CSC190: Computer Algorithms and Data Structures
Lab 8
Assigned: Mar 30, 2015; Due: April 6, 2015 @ 10:00 a.m.

1 Objectives

This is the final lab for this course! In this lab, you will implement the breadth-first path search function that can be used to find an augmenting path in a flow network. The function that you will be implementing in this lab can be used to fulfill the requirements of Part 2 of Assignment 3. Please refer to Assignment 3's instruction manual for more information on flow networks.

You will download content in the `Lab8` folder which contains two sub-folders (`code` and `expOutput`) into your ECF workspace. Folder `code` contains a skeleton of function implementations and declarations. Your task is to expand these functions appropriately in `lab8.c` as required for implementation. `main.c` evokes all the functions you will have implemented and is similar to the file that will be used to test your implementations. Use `main.c` file to test all your implementations. Folder `expOutput` contains outputs expected for the supplied `main.c` file. Note that we will **NOT** use the same `main.c` file for grading your lab. Do **NOT** change the names of these files or functions.

2 Grading

It is **IMPORTANT** that you follow all instructions provided in this lab very closely. Otherwise, you will lose a *significant* amount of marks as this lab is auto-marked and relies heavily on you accurately following the provided instructions. Following is the mark composition for this lab (total of 20 points):

- Successful compilation of all program files with no memory leaks i.e. the following command results in no errors (2 points):
  ```
  gcc main.c adjMatrix.c linkedQueue.c lab8.c -o run
  valgrind --quiet --leak-check=full --track-origins=yes ./run
  ```
- Output exactly matches expected output (10 points)
- Code content (8 points)

Sample expected outputs are provided in folder `expOutput`. We will test your program with a set of completely different data files. Late submissions will **NOT** be accepted.

Breadth-first Path Searching Algorithm

Since this lab is a continuation of Lab 7, you should copy over your `adjMatrix.c` file into your workspace. Also copy your implementation of the functions `struct flowNetwork * initFlowNetwork()` and `void deleteFlowNetwork(struct flowNetwork * fN)` from `lab7.c` into `lab8.c`.

The main task in this lab is the implementation of the function `int breadthFirstPathSearch(struct flowNetwork * fN, int s, int d)` in `lab8.c` file. This function uses breadth-first search to find an augmenting path connecting vertices `s` and `d` to which flow can be added without infringing on capacity constraints of the network. Since this is a breadth-first search algorithm, you will have to use a queue in your implementation. You will be provided with the queue interface functions in the `linkedQueue.c` file:

- `void initQueue(struct Queue ** qPtr);`
- `int isQueueFull(struct Queue * Q);`
- `int isQueueEmpty(struct Queue * Q);`
- `void enqueue(struct Queue * Q, struct Data d);`
- `void dequeue(struct Queue * Q, struct Data * d);`

struct Data will be enqueued and dequeued from the queue. This structure keeps track of the vertex label of the node in its vertex member.

In order to understand the general breadth-first path-finding algorithm, consider Figures 1, 2, 3. Figure 1 illustrates the initial flow network which has no flow yet on any edges. The path-finding algorithm should find a path connecting s and d which are source and destination nodes that consists of no edges with flow being equal to the maximum edge capacity. The breadth-first search algorithm attempts to find such a path that has the shortest distance (i.e. smallest number of edges) from s to d. A breadth-first search algorithm will start from s and examine all children/adjacent nodes of s (by enqueueing these nodes into a queue) one by one. For the network in Figure 1, node 1 will be examined first and checked to see if it has been visited or not and whether the current flow in the edge (v_0, v_1) has not reached maximum capacity. If these conditions are met, node 1 is set to be visited and in the index corresponding to node 1 in the parent array is set to node 0 as node 0 is the parent of node 1 forming the edge (v_0, v_1) and then node 1 is enqueued into the queue. All the children of node 0 are examined in a similar manner and enqueued into the queue if the checking conditions are met.

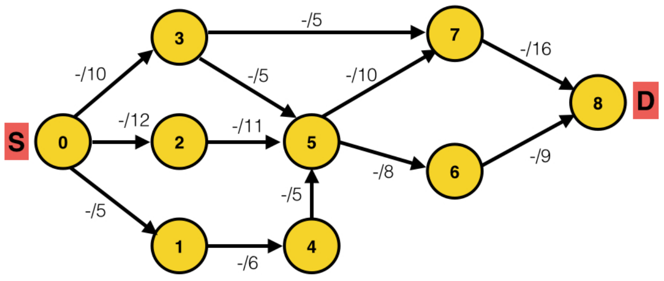

Figure 1: Initial network with no flow

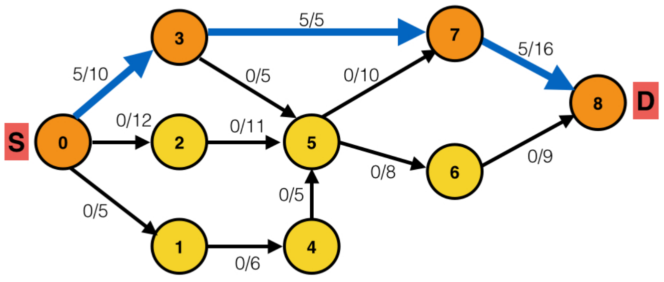

Figure 2: First Augmenting Path from S-D

After examining all adjacent nodes of node 0, parent={-1, 0, 0, 0, -1, -1, -1, -1, -1}. The next node in the queue will be node 1 and its child node 4 is examined. As 4 is unvisited and has no flow, the parent of node 4 is set to node 1, marked as visited and enqueued into the queue. Proceeding in this manner, parent of 5 is set to 2 and node 5 is enqueued into the queue. The next node examined is 3 and its children are 5 and 7. Since 5 has already been visited, its parent will not change and it will not be enqueued into the queue. However, since 7 is not visited, its parent is set to 3, marked as visited and enqueued into the queue. This will proceed until all nodes in the graph are examined (i.e. the queue is empty). At this point, the parent array for the flow network in Figure 1 will be parent={-1, 0, 0, 0, 1, 2, 5, 3, 7}. Starting from the last index corresponding to node 8 which is the destination node, it is clear that the parent of 8 is 7, the parent of 7 is 3, the parent of 3 is 0. Hence, the augmenting path from s to d in which flow can be increased is 0-3-7-8 and this is the shortest available path from node 0 to 8 as illustrated in Figure 2.

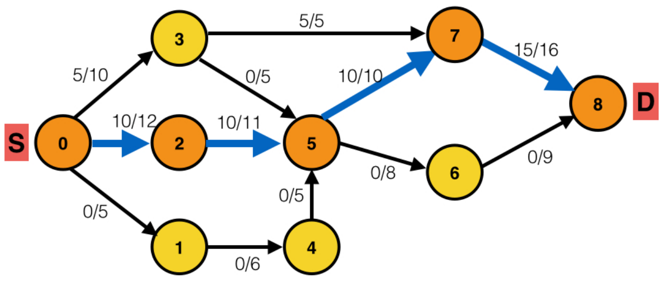

Figure 3: Second Augmenting Path from S-D

Suppose that a flow of 5 has been added to the path 0-3-7-8. If the breadth-first search algorithm is called again to operate on this flow network, the parent of 7 will not be set to be 3 anymore as the flow on the edge (v_3, v_7) is 5 which is the maximum capacity of that edge (no more flow can be added without infringing on the capacity constraint of that edge). The parent of 7 in this case will be set to 5. The augmented path resulting from this graph will then be 0-2-5-7-8 as illustrated in Figure 3. Hence, the path-finding algorithm will successfully avoid the edge that is full. At the conclusion of the path search, the function will return 1 if the destination node has been visited and 0 otherwise. If the return value is 0, then an augmenting path does not exist that connects s to d. This breadth-first searching algorithm is called the **Edmonds-Karp** algorithm. The implementation of this algorithm is summarized in the following:

- First set all elements in the `visitedNodes` array to 0 (to mark all nodes as unvisited)
- Initialize a queue and enqueue the starting node
- Begin a while-loop in which:
  - The queue is dequeued (say the dequeued node is v_i)
  - For every node v_j that is adjacent to the dequeued node v_i, check if this adjacent node is unvisited and whether the current flow in the edge (v_i, v_j) is such that more than 0 flow added to the edge (i.e. $c_{i,j} - f_{i,j} > 0$)
  - If these conditions are met, then set the parent of this adjacent node v_j to be the dequeued node v_i and enqueue the adjacent node into the queue
  - Repeat the above until the queue is empty
  - Before the function terminates return 1 if d is visited and 0 otherwise

This lab will be tested via the following commands:

- `./run`
- `valgrind --quiet --leak-check=full --track-origins=yes ./run`

Outputs from these tests must match the content of `lab8.txt` which is the result of the parameters passed to function calls in `main.c`.

3 Code Submission

You can submit via git or the `submitcsc190s` command on your ECF machine (no bonus points for submissions either way). Ensure that you submit through only one venue.

3.1 Submission through `Git`

Once you have completed this lab, you will submit your work by:

- Log onto your ECF account
- Browse into the directory you had cloned in Lab 0 (i.e. `cd ~/UTORID/`)
- Create a folder named `Lab8` (i.e. `mkdir Lab8`) in that cloned directory
- Ensure that your code compiles in the ECF environment
- Copy all your completed code (`adjMatrix.c` and `lab8.c`) into the Lab8 folder
- Browse into the `~/UTORID/` directory
- Add all files in the Lab8 folder (i.e. `git add *`)
- Commit all files that have been modified in the Lab 8 folder (i.e. `git commit -m "adding lab files"`)
- Push all changes committed to the `git` server (i.e. `git push origin master`)

3.2 Submission through `submitcsc190s`

- Log onto your ECF account
- Ensure that your completed code compiles
- Browse into the directory containing your completed code (`adjMatrix.c` and `lab8.c`)
- Submit by issuing the command:
  `submitcsc190s 11 adjMatrix.c lab8.c`

3.3 Checklist

ENSURE that your work satisfies the following checklist:

- You submit before the deadline
- All files and functions retain the same original names
- Your code compiles without error in the ECF environment (if it does not compile then your maximum grade will be 3/20)
- Do not resubmit any files in Lab 8 after the deadline (otherwise we will consider your work to be a late submission)