MINISTRY OF EDUCATION OF REPUBLIC OF MOLDOVA
Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics
Department of Software Engineering and Automatics

Report

on Artificial Intelligence Fundamentals
Laboratory Work nr. 2

Performed by: **Ejova Ecaterina**, FAF-172

Verified by: **Mihail Gavrilița**, asist. univ.

Chișinău, 2021

Contents

The Task

Find an example implementation of the A* algorithm and execute it on your computer. Answer the following questions:

1. What is A*?
2. What are 3 examples where A* could be used?
3. What are some alternatives to A*?
4. What is a heuristic?
5. What is a boid?

Solution Description

1. What is A*? The A* search algorithm is useful for finding the lowest cost path between two nodes (aka vertices) of a graph. The path may traverse any number of nodes connected by edges (aka arcs) with each edge having an associated cost. Each time A* enters a state, it calculates the cost, f(n) (n being the neighboring node), to travel to all of the neighboring nodes, and then enters the node with the lowest value of f(n). These values are calculated with the following formula:

   $$f(n) = g(n) + h(n)$$

   g(n) being the value of the shortest path from the start node to node n, and h(n) being a heuristic approximation of the node's value.

   For us to be able to reconstruct any path, we need to mark every node with the relative that has the optimal f(n) value. This also means that if we revisit certain nodes, we'll have to update their most optimal relatives as well. More on that later.

2. What are 3 examples where A* could be used?

2.1 Search Space

In any game environment, AI characters need to use an underlying data structure – a search space representation – to plan a path to any given destination. Finding the most appropriate data structure to represent the search space for the game world is absolutely critical to achieving realistic-looking movement and acceptable pathfinding performance.

2.2 In maps the A* algorithm is used to calculate the shortest distance between the source (initial state) and the destination (final state).

2.3 It is used in distance calculations, in mazes, in pathfiding from point A to point B.

3. What are some alternatives to A*?

   Dijkstra's Algorithm works by visiting vertices in the graph starting with the object's starting point. It then repeatedly examines the closest not-yet-examined vertex, adding its vertices to the set of vertices to be examined. It expands outwards from the starting point until it reaches the goal. Dijkstra's Algorithm is guaranteed to find a shortest path from the starting point to the goal, as long as none of the edges have a negative cost.

   The Greedy Best-First-Search algorithm works in a similar way, except that it has some estimate (called a heuristic) of how far from the goal any vertex is. 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 Dijkstra's Algorithm because it uses the heuristic function to guide its way towards the goal very quickly. For example, if the goal is to the south of the starting position, Greedy Best-First-Search will tend to focus on paths that lead southwards.

4. What is a heuristic?

   Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal.

5. What is a boid?

   Boids is an artificial life simulation originally developed by Craig Reynolds. Is generic simulated flocking creature. The aim of the simulation was to replicate the behavior of flocks of birds and fish.

   How does it work? Each of the boids (bird-oid objects) obeys three simple rules:

   1. Coherence Each boid flies towards the the other boids. But they don't just immediately fly directly at each other.

   2. Separation Each boid also tries to avoid running into the other boids. If it gets too close to another boid it will steer away from it.

   3. Alignment Finally, each boid tries to match the vector (speed and direction) of the other boids around it.

Conclusions

A* is a very powerful algorithm with almost unlimited potential. The A* search algorithm, builds on the principles of Dijkstra's shortest path algorithm to provide a faster solution when faced with the problem of finding the shortest path between two nodes. It achieves this by introducing a heuristic element to help decide the next node to consider as it moves along the path.

References

[1] Ejova Ecaterina. Source code for the laboratory work. Accessed February 18, 2021. X.

[2] Introduction to A* http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.

[3] What is the A* Algorithm and How does it work? https://www.edureka.co/blog/a-search-algorithm/

Appendix

Listing 1: A* implementation in Python

```python
class Node():
    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position
        self.g = 0
        self.h = 0
        self.f = 0
    def __eq__(self, other):
        return self.position == other.position
def astar(maze, start, end):
    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end)
    end_node.g = end_node.h = end_node.f = 0
    open_list = []
    closed_list = []
    open_list.append(start_node)
    while len(open_list) > 0:
        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index
        open_list.pop(current_index)
        closed_list.append(current_node)
        if current_node == end_node:
            path = []
            current = current_node
            while current is not None:
                path.append(current.position)
                current = current.parent
            return path[::-1]
        children = []
        for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (1, 1)]:
            node_position = (current_node.position[0] + new_position[0], current_node.position[1] + new_position[1])
            if node_position[0] > (len(maze) - 1) or node_position[0] < 0 or node_position[1] > (
                    len(maze[len(maze) - 1]) - 1) or node_position[1] < 0:
                continue
            if maze[node_position[0]][node_position[1]] != 0:
                continue
            new_node = Node(current_node, node_position)
            children.append(new_node)
        for child in children:
            for closed_child in closed_list:
                if child == closed_child:
                    continue
            child.g = current_node.g + 1
            child.h = ((child.position[0] - end_node.position[0]) ** 2) + (
                    (child.position[1] - end_node.position[1]) ** 2)
            child.f = child.g + child.h
            for open_node in open_list:
                if child == open_node and child.g > open_node.g:
                    continue
            open_list.append(child)
def main():
    maze = [[0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 0, 0]]
    graph = [[0, 1, 0, 0, 0, 0],
             [1, 0, 1, 0, 1, 0],
             [0, 1, 0, 0, 0, 1],
             [0, 0, 0, 0, 1, 0],
             [0, 1, 0, 1, 0, 0],
             [0, 0, 1, 0, 0, 0]
             ]
    start = (0, 0)
    end = (5, 5)
    end1 = (5, 5)
    path = astar(maze, start, end)
    print(path)
    path1 = astar(graph, start, end1)
    print(path1)
if __name__ == '__main__':
    main()
```

Output:

```
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 3), (5, 4), (5, 5)]
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5)]
```