COURSE: CS2500 - ALGORITHMS I

INSTRUCTOR: DR. CHAMAN LAL SABHARWAL

MAX FLOW PROBLEM:

FORD-FULKERSON VS EDMONDS-KARP

May 3, 2016

Connor Coward

Missouri University of Science and Technology

5/3/2016

Contents

MOTIVATION

Moore's law states that computing power will double roughly every two years. For decades, this held true; however, in recent years, Moore's law has slowed as hardware manufacturers struggle to continue to shrink die size. Increasingly, the focus has turned to improving software algorithm efficiency over hardware in order to keep up with demand. Software users want the capability of processing more and more data in less and less time.

One well-known problem in computer science is the max flow problem for networks and graphs. This problem has far-reaching applications in areas such as electrical network analysis, routing raw goods to manufacturing facilities, improving network performance, and much more. A better understanding of the advantages and disadvantages of different solutions to the max flow problem can lead to significant improvements in these areas. This report analyses various common solutions to the max flow problem in order to better understand the advantages and disadvantages of the solutions.

LITERATURE REVIEW

Problem Description

Given a directed, weighted network with one or more sources and sinks, the task of the max flow problem is to determine the maximum flow from all of the sources to all of the sinks [2] There are many methods to solve this problem which are discussed in the sections below.

Solutions

Brute Force

One possible method to solving the max flow problem would be with brute force. That is, trying every possible legal flow value for each edge in the graph, and checking which combination results in the highest total flow from source to sink. Note that for the flow values to be legal, the input flow must equal the output flow for each node in the graph. The brute force solution is guaranteed to give a correct solution, however, for graphs with more than a few nodes, the brute force solution is terribly inefficient. The algorithm would have a complexity on the order of $O(max(F)^N)$ where max(F) is the maximum flow of the nodes and N is the number of nodes. This is because the algorithm must perform operations for all possible flow combinations for each node.

Ford-Fulkerson

The Ford-Fulkerson algorithm was first described in 1956 by L. R. Ford, Jr. and D. R. Fulkerson [1]. This algorithm involves checking for a valid path of flow, and incrementing the flow by the minimum flow of each edge in the path. This process is repeated until there is no more path capable of supporting more flow. The Ford-Fulkerson algorithm is $O(E * max(F))$ where E is the number of edges and max(F) is the maximum flow [2].

Edmonds-Karp

Similar to the Ford-Fulkerson algorithm, the Edmonds-Karp algorithm also involves checking for a valid path of flow, and incrementing the flow by the minimum flow of each edge in the path. However, the Edmonds-Karp algorithm improves upon the Ford-Fulkerson algorithm by using a breadth-first search to find augmenting paths. This improves the time-complexity of the algorithm. In fact, the Edmonds-Karp algorithm has a time complexity on the order of $O(N * E^2)$ where N is the number of nodes and E is the number of edges in the graph [2].

Multiple-Source Multiple-Sink

Note that multiple-source multiple-sink (MSMS) graphs can also be analysed with the same algorithms by simply adding a 'super source' which feeds all the sources and 'super sink' into which all the sinks feed. The connections to and from the super source and super sink must be infinite flow. Infinite flow for these edges can be achieved by setting the maximum flow to be the maximum possible integer value. For almost all graphs, this will yield accurate results.

PSEUDOCODE

Ford Fulkerson

The pseudocode algorithm for the Ford-Fulkerson solution is shown in the code listing below:

```
FordFulkerson(G):
        residual = G
        while (parentList = findPath(residual)):
                # Invariant: parentList contains complete path
                #     from source to sink
                # Invariant: size of parentList is number of
                #     columns/rows in g
                minimum flow = 0
                for each node in the path:
                        minimum = min(node.flow, minimumFlow)
                for each node, v, in the path:
                        u = parentList[v]
                        residual[u,v] -= minimumFlow
                        residual[v,u] += minimumFlow
                maxFlow += minimumflow
```

. . .and the findPath() function is implemented as follows:

```
# Pre-condition: G is a 2-d array containing a node map
#     to describe a graph.
FindPath(G, V, P, node)
        if node is the sink node:
        return true
    for each link in the node map:
        if not V[link] and G[node, link]:
                V[link] = true
            if (FindPath(G, V, P, link):
                P[link] = node
                return true
     return false
# Post−condition: True if a path is found, otherwise false.
#                 P contains parent list of path (if found)
```

Edmonds-Karp

The Edmonds-Karp implementation is very similar to the Ford-Fulkerson implementation, except that the findPath algorithm is a breadth-first algorithm instead of depth-first.

```
# Pre−condition: G is a 2−d array containingap
#     to describe a graph.
FindPath(G):
        q.push(G.source)
    visited[G.source] = true
    parent[G.source] = −1

    while not q.empty():
        u = q.pop()

        for each node in G:
                if not visited[node] and node.flow > 0:
                q.push(node)
                parent[node] = u
                visited[node] = true

    return parent
# Post−condition: parent contains a parent list describing a path
```

Invaraints

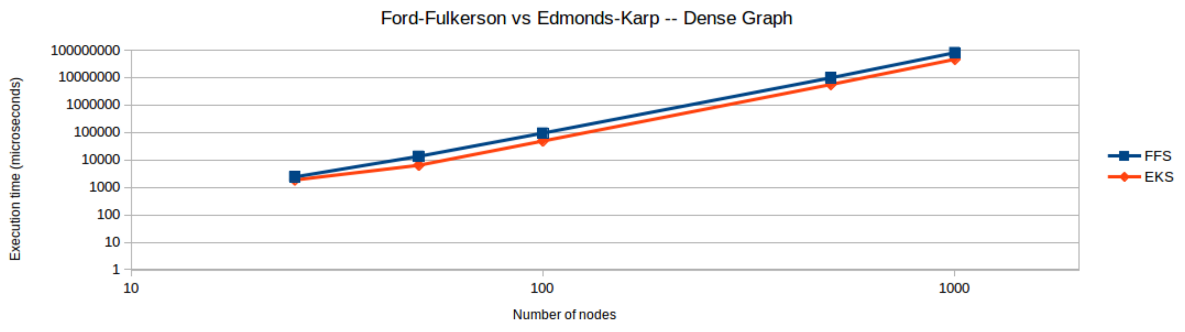
The invaraints that will be used are listed in the pseudocode above. These invariants will be implemented as asserts, which will be called to verify the validity of the values being manipulated by the program at runtime.

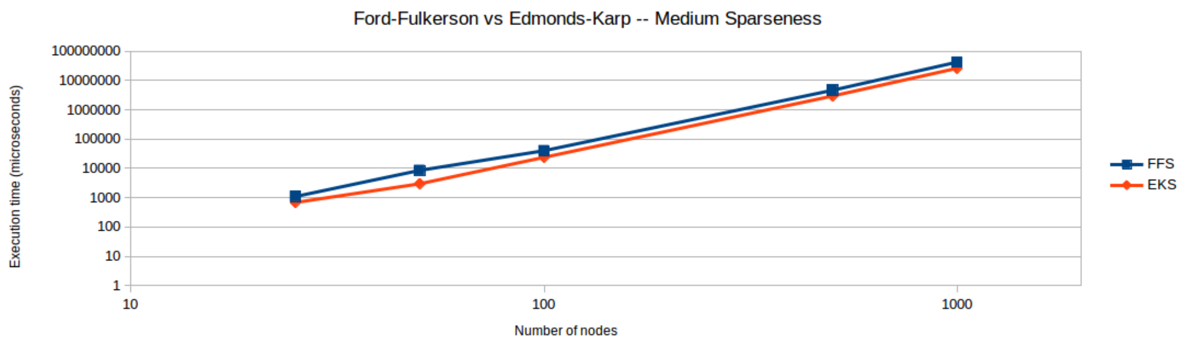
TESTING PLAN

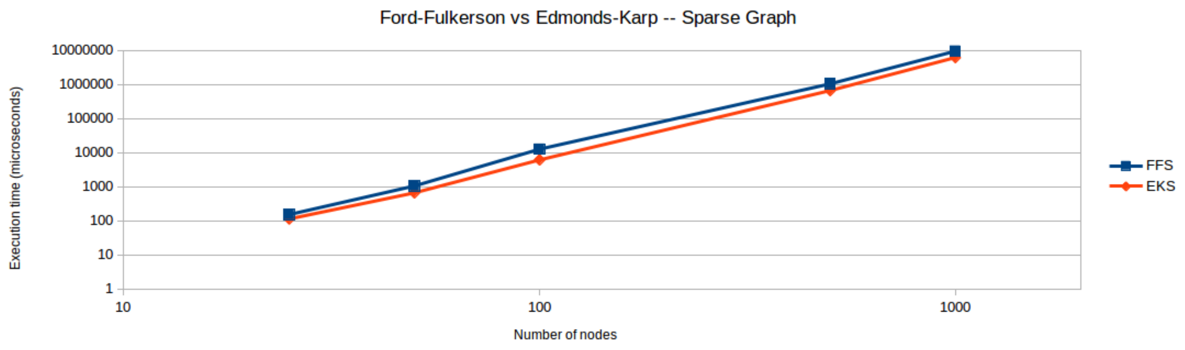
In order to analyze and compare the two algorithms, a thorough testing plan will be implemented. Both algorithms will be tested for single-source single-sink as well as multiple-source multiple-sink. In addition, three levels of sparseness will be tested, corresponding to 10 percent, 50 percent and 90 percent of the possible edges existing. Finally, the number of nodes will be plotted against the runtime for analysis.

RESULTS

The following charts show the run times versus item count for a range of node counts:





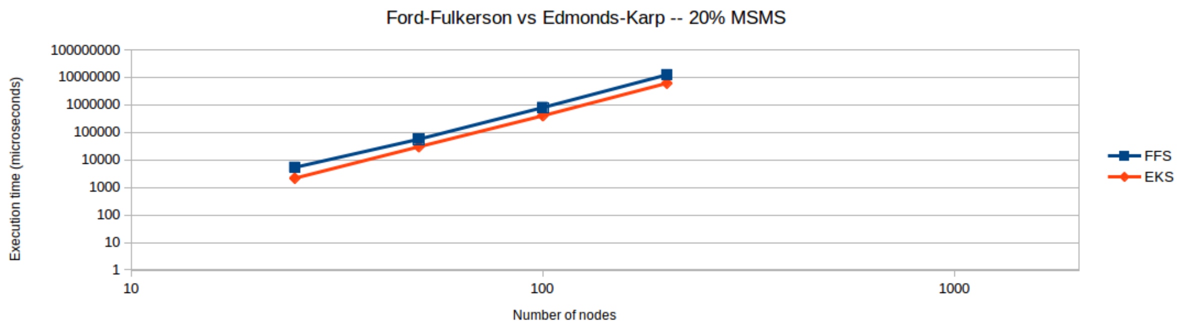


Note that the Edmonds-Karp algorithm performed consistently better than the Ford-Fulkerson algorithm. This held true across a wide range of node counts and sparseness.

This is due to the Edmonds-Karp algorithm utilizing a breadth-first search to determine augmenting paths rather than a depth-first search.

Also, the sparseness of the graphs had a significant impact on the run time of the algorithms for both Ford-Fulkerson and Edmonds-Karp. The difference is roughly one order of magnitide (10 times) between the dense graph (90%) and the sparse graph (10%).

The following chart shows data for multiple-source multiple-sink. The number of sources and sinks is set to 20% of the number of nodes. Clearly, adding multiple sources and sinks drastically increases the run-time of the algorithm (again, by about 1 order of magnitude in this case).



Bibliography

[1] "Maximum Flow Problem." Wikipedia. Wikimedia Foundation, n.d. Web. 03 May 2016.

[2] Nedich, A. (2009, October 28). Max-Flow Problem and Augmenting Path Alorithm. Retrieved May 3, 2016, from http://www.ifp.illinois.edu/~angelia/ge330fall09_maxflowl20.pdf