Algorithms & Data Structures II (course 1DL231)
Uppsala University – Autumn 2021
Report for Assignment 2 by Team 15

Jakob Nordgren Aljaz Kovac

15th December 2021

Part 1
Controlling the Maximum Flow

A Design and implement an efficient algorithm

The algorithm was implemented following the proof of the Max-flow min-cut theorem. The theorem states that if f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent:

- f is a maximum flow in G.
- The residual network G_f contains no augmenting paths.
- $|f| = c(S, T)$ for some cut (S, T) of G.

We will now succinctly describe the steps of the proof, and in the following paragraph describe how we followed those steps to arrive at a solution for the algorithm. The proof follows like this:

Since G is a flow network with maximum flow f, we know that its residual network G_f will contain no augmenting paths. If G_f did contain an augmenting path then f would not be the maximum flow and there would be a contradiction. If we now assume that G_f contains no augmenting path and define $S = \{v \in V : \text{ there exists a path from } s \text{ to } v \text{ in } G_f\}$ and $T = V - S$, then we can consider vertices $u \in S$ and $v \in T$. "If $(u, v) \in E$, we must have $f(u, v) = c(u, v)$, since otherwise $(u, v) \in E_f$, which would place v in set S. If $(u, v) \in E$, we must have $f(u, v) = 0$, because otherwise $c_f(u, v) = f(v, u)$ would be positive and we would have $(u, v) \in E_f$, which would place v in S. Of course, if neither (u, v) nor (v, u) is in E, then $f(u, v) = f(v, u) = 0$."(Quoted from page 724 of CLRS3 [1].) Therefore, by Lemma 26.4 (see page 721 of CLRS3 [1]), we have: $|f| = f(S, T) = c(S, T)$. In simpler terms, there must exist a cut whose capacity will equal the max flow of G, or $|f| = c(S, T)$. We call this cut the "min cut".

What this means in terms of our algorithm is the following: if we can find the min cut S of G, and observe the edges that connect all vertices in S with all the vertices in T where $T = V - S$, then we have found our "sensitive edges" (a sensitive edge, as described in the assignment, is an edge whose capacity cannot be reduced without affecting the maximum flow of the network). In order to find the min cut of G with maximum flow f, we need to calculate a residual network G_f. If we follow the edges in G_f from the source s, using either breadth-first or depth-first search, and save them in S, then we will eventually arrive back at s, and all the vertices that we visited, stored in S, will be our min cut. This holds true from the proof of the second statement of the theorem. Intuitively, this makes sense because if a vertex (u_f, v_f) in a residual network G_f represents the difference between the flow of edge (u, v) in G and the capacity of edge (u, v) in G then there will be no edges going from the min cut S to T as the flow f of those edges equals their capacity: $|f| = c(S, T)$, as stated by the Max-flow min-cut theorem.

For the implementation of this algorithm in Python, please see Listing 1.

B Time complexity of the algorithm

The worst-case time complexity of our algorithm is the following:

- We first calculate the residual graph G_f of $G = (V, E)$, which is a breadth-first search algorithm that takes $\mathcal{O}(|E|)$ time.
- We then run a breadth-first search on G_f, which again takes $\mathcal{O}(|E|)$ time.
- In the end, we need to find a sensitive edge, which involves checking if an edge from any vertex in the min-cut S to any vertex in T, where $T = V - S$, exists in E. This algorithm has a time complexity of $\mathcal{O}(|E|)$.

The worst-case time complexity for the entire algorithm is therefore:

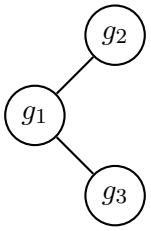
$$\mathcal{O}(|E| + |E| + |E|) = \mathcal{O}(3|E|) = \mathcal{O}(|E|)$$

Part 2

The Party Seating Problem

A Formulate the problem as a graph problem

For n number of guests, the set $\{known[g_1], known[g_2], ..., known[g_n]\}$ can be seen as an adjacency list for an undirected graph $G = (V, E)$ where we have a vertex $v \in V$ for each guest g. For any guests g_1 and g_2, represented by vertices u and v respectively, there is an edge $(u, v) \in E$ if $g_2 \in known[g_1]$. For example, let $n = 3$ and $known[1] = \{2, 3\}, known[2] = \{1\}, known[3] = \{1\}$. The corresponding graph would be:

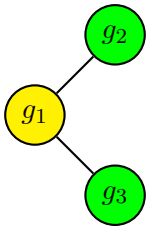


What we want is a way to find two sets of nodes V_1 and V_2 such that there is no edge (u, v) where $u \in V_1$ and $v \in V_2$. This is analogous to dividing guests into 2 groups where no guests in each group know each other. We hope the reader is now convinced that G successfully models the party seating problem, so in the following sections we will only discuss the solution to the problem in terms of its graph representation.

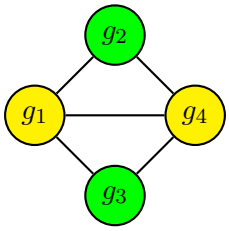
B Design and implement an efficient algorithm

To find the two subsets in a graph G as described in part A we do a depth-first search starting from any node. Whenever we visit a new node we add it to one of two subsets V_1 or V_2. When deciding what subset to add it to, we alternate between the two. So if the DFS first visits node g_1 it is added to V_1 and its neighboring nodes are added to V_2, and so on.

The following graph illustrates such a traversal where nodes added to V_1 are colored yellow and nodes added to V_2 are green. Here, the DFS visits vertices in the order g_1, g_2, g_3:



If we encounter a node already visited, it must not be in the same set as the previous node. If that is the case, then there is no solution, since no matter how we would partition the graph, there would always be at least 2 vertices in the same set with an edge between them. For example, consider the following graph, where there is no solution since the edge (g_1, g_4) connects two nodes in the same set:



Lastly, before discussing implementation, we note that the graph G may consist of several unconnected subgraphs. Among the guests, there may be subgroups, where no guest from one subgroup knows anyone from another group. Because of this, every time the DFS is done, we check if there is any unvisited vertex in G, in which case we restart the DFS from that vertex.

Listing 2 shows the Python implementation of this algorithm. Here we define a nested function `seat(u, table_flag)` that is a top down recursive DFS. The parameter `table_flag` is used to indicate which of the two sets the current node should be added to. On every recursive call, we pass the negation of the flag (line 75) so that we alternate between the two sets. If we encounter a neighboring node in the same set as the current `table_flag`, we know there is no solution and exit the DFS.

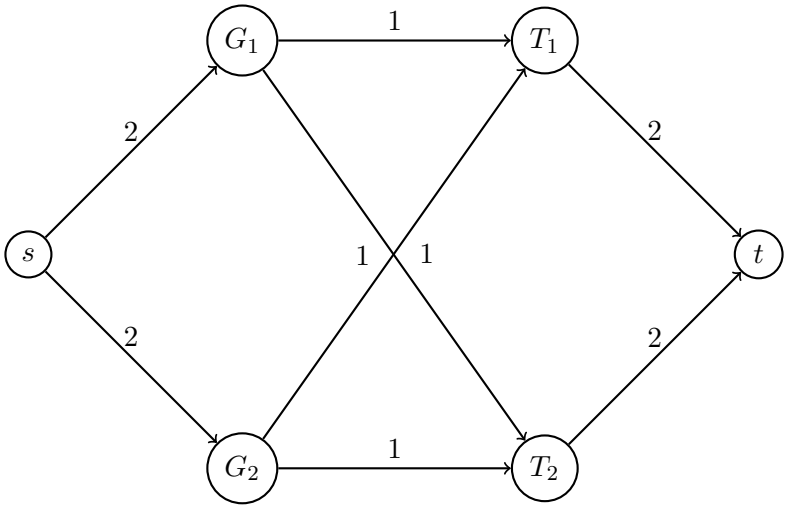
C Time complexity

The algorithm consists of a DFS which is performed once for each disconnected component of G. In the DFS function `seat`, we have a for loop iterating over all neighbors of a node and a recursive call that is made only if the neighboring vertex has not yet been visited. Furthermore, the DFS will visit every node in a connected component of G, so no node will be visited more than once. The number of neighbors of a node is equal to the sum of outgoing edges from that node, so the `seat` function will contribute $\mathcal{O}(|E|)$ to the total complexity. The outer block has a for loop that iterates over $V \in G$. The total complexity of the algorithm is therefore $\mathcal{O}(|V| + |E|)$. As stated in part A, we have one node for each guest, so $|V| = |known|$. Furthermore, there is one edge for each relation among guests, but a relationship between guests g_1 and g_2 is represented twice in $known[]$, that is one entry in $known[g_1]$ and one entry in $known[g_2]$. We therefore have $|E| = \ell/2$ where ℓ is the sum of the lengths of the lists in the set $known[]$. The total time complexity is thus $\mathcal{O}(|known| + \ell/2) = \mathcal{O}(|known| + \ell)$

D A formulation of the modified problem as a maximum-flow problem

Let $G_1, G_2, ...G_p$ be the different groups of guests and $T_1, T_2, ...T_q$ the different tables. We can draw a graph where we have one vertex for each G_n and one for each T_m, where $0 < n \leq p$ and $0 < m \leq q$. At most one guest from each group can sit at a given table, so add edges of capacity 1 from each G_n to each T_m. Next add a vertex s, which is a super source, with edges to each G_n. Now give each edge the capacity equal to the size of that group, so that $c(s, G_n) = |Group[n]|$. Lastly, add a vertex t (a super sink) that has incoming edges from each T_m. Give each edge the capacity equal to the size of the table, so $c(T_m, t) = |Table[m]|$. We have now constructed a flow network $G = (V, E)$, where at most one guest from each group can "flow" to a given table. The sum of the outgoing capacities from s will be equal to the total number of guests. A solution to the problem exists if and only if $|f| = Total\ number\ of\ guests$. This is analogous to seating each guest at some table, with the restriction that at most one guest from each group can sit at one table.

The following is an example of such a flow network, where we have $Guests = [2, 2]$ and $Tables = [2, 2]$. The maximum flow can be found using the Ford-Fulkerson algorithm. If $|f| = number\ of\ guests$, then the arrangement can be found by simply listing all edges (G_n, T_m) where $f(G_n, T_m) = 1$. In the example graph, the maximum flow would be 4 and the matching would be $M = \{(u, v) : u \in G_n, v \in T_n,\ and\ f(u, v) = 1\}$.



References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

```python
def sensitive(G: Graph, s: str, t: str) -> Tuple[str, str]:
    """
    Sig: Graph G(V,E), str, str -> Tuple[str, str]
    Pre:
    Post:
    Ex: sensitive(g1, 'a', 'f') = ('b', 'd')
    """
    # Calculate the residual graph
    G_residual=calculateResidualGraph(G, s)
    queue = []
    visited = []

    def bfs(visited: List, v: str) -> List[str]:
        visited.append(v)
        queue.append(v)
        while queue:
            # VARIANT: nr. of vertices encountered for the first time, which have / not yet had their adjacency lists fully examined
            n=queue.pop(0)
            for neighbor in G_residual.neighbors(n):
                # VARIANT: length of G_residual.neighbors - nr. of iterations
                if neighbor not in visited:
                    visited.append(neighbor)
                    queue.append(neighbor)
        return visited

    # Perform breadth-first search on the residual graph to find the min-cut (all / the vertices reachable from the source)
    S=bfs(visited, s)

    # Set the vertices that are not in the min-cut S into T (T = G - S)
    T=list(set(G.nodes) - set(S))

    # Find a sensitive edge (min-cut edge), meaning an edge from a vertex in the / min-cut (S) to a vertex in the rest of the graph (T)
    for u,v in G.edges:
        # VARIANT: length of G.edges - nr. of iterations
        if (u in S and v in T) or (u in T and v in S):
            return u,v

    return None, None
```

Listing 1: Python implementation of the solution of Problem 1.

```python
def party(known: List[Set[int]]) -> Tuple[bool, Set[int], Set[int]]:
    """
    Sig: List[Set[int]] -> Tuple[bool, Set[int], Set[int]]
    Pre:
    Post:
    Ex: party([{1, 2}, {0}, {0}]) = True, {0}, {1, 2}
    """
    t1, t2, visited = set(), set(), set()
    solution = True

    def seat(u: int, table_flag: int):
        # VARIANT: (number of elements in connected component of 'known') - /
            len(visited)

        # Access the outer variable
        nonlocal solution

        # Select the right table to add to
        if table_flag == 1:
            table = t1
        else:
            table = t2

        # Mark vertex as visited and add it to a table
        visited.add(u)
        table.add(u)

        for v in known[u]:
            # VARIANT: len(known[u]) - (number of loop iterations)
            if v in table:
                # We have found a vertex with edges to two nodes
                # in different sets, so there is no solution
                solution = False
                break
            elif v not in visited:
                # Recursive call where we add to the OTHER table
                seat(v, table_flag * (-1))


    for i in range(len(known)):
        # VARIANT: len(known) - i
        # The graph may not be connected,
        # so we need to try a DFS from each vertex
        if i not in visited:
            seat(i, 1)

    if solution:
        return True, t1, t2
    else:
        return False, set(), set()
```

Listing 2: Python implementation of the solution of Problem 2.