Algorithms & Data Structures II (course 1DL231)
Uppsala University – Autumn 2019
Report for Assignment 3 by Team 55

Li Ju, Georgios Panayiotou

20th December 2019

Part I
Problem 1: Controlling the Maximum Flow

A Exploiting the proof

For the problem given, which we will refer to as $sensitive(G, s, t, F)$, G is a network flow graph with s its source and t its sink, and F is a calculated maximum flow array, where $F[a][b]$ is an integer flow amount over the edge (a, b). In this task, one of the sensitive edges in graph G is supposed to be returned. Our solution is: all edges which are in the collection of edges of minimum s-t cut $C = (S, T)$ ($s \in S$ and $t \in T$), are sensitive and returning any of these edges will solve the task. The justification of the solution is shown following:

1. According to max-flow min-cut theorem, for G, if a flow F is maximum, $val(F) = cap(S, T)$ while (S, T) is the minimum cut of the graph $\implies$ any reduction of $cap(S, T)$ will reduce $val(F)$;

2. $cap(S, T) = \sum_{e \in (S,T)} cap(e)$ $\implies$ reduction of capacity of any edge in (S, T) will reduce $cap(S, T)$ and $val(F)$ correspondingly $\implies$ any edge in (S, T) is sensitive. Moreover, according to max-flow min-cut theorem, there is guaranteed that at least one sensitive edge exists, therefore it is not possible to return a result with (None, None).

Notation:
G: given graph;
E: edges of given graph;
(S, T): collection of edges in the cut between S and T part of a graph;
$val(F)$: flow value of the given flow;
$cap(S, T)$: flow capacity between S and T part of a graph;
$cap(a)$: flow capacity of node a.

B Algorithm design

The algorithm for $sensitive(G, s, t, F)$ problem is designed as following:
1. Firstly we create a function named $res_graph(G, F)$ to construct the residual graph of given

graph G with its flow F;

2. Then a function named $explore(G, node, reach_dict)$ is created to explore all reachable nodes from a given node in a DFS-like approach. Parameter $reach_dict$ is the dictionary recording the reachable nodes, in which reachable nodes are marked as 1 while unreachable nodes and unexplored are noted as -1;

3. With the above two functions, we construct the residual graph rG of given graph G and its maximum flow F and initialize a *reach_list* with all nodes noted as -1. Then from the resource node s we start to explore the graph to find all reachable nodes from s in rG. The collection of all reachable nodes from s are noted as S while all unreachable nodes collection are T. According to Ford–Fulkerson algorithm, (S, T) is the minimum cut;

4. Then we check all edges to record saturated nodes whose flow equals its capacity in a list named *satur_edges*. Saturated edges whose start node is in S and ending node is in T are sensitive. We return the first edge which satisfies this condition.
The implementation of the algorithm is shown as following:

```
def sensitive(G, s, t, F):
    """
    Sig: graph G(V,E), int, int, int[0..|V|-1, 0..|V|-1] ==> int, int
    Pre: G: a directed graph, in which every edge has its capacity;
          s: flow starts from node s; t: flow ends at node t;
          F: the given max flow matrix, in which flows each edge holds are stored
    Post: return a sensitive edge by giving its starting node and ending node
    Ex: sensitive(G,0,5,F) ==> (1, 3)
    """
    ## Function to construct residual graph of given graph with its flow
    def res_graph(Graph, Flow):
        """
        Sig: graph G(V,E), int[0..|V|-1, 0..|V|-1] ==> graph eG(V, E)
        Pre: G: a directed graph, in which every edge has its capacity;
              F: the given max flow matrix, in which flows each edge holds are stored
        Post: return the residual graph of given graph and flow
        Ex: res_graph(G,F) ==> rG
        """
        rG = Graph.copy()
        ## update the copy of original graph edge by edge
        # Loop variant: number of nodes in G not added to rG decreases
        for start_node in Flow:
            # Loop variant: number of nodes in a flow starting from start_node not /
                checked decreases
            for end_node in Flow[start_node]:
                ## add the reversed edge with capacity of current flow
                rG.add_edge(end_node, start_node, capacity = /
                    Flow[start_node][end_node])
                ## update the original edge's capacity to original capacity minus flow
                rG[start_node][end_node]['capacity'] -= Flow[start_node][end_node]
        return rG

    ## build a function to explore all reachable nodes from a given node,
    ## reach_dict is the dictionary recording the reachable nodes,
    ## in which reachable nodes are marked as 1 while unreachable nodes are noted /
        as -1
    def explore(G, node, reach_dict):
        """
        Sig: graph G(V,E), int, dictionary ==> dictionary
```

```
37        Pre: G: a directed graph which is the residual flow graph for a graph and /
              its flow;
38             node: node we start to explore
39             reach_dict: a dictionary in which all nodes are discovered or not /
                   from node
40             are recorded
41        Post: return the dictionary in which every node are reachable or not are /
              recorded:
42             if reachable, it is noted as1 otherwise it's noted as -1
43        Ex: explore(G,s,reach_dict) ==> reach_dict
44        """
45        reach_dict[node] = 1
46        ## a DFS-like approach, to explore each node directly linked with current /
              node
47        # Loop variant: number of nodes in graph G not checked decreases
48        for each in G[node]:
49           if G[node][each]['capacity'] > 0 and reach_dict[each] == -1:
50              # Recursive variant: number of unexplored nodes in G decreases
51              explore(G, each, reach_dict)
52        return reach_dict
53
54     ## construct the residual graph rG of given graph G
55     rG = res_graph(G, F)
56     ## build a the status dictionary and mark every node in rG as -1,
57     ## which means unreachable (currently)
58     reachable_dict = dict.fromkeys(list(rG.nodes), -1)
59     ## explore the graph from the the node s, to find all reachable nodes
60     # Recursive variant: number of nodes in residual graph unexplored decreases
61     reachable_dict = explore(rG, s, reachable_dict)
62
63     ## find all saturated edges, whose capacity equals the flow it holds
64     satur_edges = []
65     # Loop variant: number of nodes in flow F not checked decreases
66     for start_node in F:
67        # Loop variant: number of nodes in a flow starting from start_node not /
              checked decreases
68        for end_node in F[start_node]:
69           if F[start_node][end_node] == G[start_node][end_node]["capacity"]:
70              satur_edges.append([start_node, end_node])
71
72     ## NOTICE: not all saturated edges are sensitive edges -- only those which are /
           in minimum
73     ## cut are sensitive
74     sens_edges = []
75     # Loop variant: saturated edges not checked decreases
76     for each in satur_edges:
77        ## to find which saturated edges are among the minimum cut
78        if reachable_dict[each[0]] == -1 and reachable_dict[each[1]] == 1:
79           sens_edges.append(each)
80     ## return any edge of sensitive edges: because of the theorem of /
           maxflow-minicut, it is
81     ## guaranteed that there is at least one edge is sensitive
82     return sens_edges[0]
```

C Time Complexity Analysis

The time complexity of the algorithm above is analyzed as following:
1. Firstly a residual flow of G is created: in this step, all edges are visited and for each edge a

constant time is required. Therefore, the time complexity of this step is of $\mathcal{O}(|E|)$;

2. From s we explore the residual graph rG: as we only explore from one node following edges, therefore the time complexity of this step is upbounded by $\mathcal{O}(|E|)$;

3. Then all edged are checked to find all saturated edges, while each edge could be examined in a constant time. Also the time complexity of checking which saturated edges are in minimum cut is also upbounded by the number of saturated edges. So the time complexity of this step is of $\mathcal{O}(|E|)$;

Therefore, the overall time complexity of the algorithm is of $\mathcal{O}(|E|)$.

Part II

Problem 2: The Party Seating Problem

A The Party Seating Problem as a graph

The Party Seating Problem can be expressed as a graph problem. Since the goal is to retrieve a table arrangement such that no two people in the same table know each other, we can create a graph where every person is represented with a node, and if a person g_1 knows another person g_2, so $g_2 \in known[g_1]$, there is an edge (g_1, g_2).

The problem can be reformulated as following: Given a graph $G(E, V)$, if there exist a set of nodes A and its complement $B = E \setminus A$, satisfying that all edges are among (A, B), cut between A and B?

B Algorithm design for Party Seating Problem

For this problem, a DFS-like algorithm is designed to split the graph G into two sets while all edges are cut between two sets, A and B. As problem described, there are no any two connected nodes which are supposed to be arranged in one set, so along the edge we arrange.

If there are any nodes which are still unexplored, we choose a random unexplored node a, put it into set A and check all its edges. Following each edge of a,
1. if the node connects with a node which is not assigned, put this node in the opposite set B and explore this node recursively;

2. if the node connects with a node which has been assigned in the opposite set B, do nothing;

3. if the node connects with a node which has been assigned in the set A, return False.

If every node can be put into one of 2 sets perfectly without any contradiction, the arrangement is available, otherwise contradiction (two arranged nodes which are in the same set are connected) arises and a False is returned.

The implementation of the algorithm is shown as following:
1. A status dictionary is created to record the status of each node, in which all nodes are keys and all values are set as 0 initially. In this dictionary, the value 1 of a node means the node is arranged in set A, -1 means it is in set B and 0 means it is not arranged. Also a dictionary named $unarranged_dict$ is defined to record nodes which are not arranged yet.

2. A function named $arrange(node, parent)$ is defined, to arrange a node and its children. If its parent is None, the node is put into set A, otherwise it is put into the set different from its parent's set. Then its connected nodes are saved to be arranged after check which set they are arranged or not arranged yet.

3. From the unarranged node dictionary, we arrange all nodes with function we defined in step 2. If all nodes are assigned without any contradiction, return the two sets, otherwise return two empty lists.

The code is shown as following:

```
def party(known):

    """
    Sig: int[1..m, 1..n] ==> boolean, int[1..j], int[1..k]
    Pre: a list of list, in which stored the edges between nodes
    Post: a boolean indicating if such arrangement can be made, if True, two list containing the plan are returned,
         otherwise two empty lists are returned.
    Ex: [[1,2],[0],[0]] ==> True, [0], [1,2]
    """
    ## construct a dictionary to store the current status of every node, 0 means the node has not been arranged,
    ## 1 means it is set in the list1, while -1 means it is set in the list2. The initial status is all nodes are 0.
    status_dict = dict.fromkeys(range(len(known)), 0)

    ## A dictionary for storage of unarranged nodes:
    ## the reason for using dictionary instead of list is that the complexity of dictionary's deletion is of constant time,
    ## removal of an arranged node requires operated |nodes| times, this is used for reduction of time complexity.
    unarranged_dict = dict.fromkeys(range(len(known)), 0)

    ## The default result if such arrangement is available is True
    result = True

    ## the function arrange is defined to arrange the given node and its children recursively
    def arrange(node, parent):
        ## update the status of current node in status_dictionary according to its parent's status: opposite to its
        ## parent's status
        if parent is None:
            status_dict[node] = 1
            linked_nodes = known[node]
        else:
            status_dict[node] = -status_dict[parent]
            linked_nodes = known[node]
            linked_nodes.remove(parent)

        ## delete the current node from dictionary of unarranged nodes
        unarranged_dict.pop(node)

        ## The default result if such arrangement is available is True, if every node is put with no contradiction,
        ## it keeps True
        result = True

        ## arrange given node's children
        if linked_nodes is not None:
            for each in linked_nodes:
                ## if contradiction occurs, end the loop
                if result == True:
                    ## a child's status equals its own status: contradiction occurs
                    if status_dict[each] == status_dict[node]:
                        result = False
                    ## the child has not been arranged: recursively arrange the child
                    elif status_dict[each] == 0:
```

```
                    # Recursive variant: number of not explored nodes in known / decreases
                    result = arrange(each, node)
        return result

    ## arrange all the unarranged nodes: if contradiction occurs, end the loop
    # Loop variant: number of not arranged nodes in known decreases
    while result is True and len(unarranged_dict) > 0:
        # Recursive variant: number of not arranged nodes in known decreases
        result = arrange(unarranged_dict.keys()[0], None)
    table1, table2 = [], []
    if result == True:
        ## traverse the status dictionary: nodes whose value are 1 is put in list1, / while those whose value is -1
        ## are appended in list2
        for each in status_dict:
            if status_dict[each] == 1:
                table1.append(each)
            else: table2.append(each)
    return result, table1, table2
```

C Time Complexity Analysis

The time complexity of the algorithm above is analyzed as following:
1. Firstly two dictionaries are created, status dictionary and $unarranged_dict$: for each node a constant time will be cost, therefore, this step will is of the time complexity of $\mathcal{O}(|V|)$;

2. Then nodes are to be arranged: in the worst case, each nodes are well-put in sets following each edge. In another word, all nodes and edges are visited. Each visit takes a constant time and number of visit is of $\mathcal{O}(|V| + |E|)$. Therefore, this step is of the time complexity of $\mathcal{O}(|V| + |E|)$;

3. If all nodes are arranged, we need to return the right arrangement: if this step, each node in status dictionary is visited, while each visit takes a constant time, therefore the time complexity of this step is of $\mathcal{O}(|V|)$.

Therefore, the overall time complexity of the as-designed algorithm is of $\mathcal{O}(|V| + |E|)$. If stated in the original problem's notation, it is of $\mathcal{O}(|known| + \ell)$.

D Generalization of the Party Seating Problem

To retrieve a solution for a more general version of the party seating problem, we can create an algorithm that reduces the problem to a maximum flow problem, and specifically, to a bipartite matching. This works, since we have to match the guests with a table in such a way that no two members of the same group are in the same table. The solution would be a maximum flow on a graph constructed for that purpose.

To do this, we construct the network flow graph $G = (V, s, t, E)$, with s and t being a source node and a sink node respectively.

We create the set of nodes $V = A \cup B \cup \{s, t\}$ as follows:
A: a node for every group, named g_i, $0 < i \leq p$.

B: a node for every table, named t_i, $0 < i \leq q$.

We add an edge $e = (a, b)$ to E with capacity $c(e)$ if they satisfy one of the following conditions:
- $a = s$ and $a = g_i$: $c(e) = \mathit{Group}[i]$
- $a = g_i$ and $b = t_j$: $c(e) = 1$
- $a = t_j$ and $b = t$: $c(e) = \mathit{Table}[j]$

Running the Ford-Fulkerson algorithm on G will return a maximum flow f, according to the edge between A and B in f, we can get the arrangement of part table: if there is a flow between g_i and t_j, 1 guest from group i is assigned to sit on table j.