Programming Comps, January 2010

The task is to implement one part of the approximation algorithm for maximum multi-commodity flow. The part that we have been assigned to do is to route flow in a graph via shortest paths, and the algorithm that we use for shortest paths is Dijkstra's algorithm.

Programming Language

I chose to use Java for this project because I think it has an appropriate balance between speed and maintainability. Its main advantages are portability, ease of programming (due to not having to deal with things like pointers and header files), and the utility of the APIs. C++ is usually faster than Java, but for this task C++ would gain most during the startup of the program, and in file input and output, neither of which count towards the elapsed time that is being measured.

Structure of Code

Efficiency of memory and time was a major factor in my choice of data structures. Although there are a few places in the code which would look cleaner if there were a separate object for every arc on the graph, running out of heapspace would be a concern when working with graphs that have millions of arcs. Instead, I decided to only make objects for the nodes, and to store the arc data in arrays. It is more useful to have the nodes be objects, because the nodes of the graph have a lot of data that gets passed around and modified while the program is running.

Another factor that I considered in the design was testing and debugging the program. The priority queue and the file I/O code are not in the main class because it was easier to test each of the different parts of the program separately. Finding lambda and calculating the max flow are separate from the main class and Dijkstra's algorithm implementation for that reason as well.

The Priority Queue

The implementation of the priority queue is based on a minheap array, where the key for deciding the positions of the nodes is the distance of the shortest known path to the node from the supply node. The minheap is a tree-based data structure that maintains the property that the key of the parent node will always be less than or equal to the key of its child. The important operations of the heap-based priority queue are:

- Offer: When a node is added to the heap, it takes the last available position at the bottom of the balanced heap. From there it is a O(log n) operation (where n is the number of nodes) to upheap the new node so that it has the correct position in the heap.

- Poll: The node with the smallest key is the head of the minheap, found at the first element of the heap array. The head is replaced by the last node of the balanced heap, and then there is a O(log n) operation to downheap the new head in order to restore the heap property.

- ChangeKey: When a node that is already on the heap gets an updated key, the position of the node may need to be changed in order to maintain the heap property. Since Dijkstra's algorithm finds the minimum distance path, we know that the value of the key will never grow larger. So once we change the key, we find the node in the heap, then upheap the node to restore the heap property, which is also a O(log n) operation.

In my implementation, the heap is implemented using two arrays. The heap itself is an array, and it only takes simple constant time arithmetic operations to traverse up and down the tree. The second array is the heapIndex array, which is used to look up the position of any node on the heap in constant time. This is needed for the changeKey operation, because otherwise it would be a linear operation to check every node and find the one that we want.

Although the java.util.PriorityQueue implementation is also based on a heap, I did not use it because it does not have a changeKey operation.

Dijkstra's Algorithm

Dijkstra's algorithm can be implemented efficiently by using a priority queue. The time for this implementation of Dijkstra's algorithm is O(m log n), where m is the number of arcs, and n is the number of nodes. That is because every time a new node is added to the set of explored nodes, every arc leading out of that node is checked to see if it is part of a new minimum path. At most, each of those m operations will lead to an O (log n) offer or changeKey operation on the priority queue, hence O(m log n).

Nodes are added to S, the set of explored nodes, in order of their distance from the supply node. In this implementation of Dijkstra's algorithm, a node is not added to the priority queue until there is exactly one arc that connects it to a node that is in S. The algorithm begins with the supply node as the only node in S, and all of the nodes that are connected to the supply node by a single arc are added to the priority queue. The algorithm finishes when there are no nodes left in the priority queue.

Calculating Lambda and Max Flow

After finding the shortest paths to the supply node using Dijkstra's algorithm, the program routes the flow and then scales the flow maximally so that it obeys the capacity constraint. To find the total flow on each arc by we need to trace the path that each node takes back to the supply node, and add up the demand that will be on each arc.

This part can be made more efficient by marking nodes in the graph as leaves or not leaves. A node is a leaf if it has no arcs going out of it, or if none of the arcs going out of it are a part of any other node's shortest path. If we only trace the shortest paths to the supply node from the leaf nodes, then we can count the demand of the non-leaf nodes as we pass through them. When the algorithm is tracing the path of a leaf node through a non-leaf node, it adds the demand of the non-leaf node to its own demand, and then sets the demand of the non-leaf node to zero. That is so the non-leaf node's demand doesn't get counted more than once if it is part of more than one leaf node's path.

Every time we change the flow on an arc, we check to see if the current value of lambda is violating the capacity restraint on that arc, and change the value of lambda if it is. Once we have counted the demand of all the nodes, we maximize the flow by multiplying the flow on every arc by lambda.