Analysis of Algorithms

Alvin Lin (axl1439) and William Leuschner (wel2138)

August 2017 - December 2017

Problem 1

Algorithm Description

For this problem, we implemented a dynamic programming-esque solution to find the number of shortest paths from a starting node to every other node in the graph. By using breadth first search, we were able to populate the solution array in linear time by summing counting the frequencies of each path length and taking the frequency of the minimum path length. The dynamic programming heart of the solution is as follows:

$$S[i] = \text{the number of shortest length paths from the start node to node } i$$

$S[i]$ is calculated by looking at the shortest length paths of all its neighbor nodes. Each neighbor node may have one or more shortest length paths leading to it, and we sum up the number of those shortest length paths for all the neighbors of node i for the shortest length path leading to node i. The breadth-first search advances one neighbor during each loop iteration, thus when it touches a node, the path it has calculated to that node is guaranteed to be a shortest path. We simply count the multiplicities of these paths and propagate them to all the downstream nodes. Thus, the number of shortest length paths from the start node to the end node t is simply $S[t]$.

Running Time Estimate

The conversion of the edge list to an adjacency matrix requires $m + n$ time. The breadth first search touches every node and some of the edges, requiring $m + n$

time. The calculation of the number of shortest length paths is constant time, so our algorithm is $O(m+n)$.

Pseudocode

```
def findNumPaths(n, start, target, adjacencyList):
  let queue = new queue()
  let visited = boolean array of size n
  let pathMultiplicities = int array of size n

  queue.push(s)
  visited[s] = true
  pathMultiplicities[s] = 1

  while queue is not empty:
    let queueSize = queue.size()
    let visitedDuringCurrentStep = boolean array of size n
    for i = 0 to queueSize:
      let node = queue.pop()
      for each neighbor of node:
        if not visited[neighbor]:
          queue.push(neighbor)
          pathMultiplicities[neighbor] = pathMultiplicities[node]
          visited[neighbor] = true
          visitedDuringCurrentStep[neighbor] = true
        else if visitedDuringCurrentStep[neighbor]:
          pathMultiplicities[neighbor] += pathMultiplicities[node]

  return pathMultiplicities[t]
```

Problem 2

Algorithm Description

For the problem, we implemented a dynamic programming solution to find the longest prerequisite chain by topologically sorting the graph and propagating each course to every upstream requirement. The dynamic programming part is as follows:

$$S[i] = \text{the longest chain of prerequisite courses required for course } i$$
$$S[i] = max(S[i], S[requirement] + 1) \text{ for every prerequisite of course } i$$

The solution is given as the largest element in S.

Running Time Estimate

The topological sort runs in $m + n$ time since it is composed of a depth first search operations that touch every node and edge. The dynamic programming part runs in $m + n$ time as well since it must iterate through each node in the prerequisite graph and each neighbor of the nodes. This makes our algorithm $O(m + n)$ overall.

Pseudocode

```
def getUnvisited(visited):
  for i = 0 to visited.length:
    if not visited[i]: return i
  return -1

def sortTopological(n, adjacencyList):
  let sorted = new stack()
  let visited = boolean array of size n
  let unvisited = getUnvisited(visited)
  while unvisited != -1:
    visit(unvisited, adjacencyList, sorted, visited)
    unvisited = getUnvisited(visited)

  let result = int array of size n
  while not sorted.isEmpty():
    result.append(sorted.pop())
  return result

def visit(node, adjacencyList, sorted, visited):
  if visited[node]:
    return
  for neighbor in adjacencyList[node]:
    visit(neighbor, adjacencyList, sorted, visited)
  visited[node] = true;
  sorted.push(node)

def prerequisites(n, adjacencyList):
  let topologicalSort = sortTopological(n, adjacencyList)
  let prerequisites = int array of size n
  for node in topologicalSort:
    for upstream in adjacencyList[node]:
      prerequisites[upstream] = max(
        prerequisites[upstream],
        prerequisites[node] + 1
      )
  return max(prerequisites)
```

Problem 3

Algorithm Description

For this problem, we implemented a solution that is probably more complex than it needed to be. First, we generate a tree of all possible moves for the Things to make. This tree, rather than re-generating moves it has already generated, simply contains links to the portion of the tree where that generation first occurred. We did this by recording every configuration's node in a 4D matrix (where the indices are the row and column of each Thing), and checking to see if a node already exists in the matrix at that configuration's set of indices. If it does, we replace the node we just generated with the one from this matrix.

Then, we search through the linked tree with BFS in order to find the closest node where the Things can leave the grid on the same edge in one more move (we call this the winning configuration). Rather than the bizarre linking thing, the BFS simply uses a visited array in order to determine if we have already seen a node, and skip it if necessary.

This solution seems to be correct by inspection, although our implementation of it is flawed in a way that we were unable to determine, and consequently does not work.

Running Time Estimate

If it **did** work, it would run in $O((ab)^2)$ time, or equivalently $O(m+n)$. The operation of searching the matrix to identify where Thing 1 and Thing 2 are takes $O(ab)$ time, twice, to look through at worst every location in the matrix. The operation of building the graph requires $O((ab)^2)$ time, since it will touch every index of the matrix (ab) times in order to build the tree. Finally, searching through this tree will take $O(m + n)$ time, since we are using BFS. All together, this makes an $O((ab)^2)$ time algorithm.

Problem 4

Algorithm Description

For this problem, we implemented a variation of Kruskal's minimum spanning tree algorithm. Before running the greedy portion of Kruskal's algorithm, we sorted the edges first by whether or not they were in the subset F, and then by weight. During the edge inclusion part of Kruskal's algorithm, if an edge from set F is excluded, then

we can stop immediately since no such spanning tree that includes all the edges in set F is possible.

Argument of Correctness

This algorithm works because we prioritize the choosing of all the edges from set F first. If they are part of the minimum spanning tree containing all the edges of set F, then they will be chosen. If we exclude an edge that was included in the set F, then that edge would have caused a cycle in the minimum spanning tree, and thus no such spanning tree containing all the edges in set F would have been possible.

Running Time Estimate

Kruskal's algorithm runs in $O(m \log n)$ and the only modification to Kruskal's algorithm here is the sort order, so the runtime remains the same as the regular Kruskal's algorithm.

Pseudocode

```
def sortComparator(edge1, edge2):
  if edge1.isF == edge2.isF:
    return edge1.weight - edge2.weight
  if edge1.isF:
    return -1
  return 1

def kruskal(n, edges):
  let heap = heapSort(edges using sortComparator)
  let tree = []
  let set = new disjoint set
  while heap is not empty:
    let edge = heap.pop()
    if set.getBoss(edge.vertex1) != set.getBoss(edge.vertex2):
      tree.add(edge)
      set.union(edge.vertex1, edge.vertex2)
    else if edge.isF:
      return -1
  let weight = 0
  let boss = -1
  for i = 0 to tree.size():
    let edge = tree[i]
    weight += edge.weight
    if boss == -1:
```

```
      boss = set.getBoss(edge.vertex1)
    else if boss != set.getBoss(edge.vertex1):
      return -1
  return weight
```

Problem 5

Algorithm Description

For this problem, we implemented the Strongly Connected Components algorithm. The key insight we had into this problem was that if there is not exactly 1 strongly connected component with 0 incoming edges and exactly 1 strongly connected component with 0 outgoing edges, it is not possible to draw 1 additional edge and make the entire graph strongly connected. In order to determine that information, we first run Strongly Connected Components on the forward topological ordering of the nodes, then run it again on the reversed topological ordering of the nodes. If there is more than one strongly connected component produced by the algorithm for the reversed topological ordering, the problem is not solvable.

Running Time Estimate

This algorithm can be broken down into a few large pieces, all of which have known time complexities. First, it has to obtain the transpose of the graph. This is $O(m+n)$ because it needs to touch every node and every edge. Next, it needs to find a topological ordering of the graph. This is also $O(m+n)$ because it needs to touch every node in the graph. Finally, it needs to DFS through the graph, twice (once for the forward topological ordering, once for the backward). Each of these is $O(m+n)$, and since they are run in sequence, they do not multiply. In total, the algorithm runs in $O(4(m+n))$, but we omit the 4 because that's how Big-Oh notation works.

Pseudocode

```
let n be the number of nodes in the graph
let transpose be the transpose of the graph
let sorted be the topologically sorted array of nodes in the graph
let seen be an array of size n
let components be a linked list
let backwardcomponents be a linked list

for each node in sorted:
  if not seen(node):
```

```
    let component = dfs(node, transpose, seen)
    components.add(component)

for each element in seen:
  element = false

for each node in sorted, in reverse order:
  if not seen(node):
    let component = dfs(node, transpose, seen)
    backwardcomponents.add(component)

if backwardcomponents.length > 1:
  output "NO"
else
  output "YES"
```