159.271 Computational Thinking for Problem Solving

Tutorial 4: Randomised s-t connectivity

The problem of finding a path from an entrance to an exit in a maze can be modelled by graphs; vertices correspond to special points in the maze, such as the entry and exit points, dead ends, and intersection points. Edges model the connections between those points. The task is to find a path between two special vertices s, the entrance, and t, the exit, of a given graph G. This problem is known as **s-t connectivity**.

In terms of searching a maze, depth-first search of the graph corresponds to moving from one intersection to the next and marking the intersections we have already seen by a pebble. In the worst case, the graph G is a path with n vertices between s and t, and we need n pebbles to complete the search. Using a breadth-first search, we will always find the shortest path from s to t, but we might waste a lot of time investigating dead ends close to s, when t might be just around the corner.

The pebbles in the maze correspond to the memory required by the search algorithm. In the breadth-first search, the queue might contain as many as O(n) vertices, where n is the number of vertices in the graph. In the depth-first search, we might have to call the algorithm recursively O(n) times, requiring that O(n) vertices are placed on the stack. How could we improve on these algorithms?

A randomised algorithm for s-t connectivity is simple. Begin at vertex s; at each step continue to one of the neighbours of the current vertex, selecting each neighbour with equal probability at random; stop when t is reached. This approach is called taking a random walk on the graph. The memory requirements are much reduced, as there is no need to memorise any history of visited vertices. Algorithmically, we can write the randomised search as follows:

```
def randomised_st_connectivity(G, s, t):
    # Input parameters: G the graph,
                        s the start vertex,
                        t the end vertex
    # Output: True, if the search terminates

    current_vertex = s
    while current_vertex != t:
       current_vertex = random vertex chosen from neighbours of
current_vertex
    return True
```

Given a graph on n vertices with two special vertices s and t, this version of **randomised_st_connectivity** finds a path from s to t with probability at least 1/2 in less than $2n^3$ steps, if such a path exists (you can just believe this for now, or Google for a proof.) If there is no such path, the algorithm runs forever; we have written an algorithm that does not necessarily terminate. This is a Las Vegas style randomised algorithm, there is a small probability that it may run for a very long time even when there is a path from s to t.

We could stop the algorithm after $2n^3$ steps. If it has not found t by that time, we return False. This version of the algorithm has the following properties: if it returns True it has found a path from s to t; and if there is such a path, it finds it with probability at least 1/2. However, it is possible that the algorithm returns False, even though there is a path from s to t. This is a Monte Carlo style randomised algorithm. One advantage of a Monte Carlo algorithm is that we can make the error probability arbitrarily small by running the algorithm repeatedly.

```
def randomised_st_connectivity(G, n, s, t):
    # Input parameters: G the graph,
                        n the number of vertices in G,
                        s the start vertex,
                        t the end vertex
    # Output parameters: True or False

    stepcount = 0
    current_vertex = s
    while (current_vertex != t) and (stepcount < 2n^3):
       current_vertex = random vertex chosen from neighbours of
current_vertex
       stepcount = stepcount + 1
    if current_vertex = t:
       return (True, stepcount)
    else:
       return (False, stepcount)
```

Implement the Monte Carlo version of **randomised_st_connectivity** outlined above. Note that each neighbouring vertex is to be chosen with equal probability at random, if there are three neighbours for the current vertex each of them should have a 1/3 chance of being chosen. You will probably find it easiest to store your input graph using an adjacency matrix.

Run your code on the three test graphs supplied on Stream (and any others that you choose) and report back on how many steps/repetitions were required to discover each of the s-t paths. Your implementation will need to handle the test graph files as input. The test graph files are text files with the following format:

```
n # number of nodes in the graph G
s # index of the start node
t # index of the end node
# blank line, followed by n rows of digits representing the adjacency
matrix for G
0 1 0 0 1 1 0 0 0 0
1 0 1 0 0 0 1 0 0 0
0 1 0 1 0 0 0 1 0 0
0 0 1 0 1 0 0 0 1 0
1 0 0 1 0 0 0 0 0 1
1 0 0 0 0 0 0 1 1 0
0 1 0 0 0 0 0 0 1 1
0 0 1 0 0 1 0 0 0 1
0 0 0 1 0 1 1 0 0 0
0 0 0 0 1 0 1 1 0 0
```

Submit your code as a single **.py** file. Submit the results report that you produce in a separate text file.