Algorithms & Data Structures II (course 1DL231)
Uppsala University – Autumn 2023
Report for Assignment 3

Yueyuan Huang

8th December 2023

Part 1
Controlling the Maximum Flow

A Detecting Sensitive Edge

As shown in 1.

```
42 def sensitive(G: Graph, s: str, t: str) -> Tuple[str, str]:
43     '''
44     Pre: G is a flow network reaching maximum flow, with s as
45          its source and t as its sink.
46     Post:
47     Ex: sensitive(g1, 'a', 'f') = ('b', 'd')
48     '''
49     Gf = Graph(is_directed=True) # residual network of G
50     for u, v in G.edges:
51         f, c = G.flow(u, v), G.capacity(u, v)
52         Gf.add_edge(u, v, c-f)
53         Gf.add_edge(v, u, f)
54 
55     S = set() # vertices reachable from s in Gf
56     def dfs(u):
57         S.add(u)
58         for v in Gf.neighbors(u):
59             if v not in S and Gf.weight(u, v) > 0:
60                 dfs(v)
61     dfs(s)
62 
63     for u, v in G.edges:
64         if G.capacity(u, v) == 0 or G.flow(u, v) < G.capacity(u, v):
65             continue
66         # Now f(u, v) == c(u, v) and are both positive
67         # if u in S and v in T or u in T and v in S (T := V - S)
68         if not ((u in S) ^ (v not in S)):
69             return u, v
70 
71     return None, None
```

Listing 1: Python function $sensitive(G, s, t)$.

The algorithm starts by computing a residual network G_f of the given graph G. Then it tries to get the minimum cut $\{S, T\}$ by performing a DFS on G_f.

As G has already reached maximum flow f, we will find edges of the minimum cut all loaded to capacity, i.e., $f(e) = c(e), \forall e \in \{S \to T\} \cup \{T \to S\}$. Therefore on G_f if we neglect edges with weight 0, we will get S via a DFS from source s (S being all vertices reachable from s on G_f). Note that performing DFS on G will not work because some vertices unreachable from s on G actually belong to S.

According to the max-flow min-cut theorem, $|f| = |S, T|$. If we reduce $|S, T|$ by decreasing flow on some edge of the minimum cut, the maximum flow $|f|$ would also be reduced. So a sensitive edge could be any edge of the minimum cut with positive capacity.

B Time Complexity

The worst-case time complexity of the algorithm 1 would be $\mathcal{O}(|E| + |V|)$. There are 3 major steps: calculating the residual network G_f, getting the minimum cut $\{S, T\}$ and finding a sensitive edge. The first step costs $\mathcal{O}(|E|)$ as every edge in G is turned into 2 edges in G_f, the second step costs $\mathcal{O}(|E|)$ in worst case since every vertex is visited once at most, and the final step costs $\mathcal{O}(|V|)$ in worst case because it has to check all edges.

C Is Edge at Capacity Always Sensitive?

No, because there might be an alternative path which is not at capacity. Let us consider this scenario: we have G at maximum flow with edges $s \to a, s \to b, b \to a, a \to t$, in which $s \to a$ and $a \to t$ are at capacity. Here $s \to a$ is not sensitive because if we reduce its capacity, we could simply mend the flow from the alternative path $s \to b \to a$, keeping the maximum flow unchanged.

D Real-World Example

A real-world example of a flow network is where the edges are roads and the nodes are intersections. For each edge, its capacity is the maximum amount of traffic it can handle and its flow is the current amount of traffic on it.

In this example, a sensitive edge corresponds to a road which, if its traffic capacity decreases, the overall traffic throughput of this traffic network would be reduced.

Finding sensitive edges in the network could be of importance because we might then avoid unintentionally affecting the overall traffic condition by shutting down some sensitive road for maintenance.

Part 2
The Party Seating Problem

A Problem Formulating

The party seating problem could be formulated as a graph bipartite problem. The relationship between guests could be turned into a graph, with every guest as a vertex and if guest u knows

guest v, there is an edge between u and v. As the knowing relationship is mutual, our graph would be an undirected graph.

Our task is then to find a partition of a given graph G, i.e., separating its vertices into 2 sets S and T so that there is no edge within each set (which means every edge is a cutting edge that connects S and T).

B Judging Valid Seating

As shown in 2.

```
def party(known: List[Set[int]]) -> Tuple[bool, Set[int], Set[int]]:
    '''
    Pre: known is a list of sets containing integers in range
         [0, len(known)-1] (Could also be empty). If integer u
         is in known[v], then v must be in known[u].
    Post:
    Ex: party([{1, 2}, {0}, {0}]) = True, {0}, {1, 2}
    '''
    if not known: # No guest
        return True, set(), set()
    V, G = range(len(known)), known # vertices, graph

    S, T = set(), set() # 2 sets of vertices (of different color)
    visited = lambda u: u in S or u in T
    # color vertices reachable from s
    def bfs(s):
        S.add(s) # the starting vertex could be in either S or T
        q = [s]
        while q:
            u = q.pop(0)
            for v in G[u]:
                if (u in S and v in S) or (u in T and v in T):
                    return False # conflict
                if not visited(v):
                    S.add(v) if u in T else T.add(v)
                    q.append(v)
        return True
    # try to color every vertex in G (until conflict)
    for u in V:
        if not visited(u) and not bfs(u):
            return False, set(), set()

    return True, S, T
```

Listing 2: Python function *party*(*known*).

The algorithm actually solves the bipartite problem as 2-coloring: it tries to color the vertices with 2 colors, with the constraint that adjacent vertices must have different color. The coloring is implemented by BFS.

C Time Complexity

The time complexity of algorithm 2 is $\mathcal{O}(|known| + l)$.

To see this, we could find it colors the graph G by performing a BFS on every connected component of G (Note that G could be not connected). In a single BFS, the time complexity

of it would be $\mathcal{O}(|V_i| + |E_i|)$, where $|V_i|$ and $|E_i|$ are the number of vertices and edges of the ith connected component, which sums to $\mathcal{O}(|known| + l)$ since BFS would have eventually travelled to every corner of G.

D Modification

We can modify the party seating problem to more than two tables in the following manner. The party is attended by p groups of guests, and there are q tables. All members of a group know each other; no guest knows anyone outside their group. The sizes of the groups are stored in the array $Group$, and the sizes of the tables in the array $Table$. The problem is to determine a seating arrangement, if it exists, such that at most one member of any group is seated at the same table.

Let us formulate this as a maximum-flow problem. We first construct a flow network in this manner: we start by treating each guest as a vertex, then we add an edge from each guest to every guest in groups behind theirs (here behind means $\succ$, which is based on an arbitrary ordering we give to the groups). we finish the construction by adding a source s having edges to all guests and a sink t having edges from all guests to it. Every edge in this flow network has unit capacity and initially zero flow.

Then the modified party seating problem actually means to get the maximum flow on the flow network under the constraint that every augmenting path we found must associate with some table i, i.e.,

$$\forall \text{ augmenting path } P, \exists i \in \{1, 2, \ldots, q\} \text{ so that } P.length = Table[i] + 1,$$

in which $P.length$ is the number of edges along path P. We would arrange guests on the augmenting path P to table i. Finding all of the q augmenting path would mean a proper seating is achieved.