PROBLEM 3 - FINDING THE SHORTEST PATH USING BRUTE FORCE (20/20 points)

We can define a valid path from a given start to end node in a graph as an ordered sequence of nodes [n_1, n_2, ... n_k], where n_1 to n_k are existing nodes in the graph and there is an edge from n_i to n_{i+1} for i = 1 to k - 1. In Figure 4, each edge is unweighted, so you can assume that each edge is length 1, and then the total distance traveled on the path is 4.

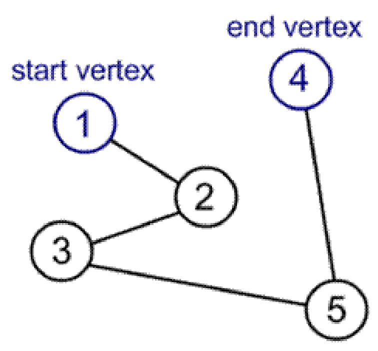

Figure 4. Example of a path from start to end node.

Note that a graph can contain cycles. A cycle occurs in a graph if the path of nodes leads you back to a node that was already visited in the path. When building up possible paths, if you reach a cycle without knowing it, you could get stuck indefinitely by extending the path with the same nodes that have already been added to the path.

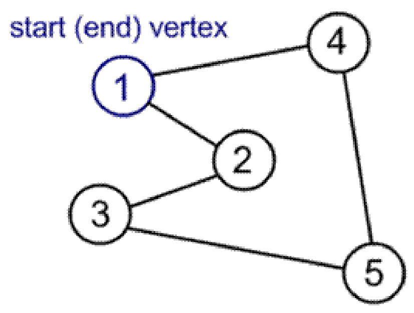

Figure 5. Example of a cycle in a graph.

In our campus map problem, the **total distance traveled** on a path is equal to the sum of all total distances traveled between adjacent nodes on this path. Similarly, the **distance spent outdoors** on the path is equal to the sum of all distances spent outdoors on the edges in the path.

Depending on the number of nodes and edges in a graph, there can be multiple valid paths from one node to another, which may consist of varying distances. We define the **shortest path** between two nodes to be the path with the **least total distance traveled**. In our campus map problem, one way to find the shortest path from one building to another is to do exhaustive enumeration of all possible paths in the map and then select the shortest one.

How do we find a path in the graph? In the depth-first search algorithm, you try one route at a time while keeping track of routes tried so far. Work off the depth-first traversal algorithm covered in lecture to discover each of the nodes and their children nodes to build up possible paths. Note that you'll have to adapt the algorithm to fit this problem. Read more about depth-first search here.

Implement the function `bruteForceSearch(digraph, start, end, maxTotalDist, maxDistOutdoor)` so that for a given digraph, you return the shortest path, from the `start` building to `end` building, such that the total distance traveled is less than or equal to `maxTotalDist` and that the total distance spent outdoors is less than or equal to `maxDistOutdoor`.

For your own benefit, we encourage you to write a sentence describing what the optimization problem is in terms of what the function to minimize is and what the constraints are.

```
# Problem 3: Brute Force Search
#
# State the optimization problem as a function to minimize
# and what the constraints are
#
```

Use the **depth-first** search approach from lecture to enumerate all possible paths from the start to end node on a given digraph. (Assume the start and end nodes are in the graph).

Warning: while you may choose to use DFS code given in lecture as a reference, you will have to adapt it to fit this problem!

Then select the paths that satisfy the constraint and from that group, pick the shortest path. Return this result as a list of nodes, [n_1, n_2, ... n_k], where there exists an edge from n_i to n_{i+1} in the digraph, for all 1 <= i < k. If multiple paths are still found, then return any one of them. If no path can be found to satisfy these constraints, then raise a `ValueError` exception.

Hint: We suggest implementing one or more helper functions when writing `bruteForceSearch`. Consider first finding all valid paths that satisfy the max distance outdoors constraint, and *then* going through those paths and returning the shortest, rather than trying to fulfill both constraints at once.

```
def bruteForceSearch(digraph, start, end, maxTotalDist, maxDistOutdoors):
    """
    Finds the shortest path from start to end using brute-force approach.
    The total distance traveled on the path must not exceed maxTotalDist,
    and the distance spent outdoor on this path must not exceed
    maxDistOutdoors.

    Parameters:
        digraph: instance of class Digraph or its subclass
        start, end: start & end building numbers (strings)
        maxTotalDist : maximum total distance on a path (integer)
        maxDistOutdoors: maximum distance spent outdoors on a path (integer)

    Assumes:
        start and end are numbers for existing buildings in graph

    Returns:
        The shortest-path from start to end, represented by
        a list of building numbers (in strings), [n_1, n_2, ..., n_k],
        where there exists an edge from n_i to n_(i+1) in digraph,
        for all 1 <= i < k.

        If there exists no path that satisfies maxTotalDist and
        maxDistOutdoors constraints, then raises a ValueError.
    """
    # TO DO
```

Help

Paste your code for both `WeightedEdge` and `WeightedDigraph` in the box below. You may assume the grader has provided implementations for `Node`, `Edge`, and `Digraph`. Additionally paste your code for `bruteForceSearch`, and any helper functions, in this box.

```
        totDist, outDist = 0, 0
        for i in range(len(path) - 1):
            u, v = path[i], path[i + 1]
            t, o = getEdgeDist(graph, u, v)
            totDist += t
            outDist += o
        return totDist, outDist

    def DFSShortest(graph, start, end, path = [], shortest = None, shortestTotDist = float('inf')):
        #assumes graph is a Digraph
        #assumes start and end are nodes in graph
        path = path + [start]
        #print 'Current dfs path:', printPath(path)
        if start == end:
            return path
```

Correct

Test results

Hide output

CORRECT

`Test: map1`

Testing map 1

Output:

```
Looking at map 1:
1->2 (10.0, 5.0)
2->3 (8.0, 5.0)
bruteForceSearch(map1, "1", "3", 100, 100)
['1', '2', '3']
Test completed
```

`Test: map2 A`

Testing map 2

Output:

```
Looking at map 2:
1->2 (10.0, 5.0)
1->4 (5.0, 1.0)
2->3 (8.0, 5.0)
4->3 (8.0, 5.0)
bruteForceSearch(map2, "1", "3", 100, 100)
['1', '4', '3']
Test completed
```

`Test: map2 B`

Testing map 2

Output:

```
bruteForceSearch(map2, "1", "3", 18, 18)
['1', '4', '3']
bruteForceSearch(map2, "1", "3", 15, 15)
['1', '4', '3']
bruteForceSearch(map2, "1", "3", 18, 0)
ValueError successfully raised
bruteForceSearch(map2, "1", "3", 10, 10)
ValueError successfully raised
Test completed
```

Test: map3 A

Testing map 3

Output:

```
Looking at map 3:
1->2 (10.0, 5.0)
1->4 (15.0, 1.0)
2->3 (8.0, 5.0)
4->3 (8.0, 5.0)
bruteForceSearch(map3, "1", "3", 100, 100)
['1', '2', '3']
Test completed
```

Test: map3 B

Testing map 3

Output:

```
bruteForceSearch(map3, "1", "3", 18, 18)
['1', '2', '3']
bruteForceSearch(map3, "1", "3", 18, 0)
ValueError successfully raised
bruteForceSearch(map3, "1", "3", 10, 10)
ValueError successfully raised
Test completed
```

Test: map4 A

Testing map 4

Output:

```
Looking at map 4:
1->2 (5.0, 2.0)
3->5 (6.0, 3.0)
2->3 (10.0, 5.0)
2->4 (20.0, 10.0)
4->3 (2.0, 1.0)
4->5 (20.0, 10.0)
bruteForceSearch(map4, "1", "3", 100, 100)
['1', '2', '3']
bruteForceSearch(map4, "1", "5", 100, 100)
['1', '2', '3', '5']
Test completed
```

Test: map4 B

Testing map 4

Output:

```
bruteForceSearch(map4, "1", "5", 21, 10)
['1', '2', '3', '5']
bruteForceSearch(map4, "1", "5", 21, 9)
ValueError successfully raised
bruteForceSearch(map4, "1", "5", 20, 20)
ValueError successfully raised
Test completed
```

Test: map5 A

Testing map 5

Output:

```
Looking at map 5:
1->2 (5.0, 2.0)
3->5 (6.0, 3.0)
2->3 (20.0, 10.0)
2->4 (10.0, 5.0)
4->3 (2.0, 1.0)
4->5 (20.0, 10.0)
bruteForceSearch(map5, "1", "3", 100, 100)
['1', '2', '4', '3']
bruteForceSearch(map5, "1", "5", 100, 100)
['1', '2', '4', '3', '5']
Test completed
```

Test: map5 B

Testing map 5

Output:

```
bruteForceSearch(map5, "1", "3", 17, 8)
['1', '2', '4', '3']
bruteForceSearch(map5, "1", "5", 23, 11)
['1', '2', '4', '3', '5']
bruteForceSearch(map5, "4", "5", 21, 11)
['4', '3', '5']
bruteForceSearch(map5, "5", "1", 100, 100)
ValueError successfully raised
bruteForceSearch(map5, "4", "5", 8, 2)
ValueError successfully raised
Test completed
```

Test: map6 A

Testing map 6

Output:

```
Looking at map 6:
1->2 (5.0, 2.0)
3->5 (5.0, 1.0)
2->3 (20.0, 10.0)
2->4 (10.0, 5.0)
4->3 (5.0, 1.0)
4->5 (20.0, 1.0)
bruteForceSearch(map6, "1", "3", 100, 100)
['1', '2', '4', '3']
bruteForceSearch(map6, "1", "5", 100, 100)
['1', '2', '4', '3', '5']
Test completed
```

`Test: map6 B`

Testing map 6

Output:

```
bruteForceSearch(map6, "1", "5", 35, 9)
['1', '2', '4', '3', '5']
bruteForceSearch(map6, "1", "5", 35, 8)
['1', '2', '4', '5']
bruteForceSearch(map6, "4", "5", 21, 11)
['4', '3', '5']
bruteForceSearch(map6, "4", "5", 21, 1)
['4', '5']
bruteForceSearch(map6, "4", "5", 19, 1)
ValueError successfully raised
bruteForceSearch(map6, "3", "2", 100, 100)
ValueError successfully raised
bruteForceSearch(map6, "4", "5", 8, 2)
ValueError successfully raised
Test completed
```

Hide output

Check Save *You have used 5 of 30 submissions*

Show Discussion New Post