Homework 3 Dynamic Programming and Graph Search

Matt Demusz, Frank Porco, Joyce Wang

Part A (35 points)

Problem 1:

Assumptions:

- Each player only has one possible path to follow.
- Calculating the distance between two stones happens in constant time.
- We have a stone.hasNext() that returns true if there is a next stone to jump to
- We have a stone.next that represents the player jumping to the next stone

Algorithm 1: Minimum rope length algorithm (Node src1, Node src2)

```
1  int minLen = distance(src1, src2);
2  int move1, move2, both;
3  Node temp1 = src1;
4  Node temp2 = src2;
5  while (temp1.hasNext() OR temp2.hasNext()):
6          if (temp1.hasNext() AND !temp2.hasNext()):
7                  move1 = distance(temp1.next, temp2);
8                  if (move1 > minLen):
9                          minLen = move1;
10                 temp1 = temp1.next;
11         else if (!temp1.hasNext() AND temp2.hasNext()):
12                 move2 = distance(temp1, temp2.next);
13                 if (move2 > minLen):
14                         minLen = move2;
15                 temp2 = temp2.next;
16         else:
17                 move1 = distance(temp1.next, temp2);
18                 move2 = distance(temp1, temp2.next);
19                 both = distance(temp1.next, temp2.next);
20                 if (min(move1, move2, both) == both):
21                         if (both > minLen):
22                                 minLen = both;
23                         temp1 = temp1.next;
24                         temp2 = temp2.next;
25                 else if (min(move1, move2, both) == move1):
26                         if (move1 > minLen):
27                                 minLen = move1;
28                         temp1 = temp1.next;
29                 else:
30                         if (move2 > minLen):
31                                 minLen = move2;
32                         temp2 = temp2.next;
33 return minLen;
```

Run Time: If we assume the distance function to be constant time, then the overall run time of this algorithm would be O(n + m) with n being the number of red stones, and m being the number of blue stones. All of the variable assignments and the checks are constant time, and they run as many times as the while loop does. The maximum number of times this loop can run is the case where the shortest rope length is when one player jumps, and the other stands still every time. This in total would be n + m times, leaving the overall running time as O(n + m).

Part B (35 points)

Problem 2:

A. You have a collection of n distinct chopsticks of length $l_1, \ldots, l_n$. Any two of them can be paired for use if the length of them differ at most k. How can you easily pair as many of the chopsticks as possible? Describe a greedy algorithm of time complexity $O(n \log n)$ to solve this problem and prove the correctness of your algorithm.

First, we can sort the chopsticks from shortest to longest using mergesort. The runtime of mergesort is *nlogn*. After we sort them, we can start from the shortest chopsticks and grab them in pairs. In the pair that was grabbed, we can take the longer chopstick length and subtract that by the shorter chopstick length. If the result of the subtraction is less than or equal to k, then we can put those chopsticks aside as a pair. If the result of the subtraction is greater than k, then we can discard the smaller chopstick in the pair and grab the next chopstick in the list of sorted chopsticks to pair with the "greater" chopstick if their difference in length is less than or equal to k. Traverse through the list of chopsticks by always looking at the smallest two chopsticks, and repeat the subtraction process above until no chopsticks are left to pair. Traversing through the list of chopsticks requires one iteration only, giving us a running time of n, so we have $O(nlogn + n)$, which is equal to $O(nlogn)$.

B. Consider now a variant of the above problem. You can still only pair chopsticks that differ at most k in length. But now a value w_i is also associated with each individual chopstick. You want to maximize the sum of the values of the chopsticks that have been paired.

For example, suppose you have 7 chopsticks of length $5, 2, 3, 11, 9, 12, 16$ and corresponding values $1, 1, 2, 5, 3, 3, 10$. You are allowed to pair chopsticks that differ by at most 3 units in length. Then one of the optimal solutions here is $\{(2, 3), (9, 11)\}$ of optimal value $1+2+3+5 = 11$.

How can you pair the chopsticks so as to maximize the value? Describe a dynamic programming algorithm of time complexity $O(n^2)$ to solve this problem. Can you do better than $O(n^2)$?

Given chopsticks with their lengths and weights, we can sort the chopsticks by length in an array *sortedChopsticksArr* and store its corresponding weight value in it ($O(n)$, where n is the number of chopsticks).Then, we can make a 2D array, called *chopstickSumWeights*. We can use nested loops to iterate through the list of sorted chopsticks: in *sortedChopsticksArr*, for each chopstick i, we want to iterate through every chopstick j, take the absolute value difference of $sortedChopsticksArr[i] - sortedChopsticksArr[j]$ and check if this value does not exceed k ($O(n^2)$). If the value does not exceed k, then we can put in the sum of $sortedChopsticksArr[i] + sortedChopsticksArr[j]$ in $chopstickSumWeights[i][j]$. Otherwise, we just set $chopstickSumWeights[i][j]$ equal to null.

To begin, we can make a new hash table *pairs*, and hash all of the chopsticks as our keys by iterating through our sorted chopsticks list (we can set the values to null for now) ($O(n)$). We can also make a variable called $totalSum = 0$ to keep track of the total optimal sum. Now that we have all the values in *chopstickSumWeights*, we have to iterate through it to find the optimal value for the sum of the weights ($O(n^2)$). We can loop through the ith row and iterate through the all j columns to find the maximum value in that row. To find the maximum variable, we can have a

ChopStick object *currChopstick* that keeps track of the largest value that has been visited in that row and the other chopstick in that pair.

After we get the maximum value in the *i*th row and the pair of chopsticks for that maximum value, we go into our hash table to check if one of the chopsticks has been used already (if it's value in the hash table is not *null*). If the chopstick is not *null* and is a ChopStick object, then we compare the sum stored in the hash table for that chopstick to our *currChopstick* value. If *currChopstick* value is greater than the sum in the hash table, meaning that we found a better value to replace the old pair, then we go into the hash table, set each chopstick in the old pair to null, subtract the old sum from *totalSum*, and set each chopstick in the new pair's ChopStick object x to the pair's sum and the other chopstick in the pair ($O(1)$). Then, we do $totalSum = totalSum + x$ ($O(1)$). We continue going down the *ith* rows. If we find that all the values j in the row are *null*, then there is no maximum value, so we just move down to the next row. Continue until we are done iterating through the rows, and *totalSum* should store the optimal sum at the end. To get the actual chopstick pairs, we can create another array *optimalPairsArr* that stores all the pairs. We can iterate through *sortedChopsticksArr* ($O(n)$), and at each chopstick, we go to to the hash table to find if it has a pair ($O(1)$). If it does, then make sure that it already isn't in *optimalPairsArr* and add that pair into the array.

As noted above, we have $n + n^2 + n + n^2 + n$. Therefore, we have $O(n^2)$ as the running time, and we cannot do better than $O(n^2)$ since we must use a 2D array to keep track of the weights and we must iterate through this array, which will be $O(n^2)$.

Part C (20 points)

Problem 3: In the robotics lab the new robot has just arrived. The robot has the ability to construct a topological map of the environment, such as the graph shown in Figure **??**. The robot is allowed to move only forward along the directions of the edges on the topological map. Moreover, the graph is being constructed in such a way that will prevent the robot to execute loops, i.e., the robot is not able to visit a node that it has already visited.

A. Given a start location for your robot and a target location, provide an efficient algorithm that will return all the possible paths from the start to the target. What is the running time for your algorithm?

ANSWER: You can use a modified depth-first search:
Nodes aren't marked as visited because we're looking for all possible paths instead of just searching if a node is in the graph. The graph is a DAG as specified in the problem, so there aren't any cycles that would cause an infinite loop with this modification. You start at the starting vertex/node C and call DFS on each adjacent node N with an edge from C to N. You go as deep as you can each time, stopping when you reach the target or a vertex with no edges to adjacent nodes. You can use a stack to keep track of nodes that you're visiting, pushing each new node you move to onto the stack. When you get to the target, you can look at the stack (e.g. pop all into another stack to see the path, and push them back onto the original stack to maintain the path) to see each node/vertex that was visited. Every time a path is found, it can be added to a structure like an ArrayList. Each time you backtrack, you pop a node off the stack. You will end up with a list of all possible paths from the starting location to the target location. This method has an exponential runtime O(2^|V|) because you have to find every possible path (and not just count the number of different paths).

B. You want to check if the topological map provides enough information for your robot to be able to visit all the rooms so as to clean them. Provide an efficient algorithm that will be able to check if there is a path for the robot on the graph that can visit all the rooms (i.e., nodes on the graph).

ANSWER: You can use a topological sorting algorithm to sort the nodes in the DAG into an ordered list:
Create an empty list that will store the sorted vertices. Define a function “visit” and call it on the starting vertex:

1. If the current vertex/node C has not been visited yet, call visit on each node N with an edge from C to N.
2. Mark C as visited
3. Add C to the front of the list

This topological sort has a runtime of O(|E|+|V|) because you have do one check for every edge in the graph (has it been visited) and call visit once on each vertex/node in the graph.

Then, you can check if each vertex has an edge to the next vertex (i.e. each adjacent pair of vertices has an edge between them). If each pair has an edge, there is a path through each node/room that only visits each node/room once. You have to perform |V|-1 checks, so this has a runtime of O(|V|).

This method has an overall runtime of 2|V|+|E| = O(|V|+|E|).

Part D (20 points)

Problem 4: You are preparing a banquet where the guests are government officials from many different countries. In order to avoid unnecessary troubles, you are asked to check the list of international conflicts in the last ten years. Then, you will assign the guests to two tables, such that in each table, any two guests are not from countries that had conflicts in the last ten years.

Provide an efficient algorithm that determines whether it is possible to make such an assignment. If it is possible to do so, the algorithm should return the assignment of these two tables. What is the running time?

Given a list of international conflicts of size n and a list of guests from each country of size m:

1. Iterate through the list of guests and create a hash map where the key is the contry and the value is the guest’s name. ($O(m)$)
2. Create a queue that will be used to store the first node of each disconnected graph.
3. Traverse the list of international conflicts. ($O(n)$) For each country1 and country2 that has a conflict:
    - Use the hash map to check if country1 has a person ($O(1)$). If country1 has a person and is not already on the graph, then add that country’s person as a node to the graph ($O(1)$). If country1 has person and is already on the graph or if country1 does not have person, then don’t add anything to the graph.
    - Use the hash map to check if country2 has a person ($O(1)$). If country2 has a person and is not already on the graph, then add that country’s person as a node to the graph ($O(1)$). If country2 has person and is already on the graph or if country2 does not have person, then don’t add anything to the graph.
    - If both nodes are on the graph, then a strongly connected relationship should be denoted between these two nodes.
    - If a disconnected graph has been created, then add one of the nodes from that graph to the queue so that we can keep track of all of the disconnected graphs ($O(1)$).
4. Create two arrays: one for table1 and another for table2.

5. Dequeue from the queue ($O(1)$), and starting from that node, and using BFS, traverse through one of the graphs. At the first node of that graph, add it to table1. For all the neighbors of that node, add them to table2 if they do not already exist in table2 since being neighbors denote a conflict, which means that two nodes that are neighbors cannot be in the same table, or array.

6. Continue to traverse using BFS until all nodes have been visited, and everytime we're on a new node, make sure that it does not already exist in a table and that it is not added to the same table as its neighbors that have already been visited. If we find that we cannot add a node to either table because it already has visited neighbors in both tables, then we stop and return false. Otherwise, we continue to dequeue and repeat steps 5 and 6 until the queue is empty, and we return the two arrays of tables.

Without taking into consideration the running time of BFS, we have $O(m+n)$ so far. For BFS, let E be the count of all edges in the graph, and there are m nodes, or vertices. The total running time is of BFS is $O(|E|+|m|)$. Therefore, we have $O(m+n+E+m) = O(m+n+E)$.