**LAB 2 REPORT**

**Introduction to Artificial Intelligence**

# Group Information

|  |  |
| --- | --- |
| **Name** | **Student ID** |
| Nguyễn Lê Hùng | 22127135 |
| Trần Văn Quyết |  |

# Accomplished requirements

**Requirements overview**

|  |  |
| --- | --- |
| **Stage** | **Progress** |
| Stage 1 | Successfully implementing the A\* algorithm to discover a path from the start point (S) to the goal point (G) and documented the algorithm, along with its execution process, across three distinct maps. |
| Stage 2 | Successfully implementing three different algorithms, including A\*, greedy, and UCS (Uniform Cost Search). Report and analyze the differences observed while running these three algorithms. |
| Stage 3 | Successfully implementing the algorithm to find the path with the minimum total cost from S, passing through pickup points, to G. |
| Stage 4 | Polygons can be moved at a speed of h units per second. The simplest way of movement is to traverse a small distance, ensuring no overlap with other polygons. Implement at least one algorithm for this movement.. |
| Stage 5 | Visualize the model in three-dimensional space (3D). |

**Technical specifications**

|  |  |
| --- | --- |
| **Tools** | **Role** |
| Python version >= 3.1 | Program structure, backbone bootstrapper for other modules. |
| Matplotlib version >= 2.1 | Visualization |
| Numpy | Data structures and list-wise arithmatic |

# Algorithms and Implementations

## **Algorithms**

### A star

Successfully implementing the A\* algorithm, a pathfinding algorithm used to find the shortest path between two points on a graph. The project makes use of the BestFirst function and provides a specific evaluation function (eval) tailored for A\* employing the **Pythagorean** formula.

A\* is like Dijkstra’s Algorithm in that it can be used to find a shortest path. A\* is like Greedy Best-First-Search in that it can use a heuristic to guide itself.

The secret to its success is that it combines the pieces of information that Dijkstra’s Algorithm uses (favoring vertices that are close to the starting point) and information that Greedy Best-First-Search uses (favoring vertices that are close to the goal). In the standard terminology used when talking about A\*, g(n) represents the exact cost of the path from the starting point to any vertex n, and h(n) represents the heuristic estimated cost from vertex n to the goal.

Each time through the main loop, it examines the vertex n that has the lowest

**f(n) = g(n) + h(n).**

Helper function: def eval(grid: Graph, s, e, u, n), def BestFirst(dat: InputData, grid: Graph, f, evalOrder=0)

### Breadth-First Search (BFS)

Breadth-First Search is an algorithm for traversing or searching tree or graph data structures. BFS explores all neighbor nodes at the current level before moving to the next level.

Similar to Dijkstra's Algorithm, BFS prioritizes finding a path but focuses on finding any valid path in the shortest number of steps (levels) from the starting node to the goal. It achieves this by employing a FIFO (First-In-First-Out) queue data structure. Nodes are expanded level by level, ensuring all shallower options are explored before diving deeper.

In this implementation, the eval function assigns a uniform cost of 0 to all nodes. This, combined with the queue structure, effectively prioritizes nodes based on their depth from the starting position. BFS is guaranteed to find the shortest path if all edge costs in the graph are equal.

Helper function: def eval(grid: Graph, s, e, u, n), def BestFirst(dat: InputData, grid: Graph, f, evalOrder=0)

### Depth-First Search (DFS)

DFS prioritizes exploring a single path as deeply as possible before backtracking and trying alternative branches. It utilizes a LIFO (Last-In-First-Out) stack data structure, expanding the most recently added neighbor node first.

While DFS can be efficient in finding a solution path quickly, especially if it exists near the top of the search tree, it doesn't guarantee finding the shortest path. Unlike BFS, it might overlook shorter paths by getting stuck in deep, dead-end branches.

### Uniform Cost Search (UCS)

Uniform Cost Search is an algorithm designed to find the path with the minimum total cost from a starting node to the goal in a weighted graph. Similar to BFS and Dijkstra's algorithm, UCS utilizes a priority queue to prioritize expanding nodes. However, unlike BFS, UCS prioritizes nodes based on their accumulated cost g(n) from the starting node, not just their depth

In this specific case, due to the relatively uniform edge costs in the provided implementation, UCS behaves identically to BFS. However, for graphs with varying edge costs, UCS would prioritize expanding lower-cost paths, ensuring it finds the shortest path in terms of total cost.

Helper function: def eval(grid: Graph, s, e, u, n), def BestFirst(dat: InputData, grid: Graph, f, evalOrder=0)

### Greedy Best-First Search

Greedy Best-First Search aims to find a path to the goal by prioritizing nodes that appear closest to the goal based on a heuristic function.

Similar to A\*, GBFS utilizes a priority queue to manage node exploration. However, unlike A\*, it only considers the heuristic estimate h(n) to guide its search. The eval function in GreedyBFS return the estimated cost which is the **Euclidean distance** to the goal h(n). Nodes with lower estimated total costs **f(n) = h(n)** are prioritized for expansion.

While GBFS can be efficient in finding solutions quickly, especially if the heuristic is accurate, it doesn't guarantee finding the shortest path. The heuristic might be misleading, leading the search down paths that appear promising initially but end up being longer in the end.

Helper function: def eval(grid: Graph, s, e, u, n), def BestFirst(dat: InputData, grid: Graph, f, evalOrder=0)

### Algorithms evaluation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Data Structure | Node Expansion Order | Stop Condition | Comments |
| BFS (Breadth-First Search) | Queue (FIFO) | Explores all neighbors of a node before moving outward (level-by-level) | Reaches the goal node or the queue becomes empty | Guaranteed to find the shortest path if one exists (unweighted graphs). Might explore more nodes than UCS, GBFS, or A\*, especially in complex graphs. |
| DFS (Depth-First Search) | Stack (LIFO) | Explores one path deeply until it reaches the goal or a dead end, then backtracks | Reaches the goal node or the stack becomes empty | Space efficient but not guaranteed to find the optimal path. Might explore more nodes than UCS, GBFS, or A\*, especially in complex graphs. |
| UCS (Uniform Cost Search) | Priority Queue (ordered by g-cost) | Expands nodes with the lowest total cost (g-cost) from the starting position | Reaches the goal node or the queue becomes empty (and all reachable nodes have been explored) | Guaranteed to find the optimal path in weighted graphs. UCS might explore more nodes than algorithms like A\* (informed search) if the heuristic in A\* is very accurate. |
| GBFS (Greedy Best-First Search) | Priority Queue (ordered by h-cost) | Expands nodes with the lowest estimated total cost (f-cost = h-cost, where h-cost is a heuristic) | Reaches the goal node or the queue becomes empty | Can be faster than A\* with a good heuristic, but might miss the optimal path. |
| A\* Search | Priority Queue (ordered by g-cost + h-cost) | Expands nodes with the lowest f-cost. | Reaches the goal node or the queue becomes empty | A\* efficiently searches for the shortest path by prioritizing exploration based on a good estimate of remaining cost (h-cost), but it can miss the optimal path if that estimate is inaccurate. |

## 2. Implementations

### Graph

A graph is represented by the Graph class, which is defined in the *graph* module. It consists of a list, which contains the datas related to each Node object in the graph; the graph’s dimensions; and the intial locations of shapes’ anchor point.

A graph object consists of various helper functions to help translate from 2D axis coordinates to 1D axis coordinate. Grid data is represented as a 1D list in order to boost performance and allow various pointer arithmetic. Additionally, bounding check and movability test allow algorithms implementation to be shorter.

Each graph will have a method to reset its nodes data to the default value via *partial\_reset*. This helper method enables the ability to run multiple instances of algorithms on the same graph without reconstructing it.

*dynamic\_geo(direction, h) is* an additional feature, which moves all shapes on the graph *h*  distance in *direction* (as specified in the README documentation).

### Visualization

The program primarily uses matplotlib to visualize algorithm data. The module *visualize* is responsible for all visualization process.

By abstractionally programming, the soul entry for displaying a graph is via *visualize.show\_graph(graph.Graph, input.InputData, time = 0.0, show\_expansion = False)* signature.

The method takes in a graph and InputData as its first and second argument. The third argument, *time*, is optional and is passed as the running time of the algorithm (if exsisted). And *show\_expansion* flag displays other nodes which had been visited and the frontier.

Another main feature of the *visualize* module is animation. This feature allows the re-render of a graph list based on the current frame in a Round-Robin type order rotation, as well as dynamically update the graph at the end of each frame.

The feature is accessed via

updatable(graph\_list: list[Graph], input\_list: list[InputData], callback, init\_time=0.0, interval=1000, show\_expansion=False):

Which takes in a list of graphs to display as the first argument, and the corresponding InputData as it’s second argument. *callback* should be passed a function with the following signature:

(function) def callback(

graph: Graph,

input\_data: InputData

) -> float

This function will be called at the end of each frame and can be used to update the graph’s data through various mean, i.e. the *dynamic\_geo* method of the graph. The new data will be displayed in the next time the graph is rendered.

### Algorithms implementation

Algorithms are primarily implemented in the *algorithm* module.

5/6 implemented algorithms is based on a modified structure of the best first search algorithm. Each algorithm provides its own evaluating function and run the backbone Best first search function.

In order to generalize the backbone call, the program allows the control of goal test position, either when expanding a node, or when the node is generated, or none at all (in the case of Dijkstra algorithm).

Best first search generally use a Priority Queue data structure, allowing the algorithm iterations to run in O(nlog(n)) time. Though there is limitation. That is, in order to update a frontier node’s cost, we have to re generate that node with a new cost value and push that node back in to the queue without deleting the previous node. This causes memory issue on large graph because nodes might be added at the maximum of 4 times if there are multiple shorter path to it.

**For requirement 3:** after careful review of the problem, the group has concluded the problem to fall into the Open Traveling salesperson problem (Open TSP), which might be NP-Hard in a 2D grid graph.

So in order to increase clarity (because implementing the algorithm to solve it is quite tiddious), we have extracted the required codes into another module: *tsp*.

The general approach is to use Dijkstra on each checkpoints (including the start and the end node) in order to construct a graph in which edges between vertices (checkpoints) are the lowest path between each pair of vertices. After that, we solve the TSP problem through the use of dynamic programming, specifically a modified Held – Karp algorithm, which has a time complexity of O(2nn2) that we deemed acceptable for the current problem bound. The algorithm returns the total cost, as well as the order in which we will move through the checkpoints.

**Held – Karp:**

*This is the core algorithm for solving the TSP. It uses dynamic programming with memoization to find the shortest route that visits all checkpoints (including start and end points) exactly once. The function held\_karp takes a distance matrix as input and returns the optimal cost and path. (Held, M., & Karp, R. M. (1970). A dynamic programming approach to sequencing problems. Journal of the Society for Industrial and Applied Mathematics, 11(1), 196-210).*

**Reference:** [Github](https://github.com/CarlEkerot/held-karp)

The path is then reconstructed based on the order through the most efficient algorithm: A\*.

# Test Runs and Notes

**Presentation of Map:**

* Cost of diagonal move = 1.41 cost of horizontal/vertical move
* All nodes are initially white
* Expanded nodes are red, unexpanded nodes are green
* Obstacles/polygons are black
* Walls of map are grey
* Start and goal nodes are pink
* Solution path is highlighted in blue
* (For Stage 3 Maps) Midpoints are green

## A\*:

* **Path existence scenerio**

**A pixelated image of a check mark

Description automatically generated**

* **Branching Path scenario**

**A pixelated image of a snake

Description automatically generated**

* **Path existence scenario**

**A screenshot of a computer game

Description automatically generated**

* **BFS**

|  |  |
| --- | --- |
| **A pixelated picture of a red square with green and red squares  Description automatically generated** | **A pixelated image of a letter  Description automatically generated** |

* **DFS**

|  |  |
| --- | --- |
| A screenshot of a computer game  Description automatically generated | A pixelated image of a square with a square and a square with a square and a square with a square with a square and a square with a square with a square and a square with a  Description automatically generated |
|  |  |

* **GBFS**

|  |  |
| --- | --- |
| A pixel art of a guitar  Description automatically generated | A pixelated logo with a letter u  Description automatically generated with medium confidence |

* **UCS**

|  |  |
| --- | --- |
| A screenshot of a video game  Description automatically generated | A pixelated picture of a red square with a white line  Description automatically generated |

Test Result:

|  |  |
| --- | --- |
| **Map 1: Solution is a straight path**  **Map 2: Solution is a branching path**  **Map 3: No solution exists (same for all)** | **A pixelated square with red and white squares  Description automatically generated** |

* **Held – Karp**

|  |  |
| --- | --- |
| **Map 1: Path existence scenerio 1 (with 4 midpoints)** | A screenshot of a game  Description automatically generated |
| **Map 2: Path existence scenario 2 (with 1 midpoint, branching path)** | A pixelated image of a letter  Description automatically generated |
| **Map 3: No path exists scenario** | A pixelated red square with white and green numbers  Description automatically generated |

1. **Tự đánh giá**
2. **Tài liệu tham khảo**

*Artificial Intelligence: A Modern Approach" by Stuart Russell and Peter Norvig (Chapter 3.3.2) ([URLaima3e stanford ON Stanford University people.cs.stanford.edu])*

*https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html*