# Algorithms & Data Structures II (course 1DL231)
# Uppsala University – Autumn 2020
# Report for Assignment 3 by Team 23

William EULAU Adam ERIKSSON

14th December 2020

## Part 1
# Controlling the Maximum Flow

## A Introduction

Given a flow network with a source ($s$) and a sink ($t$) where each edge has been given a flow and capacity, an algorithm which identifies a sensitive edge can be implemented. This algorithm along with a short theoretical background and a time complexity analysis of the algorithm will be presented below.

A sensitive edge is defined as an edge whose capacity can not be reduced without compromising the total flow of the network. This implies that the flow through such an edge is equal to its total capacity. However, note that this in no equivalence relationship.

The sensitive edges can be found using a residual network which shows how much flow that can be added or removed at each edge. As described above, no additional flow can be added to a sensitive edge. Hence, flow can only be removed, which is presented in the residual network as a directed arrow in the opposite direction of the flow. If a flow network is maximized in regard to flow (max-flow) so that no more flow can be added to the system, a cut in its residual network splitting the subtrees reachable from $s$ and $t$ respective (a cut called *$s$-$t$ cut*), the connecting nodes in the cut will be sensitive disregarding potential edges without any flow. This can be derived from the **Max-flow min-cut theorem**, stating that a flow network with maximized flow and a residual network without any augmenting paths (simple path from source to sink) will have a minimum cut with with an edge weight sum equal to the networks flow. An example of a flow network with a visualized min-cut is presented in Figure 1.

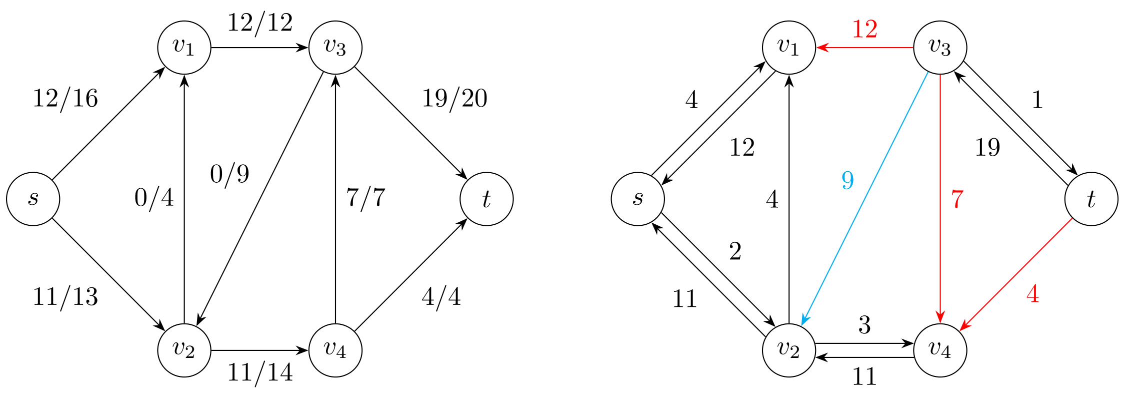

Figure 1: A flow network (left) and its residual network (right) with node $s$ as a source and node $t$ as a sink. The red edges in the residual network are sensitive and the blue edge needs to be excluded from the cut as it has no flow but is a part of the s-t cut. The example is copied from the test code provided by Justin Pearson (justin.pearson@it.uu.se) and his teaching assistants in the course 1DL231 at Uppsala University 2020.

## B Finding a Sensitive Edge

As a first step of the algorithm, it copies the entire input flow network as a skeleton for a residual network. This is done to preform operations on a network without changing the original input network. This step is important later as we discuss the over all time complexity as this step will have the complexity $\mathcal{O}(|E| + |V|)$, which we will argue is the most time complex step of the entire algorithm.

With the flow network copy as a skeleton, a corresponding residual network is created for each network edge comparing its flow and capacity. If it has a flow, the value of that flow will be added to an edge in the opposite direction in the residual network (corresponding to amount of flow that can be removed flow). Furthermore, if the capacity of an edge is grater than its flow, an edge with the weight of the difference between capacity and flow will be added in the same direction as the original edge (corresponding to flow that can still be added to the edge). As this step will repeat until all edges have been checked and added to the residual network, the time complexity of this step is $\mathcal{O}(|E|)$.

```
residual = G.copy() # Copies the Flow network G to use as a skeleton for a /
    residual network, O(|E|+|V|).

for edge in G.edges:
   # Tightly bounded by the number of edges in G; O(|E|).
   # Loops through all edges in the flow network to fill in the edges of the /
       residual network.
   from_node, to_node = edge
   capacity = G.capacity(from_node, to_node)
   flow = G.flow(from_node, to_node)
   residual.remove_edge(from_node, to_node)
   inverted_edge_weight = flow
   if flow != 0: # If condition met for edge(u,v), adds an edge(v,u) in the /
       residual network with weight = flow.
      residual.add_edge(to_node, from_node, weight=inverted_edge_weight)
   if capacity > flow: # If condition met for edge(u,v), adds an edge(u,v) in /
       the residual network with weight = capacity-flow.
      residual.add_edge(from_node, to_node, weight=capacity - flow)
```

```
# The residual network created.
```

Listing 1: Initializing and filling the residual network according to flows and capacities of the input flow network.

Using a modified Depth-first search (DFS) algorithm (explained in further detail in **Assignment 2**), the nodes reachable from $s$ found and declared as members of the $s$-partition (by marking their position as $True$ in a dictionary). All nodes not in this partition are in the $t$-partition. Finally, the algorithm checks all edges in the input graph (network) to see if the nodes connected by each edge are in different partitions, i.e. in the min-cut. It also checks whether the flow of the edge is equal to its capacity (thereby removing any edges without flow, such as the blue edge in Figure 1). If both of these conditions are met, the edge is placed in a list of sensitive edges and the nodes connected by the first edge in the list are returned. If the list of sensitive edges would happen to be empty at this time, it means that the flow network did not contain any sensitive edges and $None, None$ is returned. This part of the algorithm can be divided into two steps, each with different time complexities. The first part where the nodes are either declared as part of the $s$- or $t$-partition will go through each node in the graph and thereby have the complexity $\mathcal{O}(|V|)$. The final step where the list of sensitive edges is filled will have a time complexity of $\mathcal{O}(|E|)$ as it will traverse through all edges.

```
# Find the min-cut
cuts = dict() # cut 's'(True) or 't'(False).
visited = dict() # If a node is visited.
for node in G.nodes: # O(|V|), initialize dictionaries.
   cuts[node] = False # All nodes are set as part of 't'-cut as default.
   visited[node] = False
parent = dict() # initialize parent dictionary
parent[s] = None

# Find all nodes part of the 's'-cut (cuts[n] = True.)
stack = [s]
while len(stack) > 0:
   # At most O(|V|)
   # Loops through each node in the network as long as the stack is not empty.
   # Stats at node 's' and thereafter picks nodes from the stack.
   node = stack.pop() # LIFO.
   visited[node] = True # The node is now visited.
   cuts[node] = True # Part of the 's'-cut.
   neighbors = residual.neighbors(node) # Finds neighboring nodes
   for neighbor in neighbors: # adds neighboring nodes to the stack if they /
       are not parent or already visited.
      if parent[node] != neighbor and neighbor != t and not visited[neighbor]:
         parent[neighbor] = node
         stack.append(neighbor) # Adds neighbor to the stack.

sensitive_edges = [] # Initiates a list to keep track of any sensitive edges /
    found
for edge in G.edges:
   # Tightly bounded by the number of edges in G; O(|E|).
   # For each edge, checks whether it is part of the cut or not.
   from_node, to_node = edge
   if cuts[from_node] != cuts[to_node] and G.flow(from_node, to_node) == /
       G.capacity(from_node, to_node):
      # The edge is sensitive and its nodes are appended to the list of /
          sensitive edges
      sensitive_edges.append(edge)

if len(sensitive_edges) == 0: # Returns None,None if there are no sensitive /
    edges.
```

```
        return None, None
    else:
        return sensitive_edges[0] # Returns the the nodes connected by the first /
            sensitive edge in the list
```

Listing 2: With a DFS, finds the $s$-partition and checks for all edges in the input network if they are part of the minimum $s$-$t$ cut (sensitive edges). Stores sensitive edges in a list and returns the first value of the list if it is not empty.

## C Time Complexity Analysis

As mentioned before, this algorithm has the worst-case time complexity $\mathcal{O}(|E| + |V|)$. Each step following the graph copy has either a complexity of $\mathcal{O}(|E|)$ or $\mathcal{O}(|V|)$, which in any case implies an overall time complexity of $\mathcal{O}(|E| + |V|)$.

# Part 2
# The Party Seating Problem

## A Introduction

Given an array of which guests know each other, a two-table party seating, where no two guests at the same table should know one another, can be formulated as graph problem. In a graph, the guests are represented by nodes and the relation between two guests is represented as an edge. By this representation, a two-color graph coloring principle can be applied to generate a seating arrangement which places guests who do not know each other at the same table, if possible, and will be presented in the following section.

## B Two Table Party

To decide the seating arrangement of guests, the guests and their relationships need to be initialized. In Figure 2, an example is presented, where the guests represented as numbered nodes and relationships as edges.

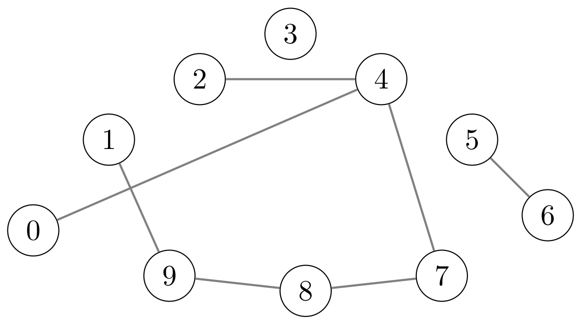

Figure 2: Nine guests attending the party represented as a graph problem. The edges between graphs show the relationship between guests.

This graph problem is a bi-partite one, as the nodes can be divided into two disjoint sets such that every edge (relationship) connects the two sets, if possible. To iterate if there is an edge between two nodes, they should be in different sets in the graph. In implementation and for visual exemplification, the two disjoint sets will be represented as of color green or red. The color of each node corresponds to which table it should be seated at. In code implementation, the graph is initialized in the following manner: a dictionary will store all nodes, and if they have an edge. Then the graph is initialized, and the dictionary updated. If nodes don't have any edges (preciously known guests), it won't technically be part of the graph. However, it is stored in the dictionary has_edge. In the graph example in this report, node 3 would not be part of the graph, but the node will be colored.

```
l = len(known)
graph = Graph(is_directed = False)

has_edge = dict()
for i in range(l): # O(l)
   has_edge[str(i)] = False #initial state - no edge is added

for i in range(len(known)): #O(|known|)
   known_party = known[i] #list
   for person in known_party:
      if person > i: #if index is lower, the edge has already been added to /
          the graph
         graph.add_edge(str(i),str(person))
         has_edge[str(i)] = True
         has_edge[str(person)] = True
```

Listing 3: Initializing the party seating as a bi-partite graph problem.

After the graph is initialized, the nodes will be colored. The identification of disjoint sets will be done using three dictionaries and a stack. The three dictionaries will store if the node has been visited, which set or table or color it belongs to, and if a node has an asserted color. Nodes will get asserted colors opposite of their neighboring node colors. If no color is asserted, the nodes will be initialized with an arbitrary color. A stack, with neighboring nodes appended and popped in a LIFO manner, is used to complete a path from one node. In the example of this report, the path 1 - 9 - 8 - 7 - 4 - 2 - 0 is finished before moving to another node. This is due to if a path is filled in from two ends, a coloring mismatch can occur, and the solvable problem is deemed unsolvable. The stack notation is enabled by a while-loop, running at most $\mathcal{O}(l)$ times.

```
colors = ['G','R']
table = dict() #table 'G' or 'R'
assert_color = dict() #if a node has an asserted color
visited = dict() #if a node is visited
for guest in has_edge: # O(l) initialize dictionaries
   assert_color[guest] = False
   table[guest] = False
   visited[guest] = False

at_node = 0
stack = [str(at_node)]
while at_node < l:
   # at most O(2*l) => O(l).
   # loops from node 0 to the total number of nodes. As long as stack is empty,
   # at_node will increment. If node is taken from stack, it is visited and /
     will not be
   # run through more than once. Ring structures aren't a problem due to /
     visited dict.
   # worst case is that every node is added to the stack and it loops through /
     all nodes twice => O(l)
   if len(stack) == 0 and visited[str(at_node)]: #run another loop
      at_node += 1 #go to the next node
   else: #else run through loop
      if len(stack) > 0: # there is a guest in the stack
         guest = stack.pop() #LIFO
      else:
         guest = str(at_node)
         at_node += 1 #update node
      if not has_edge[guest]: #if node dosn't have an edge, it's not in the /
          graph
         color = colors[int(guest)%2]
         visited[guest] = True
         table[guest] = color
      else: # if it has an edge, it is part of the graph.
         if assert_color[guest] == False:
            color = colors[int(guest)%2]
         else:
            color = assert_color[guest]

         visited[guest] = True
         table[guest] = color
         neighbors = graph.neighbors(guest) #assert colors on neighbors /
             (=known guests)
         opposite_color = colors[colors.index(color) - 1]
         for neighbor in neighbors:
            if not visited[neighbor]:
               if assert_color[neighbor] == False or assert_color[neighbor] == /
                   opposite_color:
                  assert_color[neighbor] = opposite_color #asserted opposite /
                      color
                  stack.append(neighbor)
               else:
                  #logger.debug(f"Return false. length = {l}\n Table: {table} /
                      \n studied node = {guest}, color = {color}, opposite = /
                      {opposite_color}, neighbor = {neighbor} \n assert color = /
                      {assert_color}\n known= {known}")
                  return False, [], []

tableG = []
tableR = []
for guest in table: #O(l) table length l
   if table[guest] == 'G':
      tableG.append(int(guest))
   else:
      tableR.append(int(guest))

return True, tableG, tableR
```
Listing 4: Initializing the party seating as a bi-partite graph problem.

In Figure 3, the example is solved using the above implementation and the nodes are colored. The color represents which table the guests will be seated at.

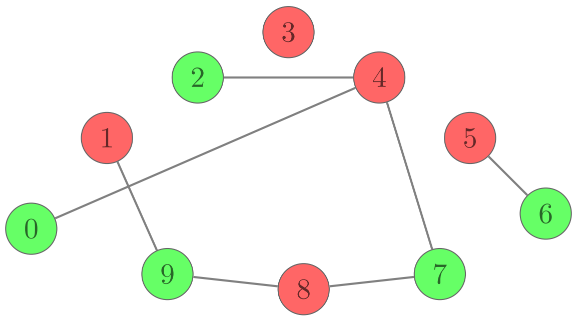

Figure 3: The nine guests with a table placement, represented as color green and red. According to bipartite graph principles, every neighboring node has a different color.

Initializing the nodes take $\mathcal{O}(l)$ time, where $l$ are the amount of guests. Initializing the edges or relationship between the guests takes $\mathcal{O}(|known|)$ time, where *known* is a two-dimensional input list with the relation between all guests. Coloring the graph implies visiting each guest once, and will therefore run in $\mathcal{O}(l)$ time. Initializing the dictionaries will also be executed under the upper bound time complexity. Therefore, the party seating problem will be solved in $\mathcal{O}(l + |known|)$ time.

## C Multiple Table and Group Party

An extended problem of the table seating problem is with multiple tables and groups. Placing p different groups of guests who know each other at q different tables so that all guests are seated with previously unknown guests can be formulate as a bipartite max-flow problem. The formulation is presented in Figure 4 and further explained below.

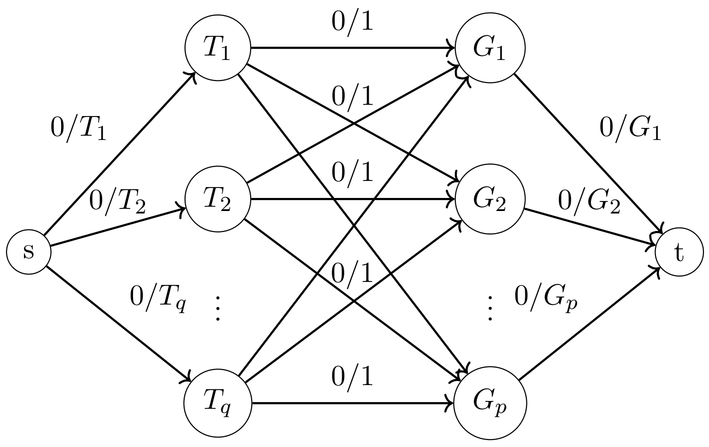

Figure 4: q tables $T_1 \dots T_q$ and p groups $G_1 \dots G_p$ represented as a max-flow problem. The problem is solved if the flow is equal to the sum of $G_1 \dots G_p$.

The two disjoint sets are the tables $T_1 \dots T_q$ and the group of guests $G_1 \dots G_p$. They are connected with directed edges from the tables to the different groups and represent how many guests from the groups will be seated at a given table. All tables have an edge connected to each group with capacity one. With a capacity one, only one guest from each group can be seated at every table to ensure that no guests have previous relationships. A source node s has directed edges to every table with capacities $T_1 \dots T_q$, representing the size, or capacity, of each table. From the groups of guests nodes to the sink node t directed edges run with capacity, $G_1 \ldots G_p$ representing the size of each group, or the number of guests in each group. When the graph has max-flow, if all edges to t are full, all guests have a table assigned to them, and the problem is solved. If not all edges to the sink are filled, the problem is not solvable as not all guests have a table assigned to them.

Part 3

# Code Appendix

## A Controlling the Maximum Flow

```python
from typing import * # noqa
import unittest # noqa
from src.sensitive_data import data # noqa
from src.graph import Graph # noqa
# If your solution needs a queue, then you can use this one:
from collections import deque # noqa
# If you need to log information during tests, execution, or both,
# then you can use this library:
# Basic example:
# logger = logging.getLogger('put name here')
# a = 5
# logger.debug(f"a = {a}")
import logging # noqa

__all__ = ['sensitive']


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')
    """
    residual = G.copy() # Copies the Flow network G to use as a skeleton for a residual network, O(|E|+|V|).

    for edge in G.edges:
        # Tightly bounded by the number of edges in G; O(|E|).
        # Loops through all edges in the flow network to fill in the edges of the residual network.
        from_node, to_node = edge
        capacity = G.capacity(from_node, to_node)
        flow = G.flow(from_node, to_node)
        residual.remove_edge(from_node, to_node)
        inverted_edge_weight = flow
        if flow != 0: # If condition met for edge(u,v), adds an edge(v,u) in the residual network with weight = flow.
            residual.add_edge(to_node, from_node, weight=inverted_edge_weight)
        if capacity > flow: # If condition met for edge(u,v), adds an edge(u,v) in the residual network with weight = capacity-flow.
            residual.add_edge(from_node, to_node, weight=capacity - flow)
    # The residual network created.

    # Find the min-cut
    cuts = dict() # cut 's'(True) or 't'(False).
    visited = dict() # If a node is visited.
    for node in G.nodes: # O(|V|), initialize dictionaries.
        cuts[node] = False # All nodes are set as part of 't'-cut as default.
        visited[node] = False
    parent = dict() # initialize parent dictionary
    parent[s] = None

```

```
# Find all nodes part of the 's'-cut (cuts[n] = True.)
stack = [s]
while len(stack) > 0:
   # At most O(|V|)
   # Loops through each node in the network as long as the stack is not empty.
   # Stats at node 's' and thereafter picks nodes from the stack.
   node = stack.pop() # LIFO.
   visited[node] = True # The node is now visited.
   cuts[node] = True # Part of the 's'-cut.
   neighbors = residual.neighbors(node) # Finds neighboring nodes
   for neighbor in neighbors: # adds neighboring nodes to the stack if they /
       are not parent or already visited.
      if parent[node] != neighbor and neighbor != t and not visited[neighbor]:
         parent[neighbor] = node
         stack.append(neighbor) # Adds neighbor to the stack.

sensitive_edges = [] # Initiates a list to keep track of any sensitive edges /
    found
for edge in G.edges:
   # Tightly bounded by the number of edges in G; O(|E|).
   # For each edge, checks whether it is part of the cut or not.
   from_node, to_node = edge
   if cuts[from_node] != cuts[to_node] and G.flow(from_node, to_node) == /
       G.capacity(from_node, to_node):
      # The edge is sensitive and its nodes are appended to the list of /
          sensitive edges
      sensitive_edges.append(edge)

if len(sensitive_edges) == 0: # Returns None,None if there are no sensitive /
    edges.
   return None, None
else:
   return sensitive_edges[0] # Returns the the nodes connected by the first /
       sensitive edge in the list
```
Listing 5: Full code supplied in sensitive.py.

## B The Party Seating Problem

```python
from typing import * # noqa
import unittest # noqa
from src.party_seating_data import data # noqa
# If your solution needs a queue, then you can use this one:
from collections import deque # noqa
# If you need to log information during tests, execution, or both,
# then you can use this library:
# Basic example:
# logger = logging.getLogger('put name here')
# a = 5
# logger.debug(f"a = {a}")
import logging # noqa
from src.graph import Graph

__all__ = ['party']


def party(known: List[List[int]]) -> Tuple[bool, List[int], List[int]]:
   """
   Sig: List[List[int]] -> Tuple[bool, List[int], List[int]]
   Pre:
   Post:
   Ex: party([[1, 2], [0], [0]]) = True, [0], [1, 2]
   """
   l = len(known)
   graph = Graph(is_directed = False)

   has_edge = dict()
   for i in range(l): # O(l)
      has_edge[str(i)] = False #initial state - no edge is added

   for i in range(len(known)): #O(|known|)
      known_party = known[i] #list
      for person in known_party:
         if person > i: #if index is lower, the edge has already been added to /
             the graph
            graph.add_edge(str(i),str(person))
            has_edge[str(i)] = True
            has_edge[str(person)] = True

   #edges initialized
   #color edges, sort in two groups

   colors = ['G','R']
   table = dict() #table 'G' or 'R'
   assert_color = dict() #if a node has an asserted color
   visited = dict() #if a node is visited
   for guest in has_edge: # O(l) initialize dictionaries
      assert_color[guest] = False
      table[guest] = False
      visited[guest] = False

   at_node = 0
   stack = [str(at_node)]
   while at_node < l:
      # at most O(2*l) => O(l).
      # loops from node 0 to the total number of nodes. As long as stack is empty,
      # at_node will increment. If node is taken from stack, it is visited and /
```

```
       will not be
   # run through more than once. Ring structures aren't a problem due to /
       visited dict.
   # worst case is that every node is added to the stack and it loops through /
       all nodes twice => O(l)
   if len(stack) == 0 and visited[str(at_node)]: #run another loop
      at_node += 1 #go to the next node
   else: #else run through loop
      if len(stack) > 0: # there is a guest in the stack
         guest = stack.pop() #LIFO
      else:
         guest = str(at_node)
         at_node += 1 #update node
      if not has_edge[guest]: #if node dosn't have an edge, it's not in the /
          graph
         color = colors[int(guest)%2]
         visited[guest] = True
         table[guest] = color
      else: # if it has an edge, it is part of the graph.
         if assert_color[guest] == False:
            color = colors[int(guest)%2]
         else:
            color = assert_color[guest]

         visited[guest] = True
         table[guest] = color
         neighbors = graph.neighbors(guest) #assert colors on neighbors /
             (=known guests)
         opposite_color = colors[colors.index(color) - 1]
         for neighbor in neighbors:
            if not visited[neighbor]:
               if assert_color[neighbor] == False or assert_color[neighbor] == /
                   opposite_color:
                  assert_color[neighbor] = opposite_color #asserted opposite /
                      color
                  stack.append(neighbor)
               else:
                  #logger.debug(f"Return false. length = {l}\n Table: {table} /
                      \n studied node = {guest}, color = {color}, opposite = /
                      {opposite_color}, neighbor = {neighbor} \n assert color = /
                      {assert_color}\n known= {known}")
                  return False, [], []

tableG = []
tableR = []
for guest in table: #O(l) table length l
   if table[guest] == 'G':
      tableG.append(int(guest))
   else:
      tableR.append(int(guest))

return True, tableG, tableR
```
Listing 6: Full code supplied in party_seating.py.