ARTIFICIAL INTELLIGENCE (E016330)
ALEKSANDRA PIZURICA
GHENT UNIVERSITY
2018/2019 AY

Assistants: Angel Lopez, Laurens Meeus, Nina Žižakić

Practical Session 1: Path planning with A*

1 Introduction

Path planning is concerned with the problem of moving an entity from an initial configuration to a goal configuration. The resulting route may include intermediate tasks and assignments that must be completed before the entity reaches the goal configuration. Path planning algorithms can be classified as either global or local. Global path planning takes into account all the information in the environment when finding a route from the initial position to the final goal configuration. Local planning algorithms are designed to avoid obstacles within a close vicinity of the entity; therefore, only information about nearby obstacles is used. This project will refer to global path planning algorithms where the entire path is generated from start to finish before the entity makes its first move.

Typically, planning operations require searching through (a subset of) all the possible routes to find the most optimal or efficient route among all possibilities. For instance, in a robotics application that must move a robot between five locations in the shortest amount of time, the path planner must perform a search to determine in what order to complete these tasks. Conventional AI search algorithms in conjunction with graph theory are used to accomplish this task. The graph nodes are placed at choice points and edges connect each node in the graph. A choice point is anywhere in the graph where a decision must be made as to where to go next. The search begins at the choice point placed at the initial location, or node, and ends when all choice points have been reached. Exhaustive, simplistic search algorithms such as breadth-first and depth-first suffer from memory issues or suboptimal results. In practice, more advanced approaches, incorporating heuristic assumptions tend to find optimal solutions within a reasonable amount of time and better memory requirements.

A popular example of this is the A* algorithm. This algorithm keeps track of the current cheapest path from the start node to an arbitrary node n by $g(n)$. Furthermore, it uses a heuristic function $h(n)$ to estimate the distance from a node n to the goal node. As a consequence, the estimated cost of the cheapest solution through node n will be given by:

$$f(n) = g(n) + h(n). \tag{1}$$

Along any path from the start node, f will always be non-decreasing provided that the heuristic function is admissible (i.e. it never overestimates the actual cost to get to the goal node). Assuming this, it can be proven that the first solution found must be the optimal one, because all subsequent nodes will have a higher f-cost. Furthermore, because it makes the most efficient use of the heuristic function, no search that uses the same heuristic function $h(n)$ and finds optimal paths will expand fewer nodes than A*. An easy heuristic for $h(n)$ is the straight-line (Euclidean) distance between n and the goal. This distance is always an underestimation of the real path between n and the goal.

2 Assignment

In this practical session, we will apply the A* algorithm on a real-world New York traffic network data set. You can download this data and additional Python source files via Minerva under "Documents/Practicals/1 A-star search/a-star.zip".

2.1 Loading the data

Load the provided distance graph and coordinate data sets (`USA-road-d.NY.gr` and `USA-road-d.NY.co`, respectively) into a graph object and an array, using the `readGraph` and `readCoordinates` functions from `graph.py`. Verify the number of vertices (or nodes) and edges are contained within the graph using the `Graph` and `Vertex` class functions. You can check your outcome at the top of the `USA-road-d.NY.gr` text file.

2.2 A* implementation

Implement the A* algorithm in the `a_star_search` function, which will take a graph object, a start and goal node as input, and return an optimal path (i.e. a path with minimal travel distance) from the start to the goal node. The pseudocode for this algorithm is as follows:

```
function cameFrom = A*(start,goal)
    closed = {}
    open = {start}
    cameFrom = {}

    g = map with default value of ∞
    g(start) = 0
    f = map with default value of ∞
    f(start) = g(start) + h(start, goal)

    while open is not empty
    current = the node in open having the lowest f value
    if current = goal
        return cameFrom

    open.remove(current)
    closed.add(current)
    for each neighbor of current
        if neighbor in closed
            continue
```

```
        tentative_g_score = g(current) + dist(current,neighbor)
        if neighbor not in open
            open.add(neighbor)
        else if tentative_g_score >= g(neighbor)
            continue

        cameFrom(neighbor) = current
        g(neighbor) = tentative_g_score
        f(neighbor) = g(neighbor) + h(neighbor, goal)
```

Use the Euclidean distance as a heuristic function:

$$d_E(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}. \tag{2}$$

Additionally, the provided priority queue class definition will be useful in the A* implementation. Generate random start and goal nodes as follows:

```
import random
random.seed(gn)
start = random.randint(0,N)
goal = random.randint(0,N)
```

where `gn` is your groupnumber and `N` is the number of vertices in the graph. Give the shortest route (including the total distance) from `start` to `goal`. You can verify your results in the `distances.txt` file on Minerva.

2.3 Shortest time path

In most GPS applications, the user prefers the shortest path with the respect to time, rather than distance. The file `USA-road-t.NY.gr` contains the same graph structure as the one we have been working on, however, the weights $w_{i,j}$ are now stating the required transportation time to get from node i to node j. Compute the minimal time path for the same start and goal nodes as in the previous section. What are your conclusions? Is this path optimal? Why (not)?

2.4 Alternative heuristics

An alternative heuristic function is the Manhattan distance:

$$d_M(\mathbf{x}, \mathbf{y}) = \sum_{i=1}^{n} |x_i - y_i|. \tag{3}$$

This distance function is characterized by the fact that it measures distances along horizontal and vertical lines. Compare your results of both the Euclidean and Manhattan distance for the previously mentioned start and stop node. What are your conclusions?

2.5 Submission

The project is done in groups of maximum two students. A report (2-3 pages) should be written which consists of a detailed explanation of the A* algorithm, your results and conclusions. The source file a_star.py (and potential external source files) should print out a list of subsequent nodes corresponding to the requested shortest path (according to the Euclidean heuristics and with respect to minimal distance) at runtime.

The report and source code should be submitted as a zip-file in your group on Minerva. The file should be named after both authors and specify the project subject (e.g. Frodo Baggins & Samwise Gamgee: a-star-fbaggins-sgamgee.zip). Deadline for the submission is **Friday, November 2, 2018 (23:59)**.