Instructors:
Michael Ritter
Hugo K. Kasuya Rosado

Exercise sheet #4
Combinatorial Optimization
SuSe 2024

Exercises with an asterisk mark "*" are optional.

1 Breadth-First Search

A *queue* Q is an abstract data structure that contains a sequence of elements. It has access to the following three operations: it can include an element e as the last element in Q in $O(1)$; it can look at the first element of Q in $O(1)$; and it can remove the first element of Q in $O(1)$. Notice that once an element e is included in Q, it can only be removed after every element that was included in Q before e has been removed from Q, this is also known as a *first-in-first-out* (FIFO) data structure.

The Breadth-First Search algorithm (BFS) is a search algorithm that makes use of a queue and is used in many other algorithms as subroutines, such as the Edmonds-Karp algorithm for the MAXIMUM FLOWS PROBLEM. The algorithm receives as an input a graph G, a starting vertex s, and a target vertex t. Starting from s, it iteratively visits adjacent vertices in G until it finds s or asserts that s cannot be reached. More precisely, it starts by labeling s as *visited* and includes it in a queue, then it starts to iterate. At each iteration it looks at the first vertex v in the queue, it labels v as *visited*, and for every *non-visited* vertex u that is adjacent to v, it includes u in the queue. This iterative process ends either when the queue is empty, outputting NULL, or when the target vertex t becomes visited, outputting t. A description is given in Algorithm 1.

Algorithm 1: BFS

input : a graph G, a starting vertex s, and target vertex t
output : target vertex t if s and t are in the same connected component, otherwise NULL

```
1 let Q be a queue
2 label s as visited and include it in Q

3 while Q is not empty do
4     let v be the first element in Q and remove it from Q

5     foreach non-visited u that is adjacent to v do
6         label u as visited and include it in Q

7         if t is visited then
8             return t
```

a. Show that the BFS runs in $O(|V| + |E|)$.
b. Show that one can modify BFS to compute a shortest path (in number of edges) between s and t (argue why it works - no need for a formal proof).
c. Show that one can modify BFS to find a cycle in G or assert that G has no cycles in $O(|V|)$.

2 Executing Maximum Flows

Consider the network (G, s, t, u) on Figure 1.

a. Execute the Ford-Fulkerson algorithm. What is the computed flow and how many augmenting paths were used?
b.* Consider an execution of the Ford-Fulkerson algorithm that chooses an f-augmenting path that increases the flow the least amount at each iteration. Execute one such execution of the Ford-Fulkerson algorithm and count the number of iterations it takes to find a maximum flow.
c. Execute the Edmonds-Karp algorithm. What is the computed flow and how many augmenting paths were used?

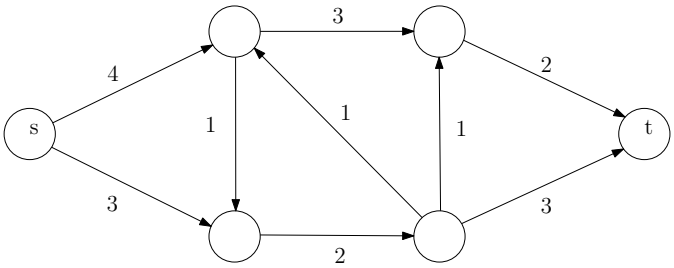

Figure 1: A flow graph with source s and sink t.

d.* Consider an execution of the Edmonds-Karp algorithm that chooses an f-augmenting shortest-path (in number of edges) that increases the flow the least amount at each iteration. Execute one such execution of the Edmonds-Karp algorithm and count the number of iterations it takes to find a maximum flow.

3 Essential backward Edges

The Ford-Fulkerson algorithm maintains a residual network and searches for an augmenting path in this network. An important feature in the residual networks is the creation of the backward edges that allow us to also reduce the amount of flow that is going through a given edge.

a. Show that if no backward edges were included in the residual network, then for any $C \in \mathbb{N}$ there exists an instance where the Ford-Fulkerson algorithm could choose f-augmenting paths that leads it to output a flow with value $\frac{f_{max}}{C}$, where f_{max} is the maximum flow value of the instance.

b.* What happens if we only disallow backward edges into the source?

Hint: Consider a graph similar to a grid graph.

4 Coding Ford-Fulkerson and Edmonds-Karp algorithm

In this exercise, we will write a program to compute the maximum flow of a network using the Ford-Fulkerson and Edmonds-Karp algorithms.

a. Download from Moodle the following files and place them directory/folder: main.py, and graph_module.py.
   (a) file main.py is the file we will be executing in the terminal and it calls functions and methods defined in the other file;
   (b) graph_module.py is the file we will be working on in this exercise. It contains an implementation of `BFS` and `DFS` for finding an augmenting path, as well as an incomplete implementation of Ford-Fulkerson and Edmonds-Karp algorithms.

   With Python 3 installed, to run the code, simply open the terminal, navigate to the folder containing the files, then run the following command to run the code

```
python3 main.py
```

   Some status messages should appear on the terminal and the execution should enter an infinite loop - this is because an essential part of the code is incomplete. To kill the execution, you may press the shortcut `Ctrl + c` on the terminal window.

b. Open graph_module.py. This file contains the definition of the method `update_residual()`, which is the method called by `ford_fulkerson()` to update the current residual graph. The method `update_residual()` has five arguments:
   (a) `source` denotes the source vertex
   (b) `sink` denotes the sink vertex
   (c) `parent_list` represents a path found from the source vertex to the sink vertex. `parent_list` is a list and `parent_list[v]` returns the parent of vertex `v` in the path.
   (d) `gamma` denotes the value of the minimum residual capacity in the path defined by `parent_list`
   (e) `residual_graph` is an adjacency matrix representing the residual matrix. `residual_graph[v][u]` is a non-negative value that represents the amount of capacity on the directed edge (`v`,`u`).

As you may notice, nothing has been implemented in this function and this is the reason the execution of the code enters an infinite loop.

Your task in this exercise is to implement method `update_residual()` - there is no need to optimize the time complexity of your code. This function should update the `residual_graph` accordingly, i.e., it should augment the residual graph along the path defined by `residual_graph` with the value `gamma` accordingly.

In file `main.py` the input network is defined on variable `adjacency_matrix` on line 6. You may modify this matrix to aid you in this task. Also, on line 18, there is a piece of code to generate larger random networks, uncomment this line to override the manual definition from line 6. You can modify the parameters `num_vertices`, `lower_cap`, `upper_cap`, to change how the random graph is built - details on this are written in the comments of the code.

c.* Look at the code that defines `ford_fulkerson()` and take a few minutes to understand it. Notice that this method calls both `find_aug_path_DFS()` and `update_residual()`.

Now notice that the method `edmonds_karp()` is incomplete. Your task in this exercise is to implement `edmonds_karp()`. Notice that the file `graph_module` already has an `find_aug_path_DFS()`, and this should help you.

d.* Uncomment line 18 to generate random graphs, and modify the parameters `num_vertices`, `lower_cap`, `upper_cap` to see how they might affect the running time of the implementation of the two algorithms.