HOMEWORK 6

It's time to add Dijkstra who finds shortest paths in terms of distance! Like the other players, Dijkstra must go from the Start, hit the Target, and then reach the Exit, but use Dijkstra's algorithm to find the optimal path.

INSTRUCTIONS

1. Use the same repository from the previous assignment.

2. In `pathing.py`, `get_dijkstra_path()` is a stub. Implement appropriate code there similar to how you added code for the Random, DFS, and BFS Players. Make sure the scoreboard updates properly. You will need to add a Player in `config_data.py`.

    a. There are many ways you can implement your priority queue. You might find it useful to create a separate class for `Node` objects so you can track their priorities as an attribute, but it is not necessary. You can use the `heapq` object, the `PriorityQueue` object, a dictionary, or basic list. Whatever method you decide, be very careful about how you track updated priorities and parents.

    i. If you use `heapq`, read its Implementation Notes **very** carefully at (https://docs.python.org/3/library/heapq.html) to understand the challenges of updating priorities within the `heapq` object.

    ii. If you use a dictionary with key-value pairs, you will need to sort the dictionary by priority whenever you add an item or update an existing item's priority to ensure you always dequeue the highest priority item. As an alternative to sorting each time, you could simply do a linear search for the highest-priority item.

    iii. If you use `heapq` or `PriorityQueue`, you will probably want to create a `Node` class so you can sort tuples and break ties with a custom comparator. Either way, it can be tricky to update priorities and ensure the next item dequeued is the highest priority because the data structures, by themselves, do not support internal updating of priorities (to update an item's existing priority, you might have to remove and re-insert it).

3. Add appropriate inline assertions to check postconditions for paths returned by `get_dijkstra_path()` checking at least the following postconditions (add more if they are useful for debugging):

    a. **Postcondition**: The result path begins at the start node.
    b. **Postcondition**: The result path ends at the exit node.
    c. **Postcondition**: Every pair of vertices adjacent in the result path is an edge in the graph.

4. Add appropriate unit tests for Dijkstra pathing to the `unit_tests.py`.

5. Push your code changes to your repository.

GRADING

This assignment is worth 100 points:

- **65 points** for a correct Dijkstra implementation
- **5 points** for inline assertions listed as postconditions for paths
- **10 points** for unit tests verifying your Dijkstra implementation
- **20 points** for using good software engineering practices
- **5 points EXTRA CREDIT**: implement `heapq` or `PriorityQueue` with methods to properly update item priorities (instead of sorting or linearly searching for the item with the highest priority)
- **5 points EXTRA CREDIT**: implementing A* using any admissible heuristic
- [If you are claiming extra credit, please make it clear in your README.]