P1: Dijkstra's in a Dungeon

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.
- Understand the ASCII-art dungeon map format that we'll be using throughout the quarter.

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 for horizontal and vertical moves should be 1. The cost for diagonal moves should be sqrt(2). (Find sqrt in the Python math module). 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).
- Create the top level logic of your program so that you can demonstrate your algorithm on any given map file and pair of waypoints. It should be possible to change these values either via command line arguments or changing a single line of code.
- Create your own interesting map that is larger than the example map, and be prepared to show us why that map is interesting. (Does it test edge cases of your algorithm? Is it based on a map you've seen elsewhere?) Have fun being a level designer.

Grading Criteria

(equal weight for each question)

- Can filenames and waypoints be changed easily enough to demonstrate functionality?
- Does the algorithm perform correctly when there exists a path?
- Does the algorithm perform correctly when no path is possible?
- Do the paths returned appear to actually be shortest paths?
- Upon inspection of the code, does it appear correct? (This includes computing the graph on the fly.)
- Upon inspection of the code, is early termination implemented?
- Was a new and interesting example map created?

Code Sketch

Download the p1_support module here: https://drive.google.com/a/ucsc.edu/file/d/0B-PPiU3Ga8Z7Rk1RTGo5eHpseUU/view?usp=sharing

Download this example code here: https://drive.google.com/a/ucsc.edu/file/d/0B-PPiU3Ga8Z7bXJVNXVSeWcwOGs/view?usp=sharing

```
from p1_support import load_level, show_level
from math import sqrt
from heapq import heappush, heappop

def dijkstras_shortest_path(src, dst, graph, adj):
      raise NotImplementedError

def navigation_edges(level, cell):
      raise NotImplementedError

def test_route(filename, src_waypoint, dst_waypoint):
      level = load_level(filename)

      show_level(level)

      src = level['waypoints'][src_waypoint]
      dst = level['waypoints'][dst_waypoint]

      path = dijkstras_shortest_path(src, dst, level, navigation_edges)

      if path:
            show_level(level, path)
      else:
            print "No path possible!"

if __name__ ==  '__main__':
      import sys
      _, filename, src_waypoint, dst_waypoint = sys.argv
      test_route(filename, src_waypoint, dst_waypoint)
```

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.
- A similar IPython Notebook implementation for breadth-first search (BFS):
    - http://nbviewer.ipython.org/urls/dl.dropbox.com/s/7oo0eu4e0p6fwl4/BFS%20Example.ipynb
- The depth-first search (DFS) code we developed live in class: https://drive.google.com/a/ucsc.edu/file/d/0B-PPiU3Ga8Z7ZlEwbEhtZjNTUGs/view?usp=sharing
- 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)

```
XXXXXXXXXXXXXXXXXXXXXX
X.....................X
X..a.............b..X
X...............X....X
XXXXXXXXXX..XXXXX..XXX
X...............X....X
X...............X.XX.X
X........XXXXXXXX....X
```

```
X.......X...X...X.XXXX
X.......X.e.X.d.X..c.X
X.......X...X...X....X
XXXXXXXXXXXXXXXXXXXXXX
```

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

```
from p1_support import load_level
level = load_level('example.txt')
level.keys() # --> ['walls', 'spaces', 'waypoints']
level['walls'] # --> {(0, 0): 'X', (0, 1): 'X', (0, 2): 'X', (0, 3): 'X', … }
level['spaces'] # --> {(7, 3): '.', (6, 9): '.', (12, 1): '.', … }
level['waypoints'] # --> {'a': (3, 2), 'b': (18, 2), 'c': (19, 9), …}
```