P1: Dijkstra's in a Dungeon

Problem Description

- Program a pathfinding algorithm which returns the path with minimal cost between two waypoints in a maze.
- Modify your solution to find the distance/cost to each reachable cell from an initial position.
- Create a custom maze to test your implementation on!

Learning Objectives

- Revisit your past knowledge of Dijkstra's shortest path algorithm.
- Practice building the graph to be searched on the fly.
- Practice using Python's high-level data structures such as lists and dicts.

Requirements

- Implement a function to compute the adjacent cells to a given cell on the level map. It should allow movement in 8 directions on the grid, including the diagonals. The cost function for horizontal and vertical moves is covered at the end of this document. Movement should only allowed between "spaces" in the level file (not "walls").
- Implement a version of Dijkstra's shortest path algorithm between a given pair of cells, returning the path (including the source and destination cells). The algorithm should stop searching as soon as the destination cell is found (not exploring the whole graph if it is not needed). If no path is possible, the algorithm should explicitly signal this (by returning None, an empty path, or raising an appropriately named exception).
- Modify your implementation to return cost to each reachable cell from a given origin cell. You will test this on your custom maze.

Grading Criteria

- Does the algorithm perform correctly?
- Do the paths returned appear to actually be shortest paths?
- Does the algorithm calculate correct costs for all reachable cells?
- Was a new and interesting example map created?

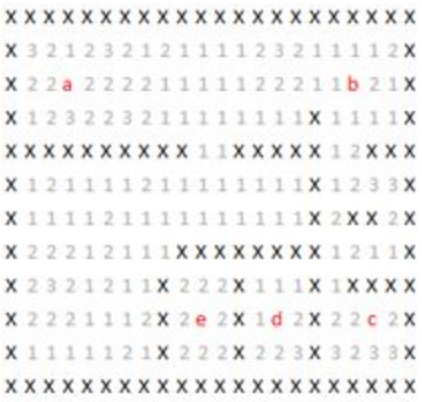
References

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

    - Suggested differences from the pseudo-code from Wikipedia:
        - Skip the entire initialization loop, instead use dicts:
            - “dist = {}” and “dist[state] = better_distance”
            - “prev = {}” and “prev[state2] = state1”
        - Because dist will not be defined for unvisited states, the expression “alt < dist[v]” must be implemented as “v not in dist or alt < dist[v]” or “alt < dist.get(v,alt+1)”.
        - Use Python’s heapq module to implement the priority queue. The queue will simply be a Python list containing tuples (distance-and-state pairs), but you’ll use the heapq library to add and remove elements from it. https://docs.python.org/2/library/heapq.html
        - Instead of returning the “dist” and “prev” tables (dicts), recover a specific shortest path and return it instead. Represent it as a list of states that starts with the source state and ends with the destination state.
- Using heapq for priority queue operations:

```
from heapq import heappush, heappop
queue = [] # Just a plain list
heappush(queue, (2, 'a')) # enqueuing some pairs
heappush(queue, (42,'b'))
heappush(queue, (1, 'c'))
p1, x1 = heappop(queue) # dequeuing some pairs
p2, x2 = heappop(queue)
p3, x3 = heappop(queue)
assert [x1, x2, x3] == ['c','a','b']
assert [p1, p2, p3] == [1, 2, 42]
assert queue == []
```

Example Level File (example.txt)

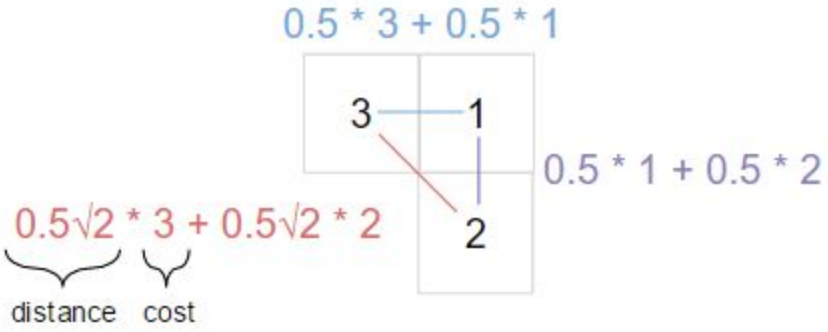


Key:

- X – an impassable wall
- a,b,c,d,e – waypoints
- 1,2,3 – the costs of moving through the open spaces of the maze (Note: all waypoints are assumed to have a cost of 1)

Path Cost Computation

Each open space is associated with a cost. The weight of an edge linking two cells is the sum of the distances traveled in each cell, as weighted by the cells' respective costs, as shown in the image below. Here we have computed the costs of transitioning between the three cells shown. Note: Euclidean distance is used for the distance between cells, resulting in a longer distance for diagonal movement.



Support Functions

The "load_level" function provided in the "p1_support" module returns a dictionary:

```
from p1_support import load_level
level = load_level('example.txt')
list(level.keys()) # --> ['walls', 'spaces', 'waypoints']
level['walls'] # --> {(0, 0), (0, 1), (0, 2), (0, 3), … }, a set
```

```
level['spaces'] # --> {(7, 3): 2, (6, 9): 2, (12, 1): 1, … }, a dictionary
level['waypoints'] # --> {'a': (3, 2), 'b': (18, 2), 'c': (19, 9), …}, a
dictionary
```

Submission Instructions

Submit a zip file named in the form of “Lastname-Firstname-P1.zip” containing:

- Your version of p1.py. Don’t change the name.
- A text file output of the path between waypoints (a) and (d) in test_maze.txt. Call this “test_maze_path.txt”.
- A new maze of your own creation, called “my_maze.txt”. Include at least four waypoints, (a)(b)(c)(d).
- A csv file (“my_maze_costs.csv”) containing the costs to all reachable cells from waypoint (a).

Submission Link