Algorithms & Data Structures II (course 1DL231)
Uppsala University – Autumn 2017
Report for Assignment 3 by Team 60

Mona Mohamed Elamin Nevine Gouda

12th January 2018

1 Controlling the Maximum Flow

1.1 Algorithm Implementation

In this section we attempted to solve the Controlling the Maximum Flow problem by detecting a *sensitive* edge. Where an edge is considered sensitive if decreasing the flow at that edge decreases the overall maximum flow. Thus our approach in attempting to solve this problem is exploiting the max-flow min-cut theorem. Where we did the following:

- Calculate the Graph's residual Flow from the provided flow graph and capacity graph.
- Calculating the minimum cut by creating sets S and T, where S is all the vertices reachable from the source and T is the rest of the vertices.
  Thus $S = v \in V$: there exists a path s $\rightarrow$ v in G_f
  T = V - S
  Where the set S is obtained by Breadth First Search (BFS) to try and find the vertices reachable from the sink (s) in the Residual graph G_f.
- Update the previously created sets by applying the following conditions given that $u \in$S and v $\in T$.
  - If (u, v) $\in$ E, must have f(u, v) = c(u, v); ELSE (u, v) $\in E_f \Rightarrow v \in$ S.
  - If (v,u) $\in$ E, must have f(v, u) =0; ELSE $c_f(u, v) = f(v, u) > 0 \Rightarrow (u, v) \in E_f \Rightarrow v \in S$.
  - *If (u, v), (v, u) $\notin$ E, must have f(u, v) = f(v, u) = 0.*
- Finally an edge is considered a sensitive edge there exists an edge between the sets S and T.

Where our attempt can be found in Listings 1, 2 and 3. And it can also be found attached under sensitive.py.

1.2 Complexity

The worst-case complexity is: $|V^3|$.
That is because the BFS as a complexity of $|V^2|$, which is called inside a loop that iterates over all the nodes as well. Leading to the V^3.

2 Reliable Communications

2.1 Algorithm Implementation

In this section we attempted to solve the Reliable Communications problem by finding the path with the least probability of failure. Where each edge has a probability of failure as its weight. And the lower the probability for a path from u, v the better reliability of the communication. Thus this problem can be considered as a shortest path problem where the weights are probabilities.However since the probability of the whole path P is the multiplication of *(1-p(e))* for all the edges in path P, thus some adjustments/ pre-processing on the graph should be made before implementation a shortest path algorithm. Thus our approach in attempting to solve this problem is the following:

- Convert each edge's e weight in the graph from
  a probability p(e), where $0 \leqslant p \leqslant 1$ to $w(e) = -log(p(e))$. Thus $0 \leqslant w(e) \leqslant \infty$.

- Since the weights are now converted to normal weights instead of probability, we can sum the weights across the path instead of multiplying. That is due to the logarithm's proprieties.

- Then we perform the Dijkstra algorithm to find the shortest path from the source to all the nodes until the destination is reached.

Where our attempt can be found in Listings 4 and 5. And it can also be found attached under reliable_communications.py.

2.2 Complexity

The worst-case complexity is: $|V^3 + E|$.
That is because the part where we update the edges weights with the log operation iterates over all the edges which leads to the $|E|$.While the minDistance as a complexity of $|V|$, which is called inside two loops where each loop iterate over all the nodes. Leading to the V^3.

3 The Party Seating Problem

3.1 Problem Formula as Graph Problem

As a graph problem, each guest represents a node in an un-directed graph and the list known[g] is the nodes connected directly to g. The problem is finding an algorithm that divide all nodes into two sets such that each set contains independent nodes i.e. for any u and v in the same set there is no edge (u,v). Therefore the problem is to check if the graph is bipartite or not and if it is bipartite the two disjoint sets must be returned.

3.2 Algorithm Implementation

The solution is designed such that we have list called assigned_table which contains table assigned to each node (guest). initially all nodes are not assigned to any table. then the algorithm visit all nodes, for each of these nodes the following is done:

- If the node is not assigned to any table it will be assigned to one table.Then check its known nodes, if the known node is already assigned to the same table then there is no

disjoint sets (not bipartite). Otherwise, if the known node hasn't a table it will be assigned to the other table and its known nodes (known of known node) will be checked to make sure no one of them has the same table.

- Else if the node is already assigned to one table nothing is done.
- When the node assigned to table the list assigned_table is updated to indicate the table assigned to each node, and then this is used to obtain the two duplicate free lists.

Where our attempt can be found in Listings 6. And it can also be found attached under reliable_communications.py.

3.3 Complexity

Next is the calculation of the time complexity. 'l' represent the sum of the lengths of duplicate free list for all nodes. For simplicity first we can consider that the graph is strongly connected then:

- The outer loop that visit each node will have time complexity —known— * c.because the condition which check the assignment of the node will only verified once and all other assignment is done through it.
- Then the while loop the time complexity will be the summation of the length of known list for each node. because for each node we will check the all nodes in its list therefore time complexity is 'l'.
- Then the time complexity to form the two disjoint sets is equal to number of nodes (guests) which is $|known|$.
- Therefore, the overall complexity is O($|known| + l$).

While for other type of graph:

- The complexity of outer loop combined with the complexity of inner loop such that for each node not assigned yet the complexity will be the summation of its length and its neighbor length. for example if we have nodes g1,g2,g3,g4,g5 where g1,g2,g3 are connected and g4,g5 are connected. then the complexity for g1 = $|known[g1]| + |known[g2]| + |known[g3]|$ and for g2 and g3 it is 'c' because of the assignment condition. the same for g4 and g5. Therefore, the total complexity to assign nodes to table is 'l'.
- Then the time complexity to form the two disjoint sets is equal to number of nodes (guests) which is $|known|$.
- Therefore, the overall complexity is O($|known| + l$).

4 Honour Declaration

We hereby certify that this report and all its uploaded attachments were produced solely by our team, except where explicitly stated otherwise and clearly referenced. We hereby state that each teammate can individually explain any part starting from the moment of submitting our report, and that our report and attachments are not and will not be freely accessible on a public repository.

5 Code Listings

```
1  def BFS_reachable(G_f, source, destination, V):
2      """
3      Sig: graph G(V,E), vertex, vertex, int ==> bool
4      Pre: the source and destination exist in G_f.
5      Post: Using BFS it returns True if the destination is reachable from the source in the given /
           graph G_f, else returns False
6      Ex: sensitive(G,0,5,F) ==> (1, 3)
7      """
8      visited = [False] * V
9      queue = [source]
10     visited[source] = True
11     while queue:
12         # Variant: queue, visited
13         # InVariant: G_f
14         n = queue.pop(0)
15         # then return true
16         if n == destination:
17             return True
18         # Else, continue to do BFS
19         for i in G_f[n]:
20             # Variant: i
21             # InVariant: G_f
22             if visited[i] is False and G_f[n][i] != 0:
23                 queue.append(i)
24                 visited[i] = True
25     return False
```

Listing 1: Controlling the Maximum Flow - Implementation of BFS Search

```
1  def sensitive(G, s, t, F):
2      """
3      Sig: graph G(V,E), vertex, vertex, vertex[0..|V|−1, 0..|V|−1] ==> vertex, vertex
4      Pre: The graph is a directed graph and the capacities in the graph are non−negative.
5      Post: Returns an edge in the format of (u, v) if the edge between u, v is a sensitive edge. And /
           will return None, None if it doesn't exist
6      Ex: sensitive(G,0,5,F) ==> (1, 3)
7      """
8      # G_f is the residual flow
9      G_f = {}
10     for node_i in F:
11         # Variant: node_i, G_f
12         # InVariant: F
13         for node_j in F[node_i]:
14             # Variant: node_j, G_f
15             # InVariant: F, G
16             flow = F[node_i][node_j]
17             cap = G[node_i][node_j]["capacity"]
18             residual = cap − flow
19             if node_j not in G_f:
20                 G_f[node_j] = {}
21             if node_i not in G_f:
22                 G_f[node_i] = {}
23             G_f[node_j][node_i] = flow
24             G_f[node_i][node_j] = residual
25
26     S = []
27     for i in G_f.keys():
28         # Variant: i, S
29         # InVariant: G_f
30         if i == s:
31             S.append(i)
32             continue
33         is_reachable = BFS_reachable(G_f=G_f, source=s, destination=i, V=len(G.nodes()))
34         if is_reachable:
35             S.append(i)
```

Listing 2: Controlling the Maximum Flow - Implementation Continued

```
    S = set(S)
    T = set(G.nodes()) − S
    for u in S:
        # Variant: u, S, T
        # InVariant: G, F
        for v in T:
            # Variant: v, S, T
            # InVariant: G, F
            if (u, v) in G.edges():
                if F[u][v] != G[u][v]["capacity"]:
                    T.remove(v)
                    S.add(v)
            if (v, u) in G.edges():
                if F[v][u] != 0:
                    T.remove(v)
                    S.add(v)
    for u in S:
        # Variant: u
        # InVariant: S, T
        for v in T:
            # Variant: v
            # InVariant: S, T
            if (u, v) in G.edges():
                return u, v

    return None, None
```

Listing 3: Controlling the Maximum Flow - Implementation Continued

```
def minDistance(G, dist, visited):
    """
    Sig: graph G(V,E), dic, dic ==> vertex
    Pre: dist and visited consists of keys of vertices in G and distances are non−negatives values.
    Post: Returns the vertex with the minimum distance to reach it
    Example: Given the following edges: [('a', 'b', fp = 4), ('a', 'c', fp = 5.6), ('a', 'd', fp = 1)]
                                        => it will return 'd' since it has the lowest weight
    """
    # Initilaize minimum distance for next node
    min_distance = sys.maxint
    # Search not nearest vertex not in the
    # shortest path tree
    min_vertex = 0
    for v in G.nodes():
        # Variant: v
        # InVariant: dist, visited
        if (v in dist) and (dist[v] < min_distance) and visited[v] == False:
            min_distance = dist[v]
            min_vertex = v
    return min_vertex
```

Listing 4: Reliable Communications - Implementation of Minimum Distance

```
1  def reliable(G, s, t):
2      """
3      Sig: graph G(V,E), vertex, vertex ==> vertex[0..k]
4      Pre: The graph is an undirected connected graph, it doesn't contain self−loops and s is not the /
           same as t.
5      Post: Returns a path from s to t with the lowest failure probability
6      Example: Test Case 1
7      """
8      assert(s in G.nodes() and t in G.nodes())
9      path = []
10     for u, v in G.edges():
11         new_weight = −math.log(1−G[u][v]["fp"])
12         G[u][v]["fp"] = new_weight
13     dist = {}
14     parent = {}
15     visited = {}
16     for node in G.nodes():
17         # Variant: node, dist
18         # InVariant: G
19         if node == s:
20             dist[node] = 0
21         else:
22             dist[node] = sys.maxint
23         visited[node] = False
24
25     for cout in range(len(G.nodes())):
26         # Variant: cout, u, visited
27         # InVariant: G
28         u = minDistance(G, dist, visited)
29         visited[u] = True
30         for v in G.nodes():
31             # Variant: v
32             # InVariant: u, visited
33             if v in G[u] and visited[v] is False and dist[v] > (dist[u] + G[u][v]["fp"]):
34                 dist[v] = dist[u] + G[u][v]["fp"]
35                 parent[v] = u
36     i = t
37     path.append(t)
38     while i != s:
39         # Variant: i, path
40         # InVariant: parent
41         path.append(parent[i])
42         i = parent[i]
43     return path[::−1]
```

Listing 5: Reliable Communications - Implementation

```
1  def party(known):
2      """
3      Sig: int[1..m, 1..n] ==> boolean, int[1..j], int[1..k]
4      Pre: known is a duplicate−free list
5      Post: If the input can be split into Two lists with no neighbours in the same list
6              then it will return True and the two non−empty lists
7              Otherwise it will return False and 2 empty lists
8      Ex: [[1,2],[0],[0]] ==> True, [0], [1,2]
9      """
10     table0 = []
11     table1 = []
12     assigned_table = [−1]∗len(known)
13
14
15     for n in range(len(known)):
16         # Variant: n
17         # InVariant: len(known)
18         if assigned_table[n] == −1:
19             nodes = []
20             nodes.append(n)
21             assigned_table[n] = 1
22             while nodes:
23                 # Variant: nodes
24                 # InVariant: assigned_table
25                 g1 = nodes.pop()
26
27                 for g2 in known[g1]:
28                     # Variant: g2,
29                     # InVariant: assigned_table
30                     if assigned_table[g2] == −1:
31                         assigned_table[g2] = 1−assigned_table[g1]
32                         nodes.append(g2)
33
34                     elif assigned_table[g2] == assigned_table[g1]:
35                         return False, [], []
36
37     for i in range(len(assigned_table)):
38         # Variant: i
39         # InVariant: assigned_table
40         if assigned_table[i] == 0:
41             table0.append(i)
42         elif assigned_table[i] == 1:
43             table1.append(i)
44
45     return True, table0, table1
```

Listing 6: Party Seating - Implementation