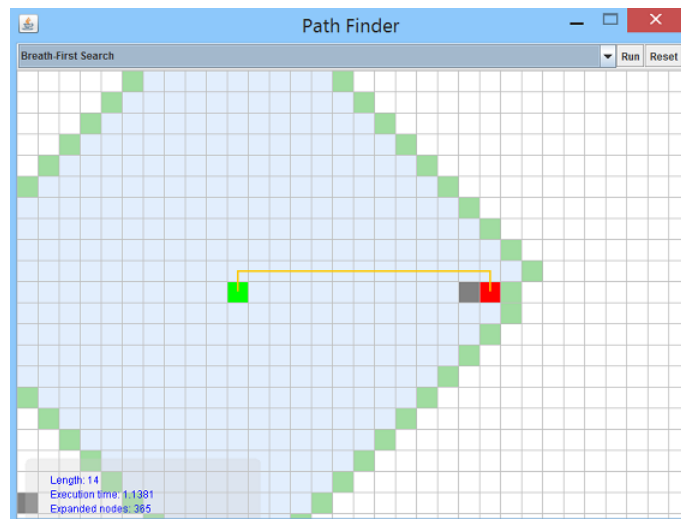


Figure 2: BFS with a block between two points

As we can see here, in breadth-first search algorithm, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded. Therefore the shape of the search space covered is guaranteed to be a square on a grid and the expanded node is the largest in case of BFS. The CPU execution time is also the longest since the algorithm is doing an exhaustive search to reach the destination node. We can see how blind (and wasteful) an uninformed search actually is. Breadth-first search is guaranteed to find the optimal path, so we can see that the path-cost is 14 in above case.

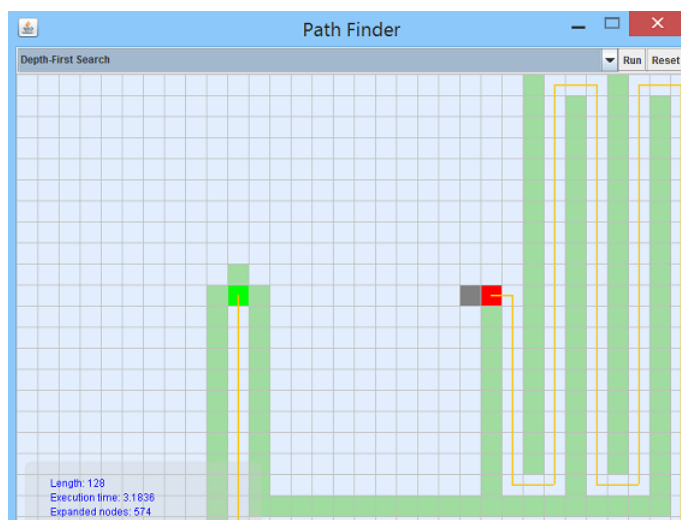


Figure 3: DFS with a block between two points

As we can see here, DFS expands the deepest node in the current frontier. Therefore the path cost is quite high in this example and the CPU execution time as well. It also expanded much more nodes than BFS, therefore proving that DFS is non-optimal.

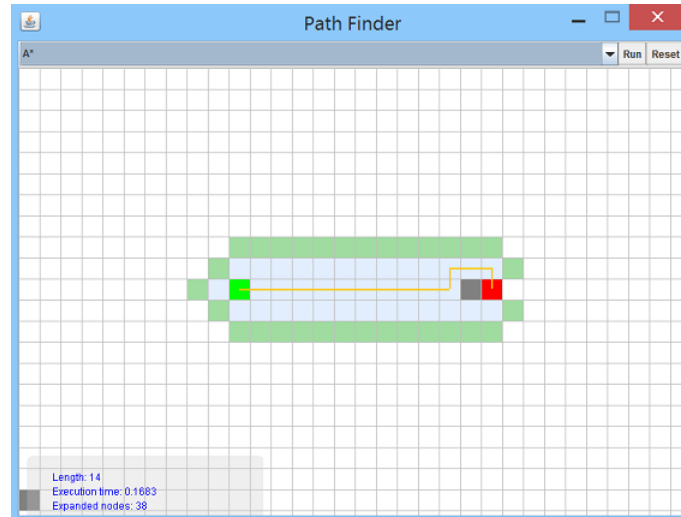


Figure 4: A* with a block between two points

We can see from the above picture how smart the heuristic search compared to the uninformed search is. A* actually has a sense of direction to look for the destination and is very effective in finding it. The length of the path is same as BFS, therefore guarantees an optimal path, because of an admissible heuristic function. We can see that the expanded node is greatly reduced and the CPU execution time accordingly.

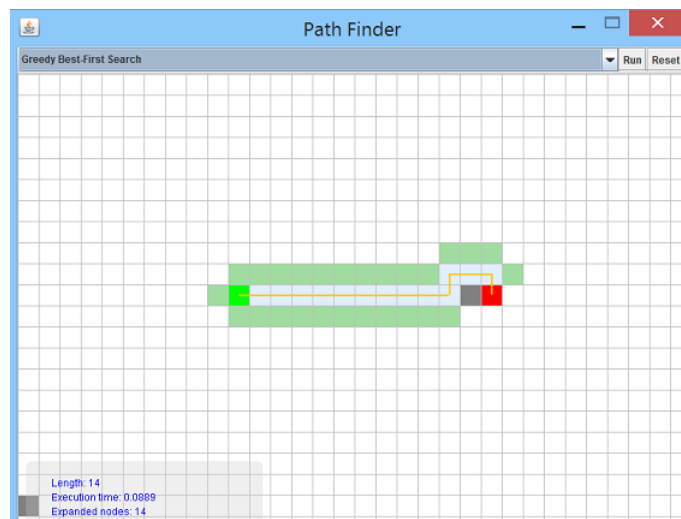


Figure 5: Greedy best-first search with a block between two points

The greedy best-first search is the most effective in this case with a minimum expanded nodes of 14, thereby having the least CPU execution time. This proves that a greedy algorithm can be very effective in the right circumstance. Instead of selecting the vertex closest to the starting point, it selects the vertex closest to the goal. Greedy Best-First-Search is not guaranteed to find a shortest path. However, it runs much quicker than A* algorithm.

As we can see (in Figure 6), a local search algorithm operates using a single current node (rather than multiple paths) and generally move only to neighbors of that node. Also the paths followed by the search are not retained. So we can testify from above that the hill-climbing search is simply a loop that continually moves in the direction of increasing value that is, uphill. And since there's an obstacle, it failed to find the path to the destination, because it simply did not keep track of alternative routes. Hence we can see the clear limitations of the hill-climbing search here.

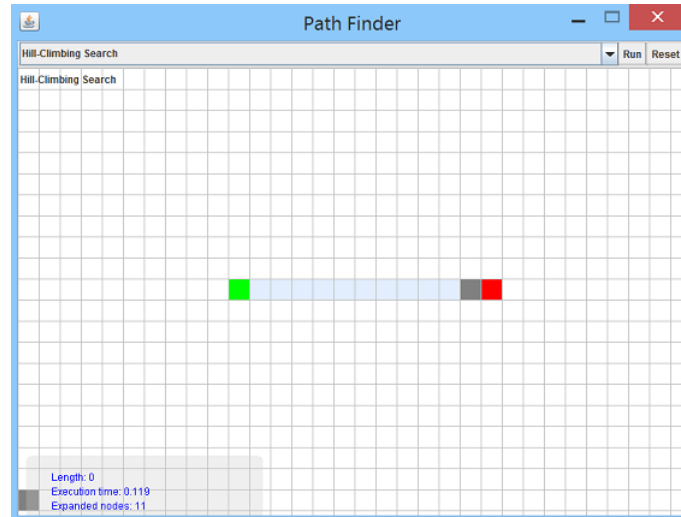


Figure 6: Hill-climbing search with a block between two points

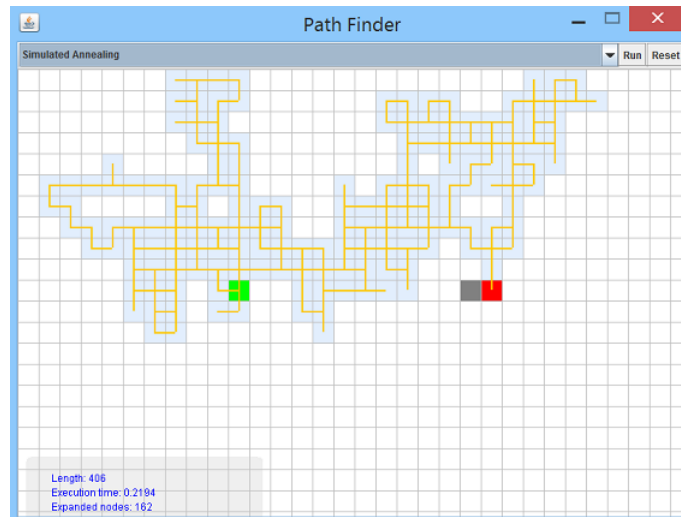


Figure 7: Simulated annealing with a block between two points

Simulated annealing is the most ineffective choice as per this example as the path length is 406 steps as shown. However, the CPU execution time and the expanded node are much less than uninformed search algorithms because of its randomness.

4 Conclusion

So the obvious question is "which algorithm should we use for finding paths on a game map?"

- If you want to find paths from or to all locations, use Breadth First Search.
- If you want to find paths to one location, use Greedy Best First Search or A*. We would prefer A* in most cases since it is optimal.

What about optimal paths? Breadth First Search is guaranteed to find the shortest path given the input graph. Greedy Best First Search is not. A* is guaranteed to find the shortest path if the heuristic is never larger than the true distance. As the heuristic becomes larger, A* turns into Greedy Best First Search.

What about performance? The best thing to do is to eliminate unnecessary locations in your graph. Reducing the size of the graph helps all the graph search algorithms. It also can be noticed that simpler queues run faster. Greedy Best First Search typically runs faster but doesn't produce optimal paths. A* is a good choice for most pathfinding needs.