CSC190: Computer Algorithms and Data Structures
Assignment 3
Assigned: Mar 19, 2015; Due: Apr 10, 2015 @ 10:00 a.m.

1 Objectives

In this assignment, you will implement an algorithm that computes the maximum flow possible in a flow network digraph. A flow network can represent a network of pipelines having different flow rates. Please refer to the diagram in Figure 1 for an example of a flow network. In a flow network, there are two special nodes called the **source** and **destination** nodes (marked as `S` and `D` in Figure 1). Material such as oil fluids can propagate through the network starting from the source `S` and flow into the destination node `D`. Intermediate nodes (i.e. 1 to 7 in the example) can represent pumping stations.

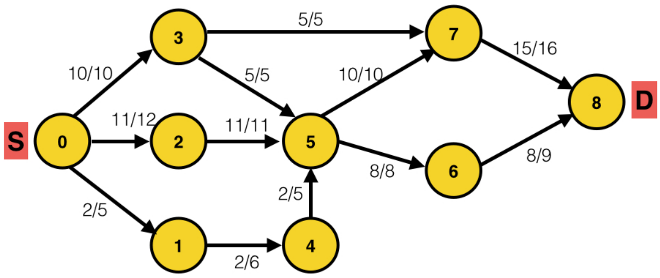

Figure 1: A sample flow network

1.1 Constraints

Since a flow network is a special kind of network subject to physical limitations, flow and capacity constraints do apply. Every edge $e_{i,j}$ or pipeline connected by vertices v_i and v_j has a maximum capacity of $c_{i,j}$. Hence, due to this physical **capacity limitation**, any substance flowing through pipeline $e_{i,j}$ must have a flow of $f_{i,j}$ that is between 0 and $c_{i,j}$ (i.e. $0 \leq f_{i,j} \leq c_{i,j}$). Referring back to Figure 1, $e_{0,2}$ has a flow of $f_{0,2} = 11$ and a capacity of 12. Since $f_{0,2} \leq c_{0,2}$, this is a valid flow.

Next, suppose three pipelines or edges meet at a single node and two pipelines leave that node (consider $e_{2,5}$, $e_{3,5}$, $e_{4,5}$, $e_{5,6}$, $e_{5,7}$). Any substance flowing into a node must be equal to the substance flowing out of the node and this is called **flow conservation**. In Figure 1, $f_{2,5} + f_{3,5} + f_{4,5} = 18$ and $f_{5,6} + f_{5,7} = 18$ and more generally this can be expressed as $\sum_{i \in V} f_{i,j} = \sum_{i \in V} f_{j,i}$ for all v_j with the exception of S and D as these are source and sink nodes (generate or consume substance).

Finally, there is the notion of **back flow**. For instance, if there exists positive flow $f_{i,j}$ on edge $e_{i,j}$, then it is possible for substance to flow in the opposite direction $e_{j,i}$ where $0 \leq f_{j,i} \leq f_{i,j}$. If there is no substance in the pipeline then nothing can propagate in the opposite direction. However, if there is some material flowing through the pipe in one direction, then it is entirely possible for this material to flow in the opposite direction.

1.2 General Algorithm for Finding Maximum Flow in the Network

If you attempt to increase the flows on edges of the flow network provided in Figure 1, you will notice that this is not possible. Suppose you are increasing flow on $e_{0,2}$ by 1 to 12. In order to satisfy flow conservation, the flow on $e_{2,5}$ must also increase by 1. This is impossible as this will violate the maximum capacity restriction on $e_{2,5}$ (i.e. maximum flow possible through this edge is 11 not 12). Hence, if you attempt to increase the flow like this on every edge, you will realize that this is not feasible. For this reason, all flows listed in Figure

1 are the maximum possible flows throughout the flow network. The amount of substance leaving from node 0 which is the source node is 23 and similarly the amount of substance entering the destination node 8 is also 23. For this reason, the maximum flow throughout this flow network is 23! In this assignment, you will implement a well known algorithm to compute the maximum flow throughout a flow network. In general, this algorithm is composed of the following steps:

- Find a path p from `S` to `D` that consists of no edges with flow that is equal to the full capacity of the edge (i.e. $f_{i,j} \neq c_{i,j}$)
- Find the maximum flow that can be added to the path so that none of the edges in the path violate their flow capacity constraints
- Add this flow to every edge in the path p
- Repeat the above until no more paths exist in the graph from `S` to `D` to which more flow can be added

This assignment is divided into three parts. In the first part, you will implement interface functions for the representing the flow network graph. In the second part, you will implement a **breadth-first** search algorithm to discover a path from S to D to which more flow can be added to all edges in this path. These paths will be referred to as **augmenting** paths. In the third part, you will implement code that calls on this path discovering algorithm repeatedly to add flow to the flow network until no more flow can be added.

2 Grading: Final Mark Composition

It is **IMPORTANT** that you follow all instructions provided in this assignment very closely. Otherwise, you will lose a *significant* amount of marks as this assignment is auto-marked and relies heavily on you accurately following the provided instructions. Following is the mark composition for this assignment (total of 40 points):

- Successful compilation of all program files i.e. the following command results in no errors (3 points):
  `gcc assignment3.c linkedQueue.c adjMatrix.c main.c -o run`
- Successful execution of each one of the following commands: (1 point)
  ```
  valgrind --quiet --leak-check=full --track-origins=yes ./run 1
  valgrind --quiet --leak-check=full --track-origins=yes ./run 2
  valgrind --quiet --leak-check=full --track-origins=yes ./run 3
  ```
- Output from Part 1 exactly matches expected output (7 points)
- Output from Part 2 exactly matches expected output (8 points)
- Output from Part 3 exactly matches expected output (9 points)
- Code content (10 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. All late submissions will be assigned a grade of 0/40.

Part 1

First, you will implement interface functions in the `adjMatrix.c` and `assignment3.c` files that can be used to perform operations on a flow network. For this assignment, you will use an **adjacency matrix** representation of the graph. One structure definition provided in the `assignment3.h` file that is to be used for the adjacency matrix definition is `struct Edge`. This structure has the following members:

- `int flow;`
- `int flowCap;`

`struct Edge` represents an edge and stores the current flow in that edge and the maximum capacity of that edge in the `flow` and `flowCap` members respectively. The adjacency matrix is an n by n matrix in which each element is of type `struct Edge`. The i, j entry of the adjacency matrix represents the edge connecting v_i and v_j. Flow on this edge is $f_{i,j}$ and the capacity of this edge is $c_{i,j}$. These metrics are stored in the `flow` and `flowCap` members. There are three interface functions you will implement which will initialize, insert elements and delete the adjacency matrix:

- `struct Edge ** initAdjMatrix();`
  - In this function, a 2-D matrix of type `struct Edge **` should be dynamically created
  - Since every matrix entry is of type `struct Edge`, the `flow` and `flowCap` members of each entry should be initialized to 0
  - The double pointer pointing to the dynamically allocated matrix should be returned by the function
- `void insertAdjMatrix(struct Edge ** aM, int vi, int vj, int flow, int flowCap);`
  - Suppose the flow network contains an edge (v_i, v_j) with a flow of `flow` and edge capacity of `flowCap`
  - This function should access the entry `aM[i][j]` corresponding to the edge (v_i, v_j) and set the members of this entry in accordance to the parameters `flow` and `flowCap` passed to this function
- `void printAdjMat(struct Edge ** aM);`
  - This function prints the flow and capacity of an edge if that edge has a capacity of greater than 0 (please refer to the files in the `expOutput` folder for details on formatting)
- `void deleteAdjMatrix(struct Edge ** aM);`
  - In this function the dynamically allocated adjacency matrix `aM` should be freed

Next, a set of functions that initialize and perform operations associated with a flow network are implemented next:

- `struct flowNetwork * initFlowNetwork();`
  - This function will dynamically allocate a `struct flowNetwork` variable and initialize its members `adjMatrix`, `visitedNodes` and `parent`
  - `adjMatrix` is the graph representation of the flow network and is to be initialized using the `initAdjMatrix` function
  - `visitedNodes` is an array that keeps track of vertices that are visited in the graph and all elements are to be initialized to 0
  - `parent` is an array that is to be used by the path-finding algorithm and all elements are to be initialized to -1
- `void deleteFlowNetwork(struct flowNetwork * fN);`
  - All dynamically allocated members in `fN` are to be freed in this function

This part of the assignment will be tested via the following commands:

- `./run 1`
- `valgrind --quiet --leak-check=full --track-origins=yes ./run 1`

Outputs from these tests must match the content of `Part1.txt` which is the result of the parameters passed from function call `p1()` in `main.c`.

Part 2

In this part, you will implement the function `int breadthFirstPathSearch(struct flowNetwork * fN, int s, int d)` in the `assignment3.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 2, 3, 4. Figure 2 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 2, 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.

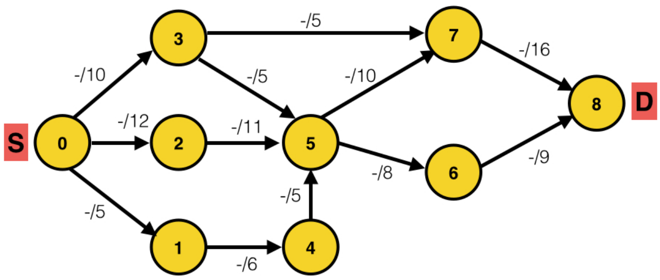

Figure 2: Initial network with no flow

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 2 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

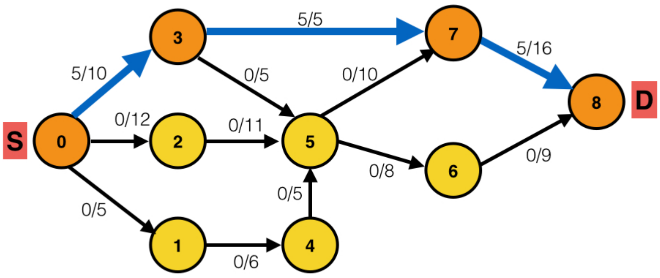

Figure 3: First Augmenting Path from S-D

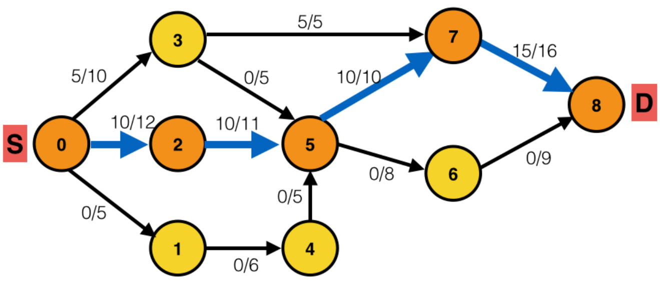

Figure 4: Second Augmenting Path from S-D

`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 3.

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 4. 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 part of the assignment will be tested via the following commands:

- `./run 2`
- `valgrind --quiet --leak-check=full --track-origins=yes ./run 2`

Outputs from these tests must match the content of `P2.txt` which is the result of the parameters passed from the function call `p2()` in `main.c`.

Part 3

Finally, in this part, you will implement the function `void maximizeFlowNetwork(struct flowNetwork * fN, int s, int t)`. This is the function that computes the maximum flow in the flow network `fN`. Following are the three main components of this function:

- The `breadthFirstPathSearch` function is called to find an augmented path
- If such a path exists then the return value of this function is 1.(i.e. if node `d` is visited by the path searching algorithm). In this case, the maximum possible flow that can be added on all edges that form the augmenting path is computed (this flow must heed the capacity constraints of the edge)
- The flow on all edges forming the augmenting path is incremented to the value computed in the previous step. For every edge in which flow is added, ensure that you account for the back flow in the opposite direction
- All three steps above are repeated until no more flow can be added to the flow network

For instance, suppose that the augmented path computed by the path searching algorithm is 0-2-5-7-8. From Figure 3, it is clear that the maximum flow that can be added/augmented to all edges in this path is 10 due to edge (v_5, v_7) resulting in the flow network illustrated in Figure 4. This algorithm is called the **Ford-Fulkerson** algorithm. This part of the assignment will be tested via the following commands:

- `./run 3`
- `valgrind --quiet --leak-check=full --track-origins=yes ./run 3`

Outputs from these tests must match the content of `P3.txt` which is the result of the parameters passed from the function call `p3()` in `main.c`.

3 Materials Provided

You will download contents in the folder `Assignment3` which contains two sub-folders (`code` and `expOutput`) onto your ECF workspace. Folder `code` contains a skeleton of function implementations and declarations. Your task is to expand these functions appropriately in `assignment3.c` and `adjMatrix.c` files. `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 function calls in `main.c`. Note that we will **NOT** use function calls with the same parameters for grading your assignment. Do **NOT** change the name of these files or functions.

4 Code Submission

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

4.1 Submission through `Git`

Once you have completed this assignment, 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 `Assignment3` (i.e. `mkdir Assignment3`) in that cloned directory
- Ensure that your code compiles in the ECF environment
- Copy all your completed code (`adjMatrix.c` and `assignment3.c` into the `Assignment3` folder
- Browse into the `~/UTORID/` directory
- Add all files in the Assignment3 folder (i.e. `git add *`)
- Commit all files that have been modified in the `Assignment3` folder (i.e. `git commit -m "adding assignment files"`)
- Push all changes committed to the `git` server (i.e. `git push origin master`)

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 3/40)
- Do not resubmit any files in `Assignment3` after the deadline (otherwise we will consider your work to be a late submission)

4.2 Submission through `submitcsc190s`

- Log onto your ECF account
- Ensure that your completed code compiles
- Browse into the directory that contains your completed code (`assignment3.c`, `adjMatrix.c`)
- Submit by issuing the command:
  `submitcsc190s 9 assignment3.c adjMatrix.c`

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 3/40)
- Do not resubmit any files in `Assignment3` after the deadline (otherwise we will consider your work to be a late submission)