challenge5

November 16, 2023

Christian Noble Shriver

0.1 Problem 1

empty

0.2 Problem 2

0.2.1 Algorithm

[ ]:
```
def minDistance(message, a, b, k):
    n = len(message)
    dp = [[[float('inf') for x in range(k+1)] for y in range(k+1)] for z in␣
 ↪range(n+1)] # initialize everything to infinity
    dp[0][a][b] = 0 # base case

    for i in range(1, n+1):
        key = getKey(message[i-1])
        for j in range(1, k+1):
            for l in range(1, k+1):
                # left thumb
                dp[i][key][l] = min(dp[i][key][l], dp[i-1][j][l] + dist(j, key))
                # right thumb
                dp[i][j][key] = min(dp[i][j][key], dp[i-1][j][l] + dist(l, key))

    return min(dp[n][i][j] for i in range(1, k+1) for j in range(1, k+1)) #␣
 ↪return minimum value in the last row

# helper functions
def dist(i, j):
    # calculates the distance between two keys i and j

def getKey(char):
    # returns the key that the character is on
```

0.2.2 Correctness

We can define $OPT(i, j, k)$ as the minimum total distance to type the first i characters (left thumb on j, right thumb on k). $OPT(i, j, k) = \min(OPT(i-1, j', k) + \text{dist}(j', j), OPT(i-1, j, k') +$

dist(k', k)) where j' and k' represent the the position of the left/right thumbs at the last step. The base case holds as $OPT(0, a, b) = 0$ since you don't move your thumbs to type 0 characters. The final solution is min($OPT(n, j, k)$ for all j, k), the minimum total distance for the entire message. In this way, the algorithm computes the optimal solution of the problem from the optimal solution of the subproblems, and is therefore correct every step of the way.

0.2.3 Runtime

The runtime of this algorithm is $O(nk^2)$, the breakdown is as follows. In this algorithm we have three nested loops, the outer one (i loop) loops from 1 to n+1 for a total of n loops. Inside this loop there are two more loops that both go from 1 to k+1 for a total of k iterations in each. Therefore the total runtime is $n * k * k \rightarrow O(nk^2)$

0.3 Problem 3

0.3.1 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
```

0.3.2 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.

0.3.3 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: $O(mn + mnC) \rightarrow O(mnC)$

0.4 Problem 4

0.4.1 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
    maxFlow, residualGraph = fordFulkerson(G, s, t)

    # 2. Make list from min-cut using residual graph. (max-flow min-cut theorem)
    minCutEdges = findMinCutEdges(residualGraph, s, t)

    # 3. remove k edges from this list.
    for edge in minCutEdges:
        if k > 0:
            deleteEdge(edge, G)
            k -= 1
        else:
            break

    # 4. return G
    return G

# Helper functions
def fordFulkerson(G, s, t):
    # G: graph
```

```
    # s: source
    # t: sink

    # runs in O(mnC)
    # return: max flow, residual graph

def findMinCutEdges(residualGraph, s):
    # residualGraph: residual graph
    # s: source
    # t: sink
    # return: list of edges in min cut

    # find all nodes reachable from s in residualGraph
    # find all nodes not reachable from s in residualGraph
    # find all edges from reachable nodes to non-reachable nodes that are fully␣
 ↪saturated
    # (implement using DFS which is O(m+n))
    # return this list

def deleteEdge(edge, G):
    # edge: an edge
    # G: a graph
    # return: None

    # iterate over edges in G find e (O(m))
    # delete e from G
```

0.4.2 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. Since all the edges have capacity 1, it doesn't matter which order these edges are removed. Also, in the case that k is larger than the number of edges in the min-cut, removing all the edges will disconnect the graph and make the maximum-flow 0, in either case the max-flow is minimized. This algorithm uses Ford-Fulkerson, a well proven algorithm, to find the maximum flow and residual graph of G. It then identifies the edges in the min-cut by seeing the nodes reachable from s in the residual, nodes not reachable from s in the residual, and then finds all edges directly between these two sets that are fully saturated. By definition, these edges cross the cut. As stated above, by removing k of these edges from G (or until we cannot remove any more), we minimize tha maximum flow.

0.4.3 Runtime

The runtime of this algorith is $O(mn)$. This is because the highest cost step is running the Ford-Fulkerson algorithm which is known to be $O(mnC)$ but since $C = 1$ this simplifies to $O(mn)$. findMinCuts most expensive piece is the DFS search, $O(m + n)$, and deleteEdge iterates over the edges of G which is $O(m)$.

$$O(mn + m + n + m) \rightarrow O(mn)$$