Blake Boswell

010728180

Algorithms HW 9

**Approach**

My approach was to first figure out a way to detect all possible cycles in a graph. Initially, I applied a simple cycle detection with DFS, but that approach fell short of traversing all cycles. Finally, I came to the conclusion to start with a backtracking approach, checking all edges of each vertex recursively, until a simple cycle is found. Once the cycle is found, the program checks the cycle for an inefficiency by accumulating a profit by multiplying each edges exchange ratio together. Given an edge (u, v) the exchange ratio is the fraction of v’s weight divided by u’s weight. For instance, the input 1 2 2.0 4.0 would translate into “One kg of product 1 can be exchanged for 4.0/2.0 kg of product 2.” If the cycle creates a profit, then the recursion stops, and the output is written to an output file. If the cycle does not create a profit, then the program keeps exploring new sequences until either all possible sequence permutations are created, or a cycle with profit is found.

**Design**

The program contains a Graph class which takes care of all the calculations that must be done on the graph representation of the trade network. It stores the number of vertices, or products in our usage, the adjacency list, the exchange rates from vertex u to v, and the original exchange vertex weights for vertex u and v.

The 2D exchange string array is only used for writing the output to the file. It works like a map, given 2 vertices, u and v, the array returns the string necessary for the output weights. For example, the input 1 2 2.0 4.0 would store an entry for exchange[1][2] = “2.0 4.0”.

The 2D exchangeRates double array is used for calculating the profit generated within a cycle. Given an edge (u, v) we can use exchangeRate[u][v] to multiply through the vertices of the cycle to come up with a profit and allow the program to check for an inefficient trade in the graph.

The logic of the program resides in the findInefficiency and findInefficientCycle functions. The former does some initialization and checking to ensure findInefficientCycle is run correctly for any type of graph. The alreadyVisited array ensures that we visit all vertices of the graph, to make sure any disconnected graphs are fully searched through each component as well. FindInefficientCycle is a recursive program that recursively does the following:

1. Check if the sequence we have traversed contains a cycle
   1. If there is a cycle, check it for profit to prove the graph is inefficient
   2. If there is a cycle and no profit, backtrack by removing the current vertex from the sequence, and searching for another unexplored sequence.
   3. If there is no cycle, continue
2. Go through each adjacent vertex to most recently explored vertex, recursively calling this function.
3. When the recursive call returns, backtrack by removing the current vertex (on the recursive instance) from the sequence and return whether or not the recursion should stop (inefficiency found).

**Analysis**

The worst case for this algorithm is when the graph is a strongly connected graph with no inefficiencies. If there exists an inefficiency the program will terminate early with a result. In the case where the permutation of the sequence is near the end of the backtracking execution, or there doesn’t exist a sequence that produces an inefficiency, the program runs in O(V!). This is each recursive call will call itself V times on V – L vertices, where L = {0, 1, 2, …, V} is the level of the tree, ending up with O(V!) calls. Of course, as the number of products in the trade graph grows on a strongly connected graph, the runtime grows at a greater and greater rate (factorial), which is not at all ideal. This program is a good start, but it is not completely solved with a runtime as inefficient as that.

**Conclusion**

To make this program more efficient, I would need to come up with a new algorithm or explore more ways to prune the backtracking solutions. Upon doing some research, I have seen Johnson’s algorithm can be used to detect all simple cycles most efficiently, but I could not understand how to utilize it in a way to detect all simple cycles. In conclusion, this program still has much room for improvement in terms of runtime.