Jonathan Peral Gort Param Manhas

CMPT 417 PROJECT REPORT

Introduction

This project set out to study the efficiency and optimality of various search algorithms in use of multi agent path finding in the gaming industry. With how advanced and complicated games are becoming, multi-agent path finding has become more important than ever, and has proved to be a necessity in most modern releases. To simulate such, we set out on implementing A* and Djkistra searches with various heuristics in a game of snake. We wanted to see how the performance of the algorithm would vary depending on the number of agents (the snakes) and what method of heuristics was used. In this report we will answer questions such as "Do the algorithms find the optimal solution?", "Which one performs better with more agents in the problem?", and more.

Implementation

Search Algorithms that were used in this project are the A* search which are given a set of agents, with their respective start position and goal positions.

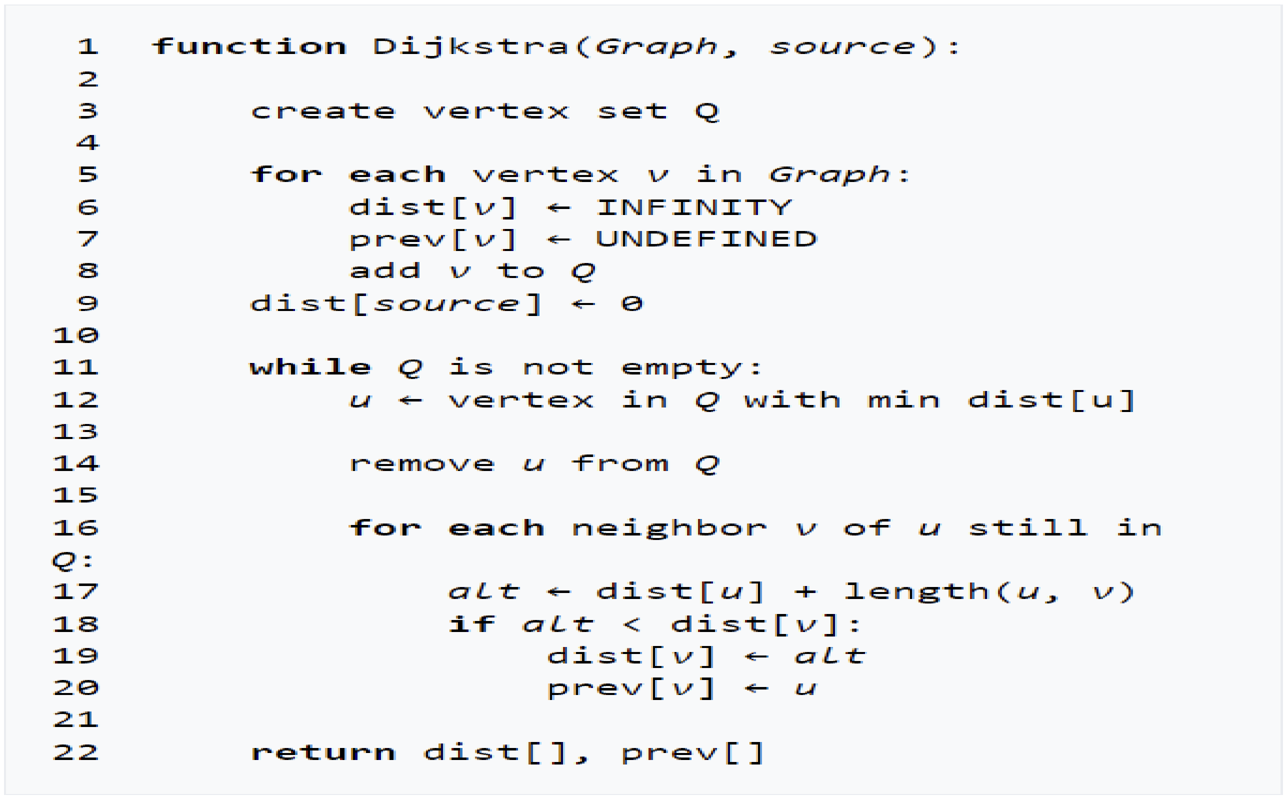
A* search pseudocode:

```
function reconstruct_path(cameFrom, current)
  total_path := {current}
  while current in cameFrom.Keys:
    current := cameFrom[current]
    total_path.prepend(current)
  return total_path
// A* finds a path from start to goal.
// h is the heuristic function. h(n) estimates the cost to reach goal from node n.
function A_Star(start, goal, h)
  // The set of discovered nodes that may need to be (re-)expanded.
  // Initially, only the start node is known.
  // This is usually implemented as a min-heap or priority queue rather than a hash-set.
  openSet := {start}
  // For node n, cameFrom[n] is the node immediately preceding it on the cheapest path from start
  // to n currently known.
  cameFrom := an empty map

  // For node n, gScore[n] is the cost of the cheapest path from start to n currently known.
```

```
gScore := map with default value of Infinity
gScore[start] := 0
// For node n, fScore[n] := gScore[n] + h(n). fScore[n] represents our current best guess as to
// how short a path from start to finish can be if it goes through n.
fScore := map with default value of Infinity
fScore[start] := h(start)
while openSet is not empty
  // This operation can occur in O(1) time if openSet is a min-heap or a priority queue
  current := the node in openSet having the lowest fScore[] value
  if current = goal
    return reconstruct_path(cameFrom, current)
  openSet.Remove(current)
  for each neighbor of current
    // d(current,neighbor) is the weight of the edge from current to neighbor
    // tentative_gScore is the distance from start to the neighbor through current
    tentative_gScore := gScore[current] + d(current, neighbor)
    if tentative_gScore < gScore[neighbor]
      // This path to neighbor is better than any previous one. Record it!
      cameFrom[neighbor] := current
      gScore[neighbor] := tentative_gScore
      fScore[neighbor] := tentative_gScore + h(neighbor)
      if neighbor not in openSet
        openSet.add(neighbor)
// Open set is empty but goal was never reached
return failure
```

Dijkstra’s search algorithm pseudocode:



```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u still in Q:
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

We tested Euclidean and Manhattan as our heuristics being that we determined that those would be the most adequate ones to determine the shortest path.

Methodology

We Strongly believe that by using the pathfinding demonstration tool there wouldn't be much need to test the algorithms in an actual game of snake, seeing that the nature of the demonstration is very similar to that of the game. In other words, the results will most likely be relatively similar.

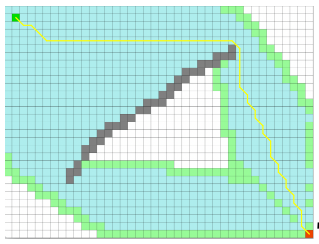
When analyzing the pathfinding demonstration, these were the settings used:

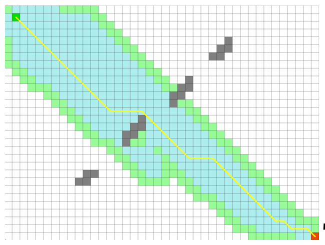
- Grid: 40 x 30
- Heuristic(s): Euclidean Distance, Manhattan Distance
- A path can be drawn diagonally unless there's at least one object interfering

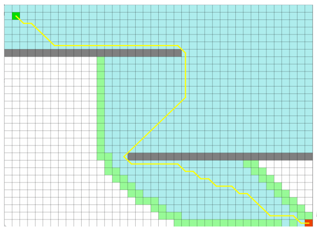
Experimental Setup

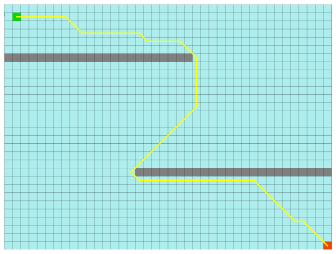
All work was performed in the MacOS operating system. Processing chip used is the Apple M1, with 8 cores and 8GB of RAM. The laptop model that was used in the research was a MacBook Air 2020. All code was ran using Java (OpenJDK 17.0.1)

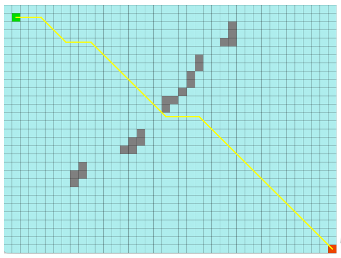
Experimental Results

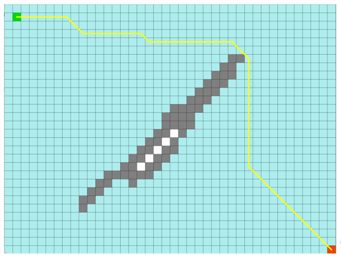

Time taken until a path is found: 20.45 seconds [A*, Euclidean Distance]

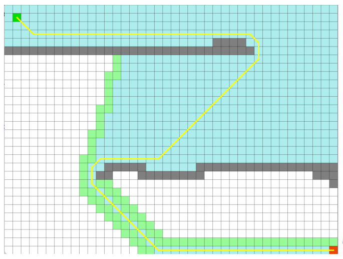

Time taken until a path is found: 09.19 seconds [A*, Euclidean]

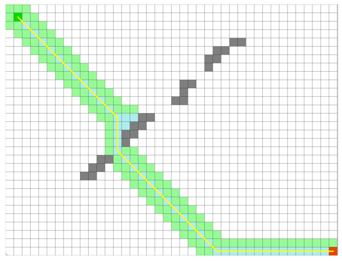

Time taken until a path is found: 18.55 seconds [A*, Euclidean Distance]

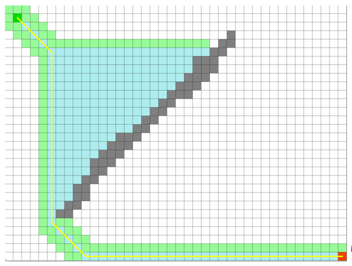

Time taken until a path is found: 27.72 seconds [Djikstra, Euclidean Distance]


Time taken until a path is found: 29.01 seconds [Djikstra, Euclidean Distance]


Time taken until a path is found: 28.47 seconds [Djikstra, Euclidean Distance]


Time taken until a path is found: 14.50 seconds [A*, Manhattan Distance]


Time taken until a path is found: 1.87 seconds [A*, Manhattan Distance]


Time taken until a path is found: 6.39 seconds [A*, Manhattan Distance]

The experiment consists of a grid and two separated nodes, green and red, the green node is the start node and the red is the goal node. The light green nodes mark the path in which the algorithm(s) traverse in search of the goal node. And for further complexity, we use the grey nodes to create obstacles for the search algorithm.

<u>A* Search vs. Djikstra's Search:</u>

We have measured time and gave all tested instances the same map with the same obstacles. As we can see A* search as a whole, with the 2 heuristics perform much better than the Dijkstra's search algorithm. As a result, we will further test other Heuristic functions with A* search and discard Dijkstra's algorithm.

<u>Euclidean vs. Manhattan:</u>

When comparing these two heuristics, Manhattan outperforms Euclidean, by far. We believe it performs better due to the fact that it is very cheap to calculate, and is very easily applicable and more accurate to our situation as it is being used in a grid structure.

Conclusion

Overall, A* search has shown to be a stronger algorithm for game development use, specifically 2D games, especially when dealing with a multi-agent situation in a game that is using a grid system. Many 2D games are implemented on a grid system, hence, based on our discovered results, we expect that A* search combined with the manhattan heuristic to be a very strong and common approach in the industry. We have shown it to be vastly superior and efficient in finding the target state. In contrast, Dijkstra proved to be a poor option compared to the other algorithms that we tried. We have not shown if it is efficient in finding the optimal solution and this is something that we would consider researching/testing in the future.

References

PathFinding Demonstration Application - GreenSlim96:

- https://github.com/GreenSlime96/PathFinding

A* Algorithm - Wikipedia:

- https://en.wikipedia.org/wiki/A*_search_algorithm

Dijkstra's Algorithm - Wikipedia:

- https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm