challenge5

November 16, 2023

1 CS 311 Challenge Problems #5

1.1 Problem 1

Pebbles Game. Two players play a game, starting with three piles of n pebbles each. Players alternate turns. On each turn, a player may do one of the following: - take one pebble from any pile - take two pebbles each from two different piles

The player who takes the last pebble(s) wins the game.

Write an algorithm that, given n, determines whether the first player can "force" a win. That is, can the first player choose moves such that, regardless of what the other player does, they will always win. In addition, write a procedure that tells a player which move (if any) they should make to force a win given the current triple of pebbles in each pile. What is the complexity of your solution?

1.1.1 Algorithm

[ ]:

1.1.2 Correctness

1.1.3 Runtime

1.2 Problem 2

1.2.1 Algorithm

[ ]:

1.2.2 Correctness

1.2.3 Runtime

1.3 Problem 3

1.3.1 Algorithm

[ ]:
```
def assignVoters(voters, pollingLocations, r, C):
    # voters: list of voters
    # pollingLocations: list of polling locations
    # r: maximum distance a voter can travel
```

```
    # C: maximum number of voters per polling location
    # return: true or false, indicating whether everyone can be assigned to a␣
 ↪polling location

    s = # initialize a source
    t = # initialize a sink
    G = # initialize a graph G with voters and polling locations as nodes,␣
 ↪source s and sink t

    for voter in voters:
        # add an edge from the source to voter in G with capacity 1
        for poll in pollingLocations:
            # add an edge from poll to the sink in G with capacity C
            if distance(voter, poll) <= r and numVoters(poll) < C:
                constructEdge(voter, poll, G)

    flowValue = fordFulkerson(G, s, t)
    return flowValue == len(voters)


# Helper functions
def distance(voter, poll):
    # voter: a voter
    # poll: a polling location
    # return: the distance between the voter and the polling location

def numVoters(poll):
    # poll: a polling location
    # return: the number of voters currently assigned to the polling location

def constructEdge(voter, poll, G):
    # voter: a voter
    # poll: a polling location
    # G: a graph
    # return: None

    # add an edge from voter to poll in G and increment capacity by 1

def fordFulkerson(G, s ,t):
    # G: a graph
    # s: source
    # t: sink
    # return: the maximum flow in G
```

1.3.2 Correctness

We will prove this algorithm by showing how the problem can be modeled as a maximum flow problem, and it is known that the Ford-Fulkerson algorithm correctly solves maximum flow problems.

We model the problem by creating a bipartite graph, where we have two sets of nodes, voters and polling locations. The source has an edge of capacity 1 to all voters. We create an edge between a voter and a polling location if the voter is within r units of the polling location and the capacity at that location has not yet been reached, the capacity of this edge is also 1. Furthermore, each polling location node has an edge to the sink of cacacity C. In my algorithm, I loop over every voter at each polling location, ensuring that if an every voter has a chance to be assigned to each poll given that it meets the restraints of C and r.

The resulting graph has the source connected to every voter node with capacity 1, voter nodes connected to polling location nodes with capacity 1, and polling location nodes connected to the sink with capacity C.

The Ford-Fulkerson algorithm is then called on this graph, correctly returning the maximum flow. The maximum flow cam be interpreted as the number of voters that can be successfully assigned to polling locations. The reasoning is that let's assume there are k voters, if every voter can be assigned to a poll, then the graph will contain k capacity 1 edges between the voter nodes and the poll nodes, since during the construction of the graph no more than C voters are assigned to each poll, the bottleneck of this graph won't be the edges from the polls to the sink, but rather the k edges from the voters to the polls. If the maximum flow returned by the Ford-Fulkerson algorithm is equal to the number of voters (k), it means that every voter can be assigned to a polling location within r miles of their home, adhering to the constraint that no poll may exceed a capcity of C voters, and we return true. If not, we return false meaning the given constraints make it impossible for every voter to be assigned to a poll.

Therefore, the algorithm correctly solves the problem.

1.3.3 Runtime

The runtime for this algorithm can be thought of in two parts, the loop that iterates over each voter and polling location while constructing a graph, and finding the maximum flow of the resulting graph.

In the worst case, each polling locations is connected to each voter so this first step has a time complexity of $O(mn)$.

Finding the maximum flow can be achieved using the Ford-Fulkerson algorithm. Lets assume the method of finding augmenting paths is by using DFS, then the algorithm runs in $O(\text{num edges} * \text{max flow})$. In the worst case each polling location is connected to each voter so the number of edges is mn, the maximum flow is limited by C, the maximum number of voters allowed at a single location. With the maximum number of edges mn and the maximum flow C, this step has a time complexity of $O(mnC)$.

These steps are in sequence so the total runtime will be: $O(mn + mnC) \rightarrow O(mnC)$

1.4 Problem 4

Network Flows (K&T Ch 7 Ex 12). You are given a flow network with unit-capacity edges: It consists of a directed graph G = (V, E), a source s V , and a sink t V ; and ce = 1 for every e E. You are also given a parameter k. The goal is to delete k edges so as to reduce the maximum s − t flow in G by as much as possible. In other words, you should find a set of edges F E so that |F| = k and the maximum s − t flow in G = (V, E F) is as small as possible subject to this. Give an efficient algorithm to solve this problem, argue it is correct, and state its running time.

1.4.1 Algorithm

[ ]:
```
def minimizeMaxFlow(G, s, t, k):
    # G: a graph representing a flow network
    # s: source node
    # t: sink node
    # k: number of edges to delete
    # return: G after deleting k edges such that its maximum flow is as small␣
 ↪as possible

    # 1. Run F-F algo
    # 2. Make list from min-cut using residual graph. (max-flow min-cut theorem)
    # 3. remove k edges from this list.
    # 4. return G
```

1.4.2 Correctness

From the max-flow min-cut theorem we know that the maximum s-t flow is equal to the capacity of the minimum cut. Therefore to reduce the maximum flow, we must remove edges that are part of the minimum cut. If we remove all the edges in the minimum cut, and still have not yet removed k edges, we find the new minimum cut after removing these edges until we have removed a total of k edges, repeating this process as necessary.

1.4.3 Runtime

[ ]:
```

```