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 games we often want to find paths from one location to another. Were not just trying to find the shortest distance; we also want to take into account travel time. Most path finding AI algorithms are designed for arbitrary graphs rather than grid-based games. Wed like to find something that can take advantage of the nature of a game map. There are some things we consider common sense, but that algorithms dont understand. We know something about distances: in general, as two things get farther apart, it will take longer to move from one to the other. We know something about directions: if your destination is to the east, the best path is more likely to be found by walking to the east than by walking to the west. On grids, we know something about symmetry: most of the time, moving north then east is the same as moving east then north. This additional information can help us make path finding algorithms run faster. To solve the path finding problem, we have implemented four major search algorithms namely A* Search, Greedy Best-First Search, Hill-Climbing Search, and Breadth-First Search. The search procedure takes place on a grid, so that we have an actual depiction of each of the algorithms to visualize them in terms of performance and characteristics.

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 agraphand to find theoptimal 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.

At its core, a pathfinding method searches agraphby starting at onevertexand exploring adjacentnodesuntil the destination node is reached, generally with the intent of finding the shortest route. Although graph searching methods such as abreadth-first searchwould 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 find the shortest path 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 heuristics for the informed search algorithms.

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 aqueuedata structure to store intermediate results as it traverses the graph, as follows:

1. Enqueue the root node
2. Dequeue a node 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 asearch algorithmwhich explores agraphby 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);
            }
        }
    }
}
```

2.4 A* search algorithm:

2.5 Hill-climbing algorithm:

2.6 Simulated annealing algorithm:

3 Results and Analysis

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 doesnt produce optimal paths. A* is a good choice for most pathfinding needs.

$$T_k = \frac{T_0}{1+\alpha \cdot k}, \quad \alpha > 0$$