

Matthew Quigley
4 December, 2011
CS253 Algorithms
Project 4 Phase II

Max Flow

In this project, the Edmonds-Karp algorithm will be implemented and examined for accuracy and run time complexity. The Edmonds-Karp algorithm is almost identical to the Ford-Fulkerson algorithm except that the shortest path from the source to the target is used for the augmenting path. These algorithms both are designed to find the maximum flow in a flow network. Max flow algorithms can be used to analyze computer networks and other physical systems that can be modeled as flow networks such as circuits, water distribution systems, shipping networks, etc.

The code for the Edmonds-Karp algorithm was implemented based on the pseudo-code discussed in class:

```
//Edmonds Karp Max-Flow Algorithm
//Pre: Graph G contains root node s and destination node t
//Post: Modifies graph G so that it has the maximum flow
edmondsKarp(G,s,t)
    //invariant check:
    for each edge e in G
        assert(e.flow<e.capacity)
    for each node v in G except s & t
        assert(v.flowIn==v.flowOut)
    //end invariant check
    R = buildResidualGraph(G)
    while there is a path from s to t in R
        p = shortestPath(s,t)
        G.augment(p)
        R = buildResidualGraph(G)
        //invariant check:
        for each edge e in G
            assert(e.flow<e.capacity)
        for each node v in G except s & t
            assert(v.flowIn==v.flowOut)
        //end invariant check

    //invariant check:
    for each edge e in G
        assert(e.flow<e.capacity)
    for each node v in G except s & t
```

```

        assert(v.flowIn==v.flowOut)
    //end invariant check

```

Then the following pseudo-code was used to develop the augment and build residual graph functions:

```

    //Augment path in graph with path from residual graph
    //Pre: path exists in the graph and residual graph
    //Post: path in graph is augmented with values from path in residual
    graph
    augment(path)
        follow path back to source finding the min_flow
        for (edge in path)
            if edge[residual].type = forward
                edge.flow += edge[residual].capacity
            if edge[residual].type = backward
                edge.flow -= edge[residual].capacity

//Build a residual graph from G
//Pre: Graph G
//Post: Returns a Graph containing the residual capacity from G
buildResidualGraph(G)
    R = new Graph(G.size)
    for e in G.edges()
        forward_capacity = e.capacity - e.flow
        backward_capacity = e.flow
        if forward_capacity >0
            R.insert_edge(e.source, e.destination, 1, forward, forward_capacity)
        if backward_capacity >0
            R.insert_edge(e.destination, e.source, 1, backward, backward_capacity)

    return R

```

A Breadth first Search algorithm was used to ensure that a path exists between s and t. The pseudo-code for this algorithm is:

```

BFS(G, s)
    for each vertex u in G.V - {s}
        u.color = WHITE
        u.d = 1
        u.p = NIL
    s.color = GRAY
    s.d = 0
    s.p = NIL
    Q D ;
    ENQUEUE(Q, s)
    while Q != 0;
        u D DEQUEUE.Q/
        for each v ∈ G:Adj[u]
            if v.color == WHITE

```

```

        v.color = GRAY
        v.d = u.d + 1
        v.p = u
        ENQUEUE(Q,v)
    u.color = BLACK

```

(from the book page 595)

Then a function was implemented to extract the shortest path between two nodes from the results of the BFS based on the print path pseudo-code found in the book on page 601:

```

Print-Path(G,s,v)
    if v == s
        print s
    else if v.p == null
        print "no path from s to v exists"
    else
        Print-Path(g,s,v.p)
        print v

```

Finally, the graph class, min heap class, SSSP functions and graph generation functions from phase I were imported so that they could be used. Some minor changes had to be made to the graph class so that it could support both flow and capacity as well as the edge type labels (e.g. forward & backward). An update edge function was also added to the graph class to allow edge weights to be easily modified. Then for testing the invariants, functions were implemented to find the incoming and outgoing edges for a given node in a directed graph.

After implementing all the algorithms and their invariants, tests were run on them using the following psuedo-code:

```

function testEdmondsKarp()
    var G = generate_graph_no_parallel()
    time_before = date()
    edmondsKarp(G, 1, nodes-1)
    time_after = date()
    runtime = time_after-time_before;
    return G

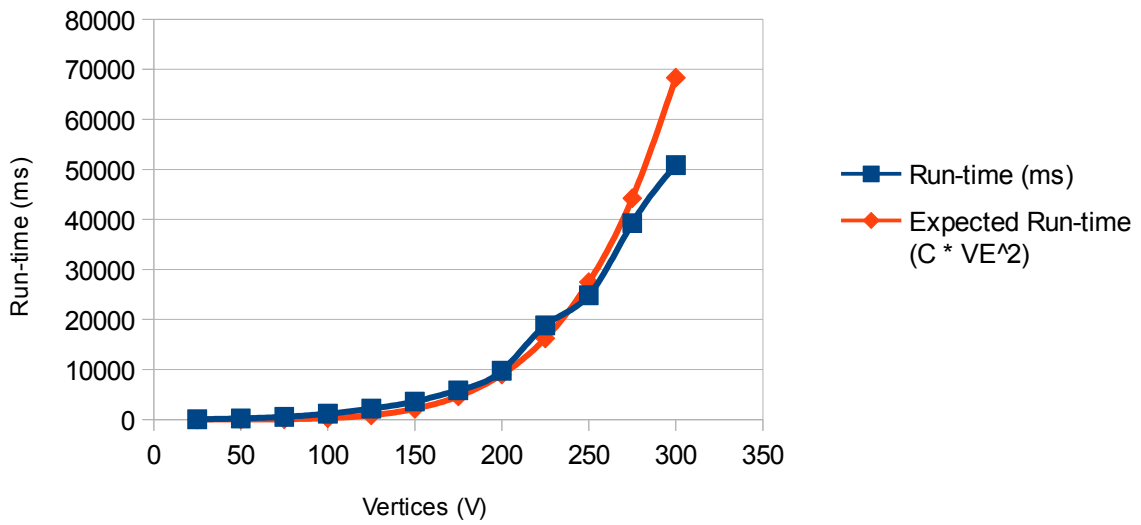
```

In addition to testing everything with the test functions, each algorithm was tested individually using the debug console. A few problems showed up while testing. The first problem was that the augmenting function did not correctly update the edge weights leading to an infinite loop within the Edmonds-Karp function. After this was addressed a few smaller problems came up that were quickly addressed and fixed. The results of the tests were examined using the debug console as well to further verify the correctness of the results. Several edge cases were tested. The first edge case was a graph containing parallel edges which originally caused the program to fall into an infinite loop. The next edge case tested was an empty graph and a graph without a connection from s to t. Both of which just exited without modifying the graph.

The expected run-time of the Edmonds-Karp algorithm is $O(VE^2)$. The Graphs below verify that the experimental run-time is approximately equivalent to the expected run-time. For an edge density of .75 a leading constant of $5.0 \cdot 10^{-8}$ was used, for 1.0, a constant of $1.85 \cdot 10^{-8}$ was used, and for 0.5 a constant of $6.75 \cdot 10^{-8}$ was used. These constants created lines that most closely matched the experimental run-times.

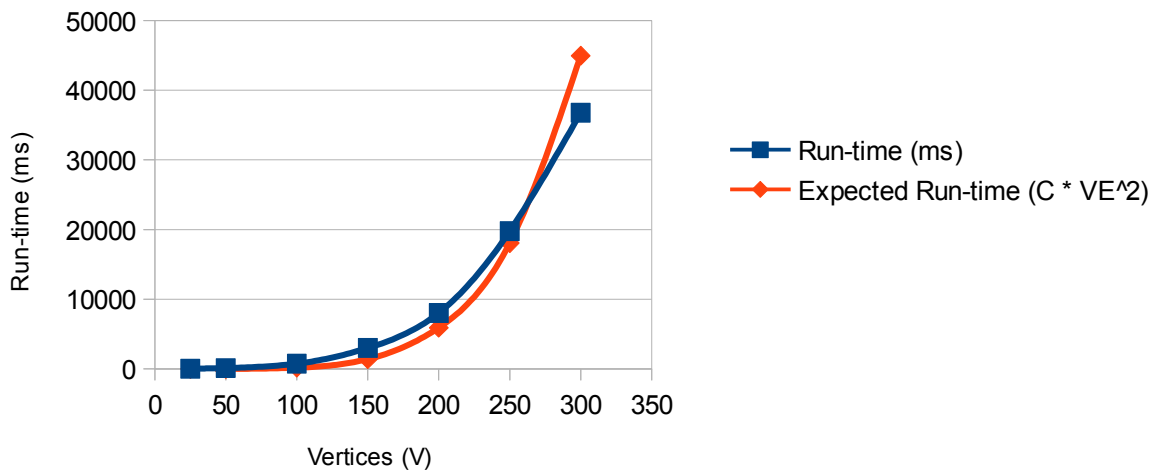
In this project, the run-time and validity of the Edmonds-Karp algorithm were verified experimentally. The Edmonds-Karp algorithm runs in $O(VE^2)$. The algorithm's correctness was shown through demonstrating that its invariants hold. Since the invariants hold, it has the greedy choice property, and the greedy choice is being made by selecting the smallest path, it will find the correct solution to the max-flow problem. Max-Flow algorithms can be used in networking and other fields that rely on flow networks. It can also be used as a cycle detector with a few minor modifications.

Edmonds-Karp (.75 Edge Density)



Edmonds-Karp runtime

(1.0 Edge Density)



Edmonds-Karp runtime

(0.5 Edge Density)

