1 Introduction

In this assignment, you will be implementing a solution to the max flow problem on undirected graphs. The general class of max flow problems is discussed in some detail in the textbook, section 9.4. However, your program will solve a slightly different version of the problem than the one in the book. Specifically, your program will work entirely with undirected graphs, and will not use the residual graph, as shown in Figures 9.40–9.46 of the text.

Consider a graph, $G$, with $|V|$ vertices in $V = \{1, 2, 3, \ldots\}$ and $|E|$ edges in $E = \{a, b, c, \ldots\}$. Assume that each edge, $e \in E$, has a non-negative “weight”, $\text{cap}(e)$, representing the maximum rate at which flow can travel through that edge in either direction, i.e., $\text{cap}(e)$ represents the “size of the pipe” from which edge $e$ was constructed. Also assume that a specific pair of vertices has been selected to act as the source (starting point) and sink (ending point) for the flow. The goal is to maximize the total flow rate leaving source (equivalently, entering sink), while maintaining flow balance at every other vertex, i.e., the total flow rate entering and leaving every other vertex must be exactly balanced.

Your task for Part 1 is to create a program that

(i) reads a graph $G$ from an input file, together with specified source and sink vertices, and

(ii) outputs the flow rate obtained through the best single flow-augmenting path from source to destination, which supports the highest flow rate among all possible paths from source to sink, followed by the ordered list of vertex names which form that path.

Your task for Part 2 is to add the following features, so your program solves the full max-flow problem:

(iii) it finds a series of flow-augmenting paths, one at a time, which jointly carry the maximum-possible total flow rate from source to sink;

(iv) it outputs the maximum total flow rate that can be sent from source to sink;

(v) it outputs the set of vertices belonging to the minimum cut (i.e., they can all still be reached from source by a flow-augmenting path, even though sink and any other vertices outside the minimum cut are no longer reachable), which shows that no further flow-augmenting paths are possible; and

(vi) it decomposes the final flow pattern into a set of individual path-flows from source to sink, where each path-flow carries a constant, positive flow rate through every edge along the path.

\[1\text{Note that each flow-augmenting path found in part (iii) does not necessarily represent a valid individual path-flow because a flow-augmenting path can cross edges where they reduce the rate of flow traveling the wrong direction.}\]
Part 1: Finding the best flow-augmenting path

To illustrate the method, I have prepared an example graph with 7 vertices and 9 edges, where source = 1, sink = 7 and each edge is labeled with its capacity in parentheses:

In this example, I use colors to indicate the status of each vertex: red for the active vertex at the current step, green for discovered vertices that are currently in the priority queue (and the little italic number is their weight), white for undiscovered vertices, and black for completed vertices that have already finished their active processing during some previous algorithmic step.

The algorithm you will be using for finding the best flow-augmenting path is a variation of the “classic” Dijkstra weighted shortest-path algorithm, described in section 9.3.2 of the textbook. The differences from the “classic” Dijkstra algorithm are:

- Let \( v \) be the active vertex with weight \( w(v) \), and let \( u \) be a not-yet completed vertex adjacent to \( v \) through edge \( e \) with weight \( \text{cap}(e) \). Then \( w(u) = \min\{w(v), \text{cap}(e)\} \) for our problem, versus \( w(u) = w(v) + \text{cap}(e) \) for the “classic” Dijkstra algorithm.
- At each step, our algorithm chooses the next active vertex as \( v^* \) for which \( w(v^*) \geq w(u) \) among all not-yet completed vertices \( u \) (in this case, indicating that the path from source to \( v^* \) has the highest capacity among all not-yet completed vertices) instead of \( w(v^*) \leq w(u) \) (in the original case, indicating the length of the path from source to vertex \( v^* \) is the shortest among all not-yet completed vertices).
- The selection of the next active vertex for each step is still handled by a priority queue, but it is organized to give the largest (not smallest) element in the queue when you ask for the next item.

Thus, at the first step (vertex 1 is active) we discover vertices 2, 3, 4 and add them to the priority queue; vertex 4 (weight 7 shown in bold) ends up as the highest-priority element. At the second step, vertex 1 is completed, vertex 4 is active and we discover vertex 6 and add it to the priority queue, where it becomes the new highest-priority element. At the third step, vertices 1 and 4 are completed, vertex 6 is active and we discover another copy of vertex 2 (notice it now has two weights) along with the sink, vertex 7, and add both of them into the priority queue. However, since vertex 7 is not the new highest-priority element, we are not yet finished.

Notice that in the fifth step, when vertex 3 becomes active because of its weight 5 entry, the priority queue still holds the other copy of vertex 3 with weight 2. In this example, the algorithm terminates before this redundant/lower-valued copy of vertex 3 reaches the top of the priority queue so we do not need to worry about it. (Indeed, the priority queue has three leftover redundant entries for vertices 3, 5, and 7 when it terminates after completing the higher-weighted entry for vertex 7.) However, if one of these redundant entries did reach the top of the priority queue while the algorithm was still running, we would simply discard it, without making active again, because of its completed status: its redundant weight is lower than its primary weight so repeating the active processing step cannot possibly increase the flow on any path continuing downstream from this vertex.

Thus, to complete part 1 of the assignment, your program must now identify a single highest-flow path from source to sink, and print out its vertices in order from source to sink, which in this example would be (1, 4, 6, 3, 5, 7).

Part 2: Complete Solution to the Max Flow Problem

In order to create a complete solution to the max-flow problem, your program must make repeated calls to the algorithm for finding the best flow-augmenting path. However, things get more interesting in these later iterations because of the presence of existing flows on the edges, which were generated in previous iterations.

The textbook describes a method for handling subsequent iterations of the flow-augmenting path algorithm in which it operates on a newly-created “residual graph” where undirected edges are split into pairs of directed edges, and edge capacities are reduced (and edges possibly even eliminated) by subtracting the current flow from their base capacity.
Instead of working with these residual graphs, I want your program to turn the set of edges into a full
object-oriented class, which supports public methods that can called by the flow-augmenting path algorithm
to:

- ask by how much can I increase the flow rate through edge e in the direction leaving vertex v.
- change the flow rate through edge e by adding Δf more units in the direction leaving vertex v.

Internally, the class knows the two endpoints and capacity for each edge. Since edges are undirected, an edge
can support any flow rate up to cap(e) in either direction, so for concreteness lets call the flow direction
from its lower-numbered vertex towards its higher numbered vertex positive, and say that the current flow
can be any value between −cap(e) and +cap(e).

If vertex v calls the first method for edge e, we determine the sign (positive or negative) for flow leaving
vertex v through edge e based on the number of the vertex at the opposite end of the edge. We also look at
the sign and magnitude for flow(e), the current amount of flow on edge e. If the signs are the same, then
return cap(e) − abs(flow(e)). If the signs are opposite, return cap(e) + abs(flow(e)).

If vertex v calls the second method for edge e, we determine the sign (positive or negative) for flow leaving
vertex v through edge e. Then we add (sign * Δf) to flow(e).

Returning to the illustrative example, consider the second iteration of the algorithm for finding a maxi-

mum flow-augmenting path, which begins with slide 8. Notice that each edge label is a pair of values of the
form (cap(·), flow(·)). In cases except edge f, the flow component has a positive sign because it is traveling
from a lower-numbered to a higher-numbered vertex.

The interesting part occurs on slide 10, when vertex 5 is active. Notice that edge e (connecting vertices
3 and 5) currently has flow(e) = +5, but vertex 5 is interested in pushing flow in the negative direction to
extend its flow-augmenting path to vertex 3. Thus, the available capacity reported would be 12 (i.e., the
difference between the current flow rate of +5 and the highest possible flow in the direction leaving from
vertex 5, which is −7), so that vertex 3 is [re]labeled with the minimum of 4 (weight of vertex 5) and 12
(available edge capacity).

At the end of the second iteration shown in slide 14, we update the flows on every edge by increasing the
flow rate by 3 units along this second flow-augmenting path (1, 2, 5, 3, 6, 7). Thus, the total flow rate from
source to sink generated by both iterations is now 8 units.

In order to terminate the algorithm we need to run a third iteration of the algorithm (shown in slides
15–21) in which the priority queue runs out of new vertices to try before we reach sink 7. In this case, we
recognize the minimum cut to be the set of completed vertices (i.e., vertices 1 – 6).

4 Decomposing the final flow pattern into individual path-flows

Describing the final flow pattern in this example as the combination of 5 units sent through flow-augmenting
path (1, 4, 6, 3, 5, 7) and 3 units sent through flow-augmenting path (1, 2, 5, 3, 6, 7) is both confusing and
non-physical, because the second flow-augmenting path sends 3 units of negative flow through edge e from
5 to 3 and again through edge f from 3 to 6. Indeed, what is really happening on edge e is that 2 units of
flow are traveling in the positive direction, rather than having 5 purple units traveling the positive direction
along with 3 orange units traveling in the negative direction. A similar situation applies to edge f. In other
words, the way in which we “move” 3 units of orange flow from node 5 to node 6 is by leaving it behind
at node 5 and calling it purple — which we balance by leaving behind 3 units of purple flow at node 6 and
calling it orange.

Individual path-flows show how the final flow pattern could be constructed by transporting some tangible
commodity through the network, without resorting to “tricks” like cancelling out negative flows. If you think
of edges as pipes with different diameters, then the collection of path-flows is like representing the total flow
pattern by a set ropes of different thicknesses, each stretching from source to sink along its own path through
the network of pipes.

Jumping ahead to slide 35, what we want is a method for describing the final flow pattern which consists
of three separate and non-interfering path-flows (i.e., a 3-unit purple rope, a 3-unit orange rope, and a 2-unit
blue rope), traveling independently from source to sink along its stated path. In other words, I could place
a physical object into the orange stream as it leaves node 1 and then watch travel through edge \( a \) to node 2, then through edge \( d \) to node 5, and finally through edge \( h \) to node 7.

The same algorithm for solving the max flow problem can also be used for finding the set of individual path-flows. The only change required is to replace the original edge capacities by the final flow values, as shown in slide 22. Notice that \( \text{cap}(b) = 0 \) so edge \( b \) effectively disappears and hence vertex 1 is not able to discover vertex 3 in step 1. The capacities of all the remaining edges also decrease, except for edges \( h \) and \( i \) which form the minimum cut.

When the sink (vertex 7) becomes active at slide 25, we identify \((1, 4, 6, 7)\) carrying 3 units as the first (purple) path-flow. All edge capacities in the graph are now updated, by subtracting the purple flow from the current total, before continuing with the next iteration.

The second iteration is shown from slide 26 to slide 29, where we identify \((1, 2, 5, 7)\) carrying 3 units as the second (orange) path-flow. Afterwards, all edge capacities in the graph must be further updated, so that \( \text{cap}(a) = \text{cap}(b) = \text{cap}(d) = \text{cap}(i) = 0 \) making all of these edges disappear. Note also that \( \text{cap}(e) = \text{cap}(g) = \text{cap}(h) = 2 \) for each of the remaining edges.

During the third iteration of the algorithm (which includes slides 30 – 35), we see how the remaining edges with non-zero capacities fit together perfectly, giving us \((1, 4, 6, 3, 5, 7)\) carrying 2 units as the third and final (blue) path-flow.

5 Program Input and Output

Your program should read the name of an input file from the command line. The input file has the following format.

Line 1: two integers, representing the number of vertices, \(|V|\), and number of edges, \(|E|\), in \( G \).

The next \(|V|\) lines, one line per vertex: the name of this vertex as a character string without whitespace or surrounding delimiters, such as: Los Angeles or 7. On one line the vertex name will be followed by the string source and on another line the vertex name will by followed by the string sink.

The next \(|E|\) lines, one line per edge: the name of this edge as a character string without whitespace or surrounding delimiters, a single blank, a number representing the edge capacity, a single blank, the name of one connected vertex as a character string without whitespace or surrounding delimiters, a single blank, the name of the other connected vertex as a character string without whitespace or surrounding delimiters.

During execution, the program generates one similarly-named output file with the original filename extension (if any) changed to “.out”. For example, if the input file were flow_ex.txt, then the output file would be flow_ex.out.

The output file is a normal text file containing the following information.

Line 1: the string Max flow obtained: followed by a single blank, then a number representing the maximum flow for this test case as determined by your program.

Line 2: the string Vertices in min cut: followed by a single blank, then a list of vertex names separated by blanks.

Remaining lines: one individual path-flow per line, consisting of the string Flow rate: followed a single blank, then a number representing the flow rate achieved by this path-flow, then another blank, then the string Path: followed by a list of vertex names separated by blanks.