Implentation report for Algorithmics 4

Tom Wallis, 2025138

February 25, 2016

An implementation report for Tom Wallis' Algorithmics 4 assessed exercise.

Part A, Task 1

THE FIRST PART of the exercise involved creating the isFlow(), getValue() and printFlow() functions.

isFlow() navigated the graph edge by edge. It traversed the graph, keeping track of edges it had seen and edges it was yet to visit. As it traversed the graph, it kept track of the flow going into and out of each vertex and confirmed that the capacity constraint was met (0<f<c). After traversing the entirety of the graph, the flows into and out of the vertices should have summed to 0, so we check the difference as calculated during traversal. If any failures arise, we return false. If no failures have arisen for the flow conservation constraint or the capacity constraint, the flow must be valid.

Implementing isFlow() for an arbitrary network

getValue() was simpler, as a valid flow's value is the sum of flows from the source of the network. This was simply collected from the source's adjacency matrix, summed, and returned.

Implementing getValue()

printFlow() collects information about each edge it sees as it traverses the graph as in isFlow(). Every edge gets information formatted and added to a StringBuilder which collects the output and prints it to the console.

Implementing printFlow() for an ordinary network

Part A, Task 2

THE SECOND HALF OF the first part concerned itself with implementing the FordFulkerson algorithm for an arbitrary network. To do this, the constructor for the residual graph needed to be implemented as well as functions findAugmentingPath() and fordFulkerson().

The constructor was the simplest part, as it could borrow mostly from the superclass' constructor. Some instance variables were set to make certain that instances of the class had everything they needed.

Beause the residual graph was created as a part of finding the augmenting path, the constructor remained almost unchanged

findAugmentingPath() traversed the network provided in the constructor and created a residual graph from that network. For every edge, if flow<capacity, then there should exist a forward edge in the residual graph with the capacity of the slack of the edge. If the edge's flow is >0, there should exist a backwards edge in the residual graph with capcity equal to the flow of the original edge. Then, we perform a breadth first search on the residual graph created to find the shortest path from its source to its terminal. Upon reaching the terminal we stop traversing the residual graph and return the winning path as a LinkedList.

findAugmentingPath() finds the residual graph from a network and uses a breadth first search to return the smallest possible augmenting path if one exists.

fordFulkerson() iterated over a network and attempted to find an augmenting path from the residual graph it creates. When the residual graph failed to return another augmenting path, we know we have found the max flow and can exit. If there exists an augment-

fordFulkerson() creates augmenting paths and, while there are still augmenting paths to be found, alters the network to reflect the improved flow that could be found. by the Augmenting Path Theorum, once there are no more augmenting paths the flow has been maximised.

ing path, we find the minimum of the capacities of the edges in the augmenting path, which we might here call 'm'. We then walk the augmenting path and alter the original network based on the edges we visit. For every edge in the augmenting path, if it exists in the original network, it must be a forward edge. In this case, so long as adding 'm' to the flow of the edge doesn't violate the capacity constraint (f+m<=c), we update the flow of the original edge to be f=f+m. If the edge does *not* exist in the original network, it must be a backwards edge, so we set the backwards edge's flow to be f=f-m. After iterating until no further augmenting paths remain, by the Augmenting Path Theorum the resulting network has maximal flow.

Part B

AFTER IMPLEMENTING FORD-FULKERSON CORRECTLY parts of the system had to be rewritten to take into account modelling a set of pupils, projects, and lecturers in a dissertation allocation setting. To make these changes, alterations needed to be made to the readNetworkFromFile() function and the printFlow() functions, as well as parts of the findAugmentingPath() and FordFulkerson() functions and the constructor for the Network class.

Changes made to disallow backwards paths, as these can't exist in the real world for the problem being modelled

The alterations needed to be made to fundAugmentingPath() and FordFulkerson() were simply to disallow backwards edges; these conditions were removed in the graph traversal, which meant that edges could not be created to allow flow from a project to a student. This was important, because it wouldn't have been realistic within the scenario we were modelling in real life.

Changes made to reading the network in from the file when dealing with students, projects, and lecturers

The readNetworkFromFile() function needed to split the input file up into sections for students, projects, and lecturers. The first three lines of the file were read so that the function could jump directly to parts of the input file it needed, as these three lines contained the number of students, projects, and lecturers that the file contained. The function could then parse the appropriate lines, creating a Network object with the new constructor (described below), and adding edges according to the information contained in the file: links between student vertexes and project vertexes where the students' choices were appropriate for their course; links between projects and lecturers according to the capacity the project could take; links between lecturers and the terminal to complete the graph and enforce the lecturers' own limits on the students they could take.

Vertex navigation and changes to the Network constructor

The vertexes were organised like a regular graph, and the data structures used were ultimately unchanged; instead of changing the data structures, the Network constructor was given a second set of parameters with numbers of students, projects, and lecturers,

so that the labels of the vertexes could be appropriately navigated. The vertexes were implicitly partitioned, so that we could create a heuristic where we knew the number of students, projects, and lecturers, and that they were kept in the network in that order. In this way, the appropriate vertices could be obtained by getting them from their label, and adding the appropriate amounts to the number of what was wanted so that the number was within its class in the network.

Changes made to printFlow() for the project allocation network printing

Finally, the printFlow() function needed to be changed to account for the fact that the edges in the graph had and associated meaning to be conveyed. The graph was traversed, counting and totalling information from the edges that connected each student, project and lecturer vertex. After processing each vertex, its information could be added to a StringBuilder that collected the output and ultimately printed it at the end. The output format was designed to meet the specification.

Part C

Ultimately, not enough time was left to attempt Part 3, and no implementation could be provided.