Part A (35 points)

Problem 1

You are participating in the design of a new stepping stones challenge for the remake of the cult Japanese TV show "Takeshi's Castle".

The challenge involves a team of two people tied with a rope that need to walk over a sequence of stepping stones. The first teammate is allowed to go over the stepping stones that are painted red $\{r_0, \ldots, r_n\}$, while the second teammate is allowed to go over the stepping stones that are painted blue $\{b_0, \ldots, b_m\}$. For the team to win, the two players have to walk over all the stepping stones in the corresponding sequence while they are connected with the rope. The teammates are not allowed to backtrack. At each point in time, either one of the players can jump from her current stone to the next one and the other one stays at his current stone, or both of them jump simultaneously from their current stones to the next ones. Such a jump is obviously feasible only if the distances between the two players before and after the jump are less than the length of the rope connecting them.

The TV show producer has already built the two sequences of red and blue stepping stones. Given the coordinates of the stepping stones, your task is to select the minimum length of the rope for which it is possible for the two players to win the game. This is what will make the show entertaining for the audience!

Provide an algorithm for this computation and argue its running time.

This problem is the same as the edit distance but instead of two words we have from r_0, r_n and b_0, b_n. Similar to that problem from class, we must solve a prefix of r_0, r_j and b_0, b_j. There are three sub problems: $r[j]$ and a gap, $b[j]$ and a gap, or $r[j], b[j]$ meaning they are the same or different and nothing needs to be added. We can use what is in the DPV book, $E(i,j) = min\{1 + E(i-1,j), 1 + E(i,j-1), diff(i,j) + E(i-1,j-1)\}$. where $diff$ is the difference, 0 if they are the same and 1 if they are different. Rather than finding the edit distance for words, we find the edit difference between the movements of each player. Our max number, n, is the length of the rope. If we need to add a gap, we increase by one. A gap means that either play moves while they other stays still. If they jump together, then we do not. The largest this distance can be is the length of the rope. Because of this, the total running time to compute this is $O(nm)$.

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.**

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 $O(n^2)$?

Describe a greedy algorithm of time complexity $O(n \log n)$ to solve this problem and prove the correctness of your algorithm.

A greedy algorithm for this would be to take every chopstick n_i and a second one, n_j that differ by k as many times as possible. When we run out of two pairs i, j that are different by k, we take two pairs that differ by $k - i$ until the set is empty. This is our greedy algorithm. This takes $O(n \log n)$. Why? Because each time we're taking 2 chopsticks (a tree basically) and we continue this n times each timing moving down a level (meaning changing the difference between the two chopsticks).

Describe a dynamic programming algorithm of time complexity $O(n^2)$ to solve this problem. Can you do better than $O(n^2)$?

A dynamic programming algorithm for this is similar to the knapsack problem. $K(w, j) = max\{K(w - wk, j-1) + v_j, K(w, j-1)\}$ where $0 \leq j \leq n$. K is the "knapsack" in this problem, or holding the chopsticks. Rather than taking each chopstick, we take two at a time. This is $O(nW)$.

It is possible to do better than $O(n^2)$ just by using a sort function on the chopsticks and pairing the best match (from our greedy algorithm).

Part C (20 points)

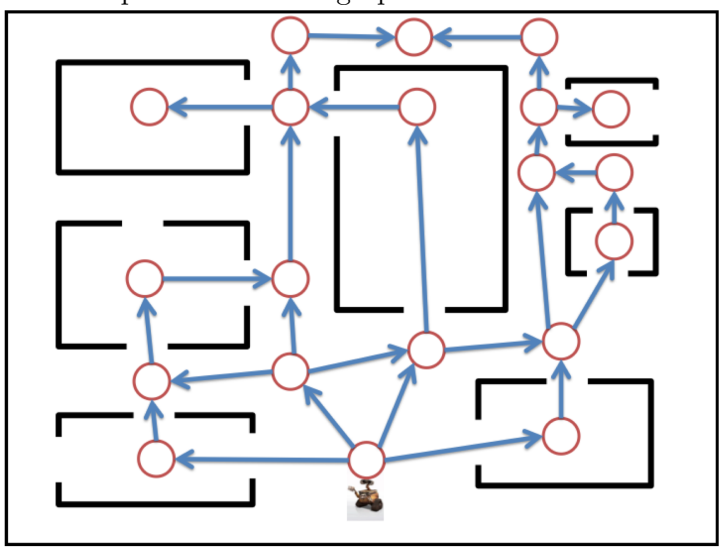
Problem 3

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 1. 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?**

```
set_of_all_paths(G, s, e, p=[]):
    if s == e {
        return s;
    }
    p = [];
    for (n in G[s]) {
        if n not in p {
            nps = set_of_all_paths(G, n, e, p)
            for np in nps {
                paths.append(np);
            }
        }
```

Figure 1: An example of a directed graph that the robot build for this map.



```
}
return p;
```

This algorithm exploits DFS, where the expensive operation is the recursive call. This will happen in the worst case the number of edges.

$$O(|E|)$$

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).**

There are two ways this problem could be interpreted: (1) you want an algorithm that will travel to every node in a single path at least once, or (2) you want an algorithm to see if every node is reachable in principle in any given trip. The ambiguity is in the phrase "is a path for the robot on the graph that can visit all the rooms," specifically whether the robot has one or more trips. If the robot only has one trip, that is the first interpretation. If this robot can have as many trips as it wants, this is a connectedness problem. The first interpretation is the traveling salesman problem, especially if a shortest path between all rooms is wanted. The second interpretation is about the connectedness problem. As it is easier, I will first solve the second problem, and then solve the second problem.

In order to check connectedness, you need to visit every node and check the resulting order of the graph against the number of rooms on your topographical map. You can use either BFS or DFS to complete this task. Roughly, you are going to begin on any given node, utilize your traversal algorithms, and then compare the number of traversed nodes to the number of total nodes. Presumably, if I have the topological map, I also have the number of nodes on that map. If the number of nodes my DFS returns is equal to number of nodes in my model, then the robot can get to every node in the graph with sufficient trips. If the returned number is less than the number on my model, then there is at least one island. If the number is more than the number on my model, my model is missing at least a few nodes.

The runtime of checking connectedness in this way will be as follows. There is one step for checking the size of my model. There is then, at worst, the number of steps equal to the number of edges of the graph. Then there is a step to compare the number returned to the number of my model.

$$O(|E|)$$

In order to get the shortest path between two nodes, we are going to need to use a more sophisticated algorithm, specifically the Floyd-Warshall algorithm. My model gives me two things: my start node, and because it's a DAG I have access to, my end node. There can be multiple endings to a DAG. But in order to have a path that can visit every node in a shortest path for a DAG, there need be one end point. This can be ascertained from the topological map that the robot has built. This means that the graph must be linearized for there to be a shortest path for all nodes.

The Floyd-Warshall algorithm operates by finding the set of permissible intermediate nodes. This happens one node at a time, and updates the shortest path length at each stage. After this operation has continued for some time, it will grow to set of all V, which is when all vertices are allowed to be on all paths.

Specifically, the algorithm is as follows:

```
for i=1 to n:
  for j = 1 to n:
    dist(i, j, 0) = infinity
for all (i, j) in E:
  dist(i, j, 0) = l(i, j)
for k = 1 to n:
  for i=1 to n:
    for j = 1 to n:
      dist(i,j,k)=min{dist(i,k,k-1)+dist(k,j,k-1), dist(i,j,k-1)}
```

Floyd-Warshall is better than the default algorithm that this, which would visit every node and compute the distance to the target.

$$O(|V|^3)$$

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?

This is an instantiation of the bipartite graph problem. Make every country on the list a node on a graph. Define an edge between two nodes as if there has been a conflict between the two countries. Take any given node on this graph an color it "blue" (for democracy). Visit all the neighbors of that now "blue" node and color those "red" (for commies). For every subsequent visit, switch color. If in this process you encounter a node that is differently colored than the color you intend to use, there is no valid seating arrangement. When you run out of nodes to

color, you have your solution, where the blue colors will be one table and the red colors another table. This will run in linear-time, worst-case, because the worse-case is the successful one, where each node is visited once. It will be equal to the number of edges, as they will each be touched once, much the same way, in fact exactly, as DFS.

$$O(|E|)$$