Graph algorithms Homework

A2 Computer science

Due date: Tuesday, December 2, 2025

Goal of the Assignment

In this assignment, you will review, deepen your knowledge of, and implement two different graph algorithms: Dijkstra's algorithm for shortest paths and it's variant, A* search algorithm. You will implement both algorithms in Python and test them on various graphs

Provided files

Please, find these files attached to the assignment in google classroom:

- `graph_algorithms.py` - A starting point for your project, you will write your code in this file.
- `graph.py` - a Python file containing a basic implementation of a graph data structure.
- `test_graphs.py` - a Python file containing some test graphs to use with your implementations.
- `test_graph_algorithms.py` - a Python file that will run some tests on your implementations, to be called when your assisgnment is done.

You are strongly encouraged to read through the provided files before starting your implementation. You will need the methods provided in `graph.py` to implement the algorithms.

Workflow

For each algorithm, you will follow the description in this file to implement intermediate function as well as the final, complete function. You may add other intermediates if you feel like you need them. When completed, you should test your final function by running test_graph_algorithms.py.

Your assignment will be completed when test_graph_algorithms.py says so. This is a learning opportunity, that would be wasted were you to use AI, so keep it clean, for your own sake.

1 Dijkstra's Algorithm

1.1 Plain English Description

Dijkstra's algorithm is a method for finding the shortest path from a starting node to an end node in a weighted graph.
Here is a step by step description of how it works:

1. Start with a graph where each edge has a weight (cost).
2. Assign a temporary distance value to every node: set it to zero for the initial node and to infinity for all other nodes. Mark all nodes as unvisited.
3. Set the initial node as the current node.
4. For the current node, consider all of its unvisited neighbors and calculate their temporary distances through the current node. If this distance is less than the previously recorded tentative distance, update it, and record the current node as previous for the updated neighbors.
5. Once all neighbors have been considered, mark the current node as visited. A visited node will not be checked again.
6. If the destination node has been marked visited or if the smallest tentative distance among the unvisited nodes is infinity, then stop. The algorithm has finished.
7. Otherwise, select the unvisited node with the smallest tentative distance as the new current node and repeat from step 4.

1.2 Implementation

Implement Dijkstra's algorithm in the `dijkstra` function in `graph_algorithms.py`. It should take three parameters:

- `graph` - an instance of the `Graph` class from `graph.py`
- `start_node` - the name of the starting node (string)
- `end_node` - the name of the destination node (string)

It should return a list containing two elements:

- The total cost (weight) of the shortest path.
- A list of node names representing the shortest path from `start_node` to `end_node`.

2 A-star Algorithm

2.1 Plain English Description

A* (A-star) is a **best-first** search algorithm that finds the lowest-cost path from a start node to a goal node by combining the **actual cost to reach** a node (g) with a **heuristic estimate** (h) of the cost from that node to the goal. The algorithm balances **exploration** and **exploitation** by **prioritising nodes with the lowest estimated total cost** f(n) = g(n) + h(n).
Here is a step by step description of how it works:

1. Provide a heuristic function h(n) that estimates the cost from node n to the goal. The heuristic should be admissible (never overestimates the true cost).

2. Initialize **two sets**: the `open_set` (nodes to be evaluated, initially containing the start node) and the `closed_set` (already evaluated nodes, initially empty).

3. For every node, maintain `g(n) =` the best known cost from the start node to n. Set `g(start) = 0` and `g(other) = infinity`. Also maintain a dictionary `came_from` to reconstruct the path.

4. While `open_set` is not empty:

   (a) Select the node in `open_set` with the lowest `f(n) = g(n) + h(n)`. This is the `current_node`.

   (b) If the `current_node` is the goal, reconstruct and return the path using `came_from` and the total cost g(goal).

   (c) Move the `current_node` from the `open_set` to the `closed_set`.

   (d) For each neighbor of the `current_node`:

      i. If the neighbor is in the `closed_set`, skip it.

      ii. Compute `tentative_g = g(current) + cost(current, neighbor)`.

      iii. If `tentative_g` is less than the recorded `g(neighbor)`, update `came_from[neighbor]` = `current_node`, `g(neighbor) = tentative_g`, and set `f(neighbor) = g(neighbor) + h(neighbor)`. If the neighbor is not in the `open_set`, add it.

5. If `open_set` is empty and `goal` was not reached, there is no path.

2.2 Implementation

Implement A* in the `astar` function in `graph_algorithms.py`. It should take four parameters:

- `graph` - an instance of the `Graph` class from `graph.py`
- `start_node` - the name of the starting node (string)
- `end_node` - the name of the destination node (string)
- `heuristic` - a function that takes two node names (current, goal) and returns an estimated cost (non-negative number)

It should return a list containing two elements:

- The total cost (weight) of the shortest path found (or `float('inf')` if no path exists).
- A list of node names representing the shortest path from `start_node` to `end_node` (empty list if no path).