CSCI 311: Homework 5 Report

By John Venditti, Gabe Gomez, Tyler DiBartolo

Introduction:

Flow networks are highly applicable in real world scenarios, whether it be for liquid flow through pipes, current through wires, or any number of other uses. For this homework we were able to generate models of random flow networks and analyze the different aspects of them. We created two classes, test_network_generator (TNG) and max_flow_generator (MFG), as well as a main function to produce 10 networks and create files we can analyze. The TNG creates a random network to be used by the MFG as the input graph. MFG takes said network and updates it as a max flow network using the Ford-Fulkerson method, outputting the output graph. The dot and png files are created and placed in their respective input/output folders by the main function.

Discussion:

```
class test_network_generator:
    constructor() :
1.      vertices = {} #new empty mutable dictionary                #O(1)
2.      edges = [] #new empty mutable array                        #O(1)
3.      randomSink = random number between 7 and 15 (inclusive)    #O(1)
4.      for i = 2, i < randomSink, i++ :                           #O(V - 2)
5.          vertices[i] = [] #empty array                          #O(1)
6.      for each key 'vertex' in vertices :                        #O(V - 2)
7.          if randomSink < 9:                                     #O(1)
8.              randomA = randomly set to either 1 or 2            #O(1)
9.          else:                                                  #O(1)
10.             randomA = randomly set to either 2 or 3            #O(1)
11.         for i = 0, i < randomA, i++ :                          #O(randomA)
12.             randConnx = random choice of another vertex        #O(1)
13.             vertices[vertex].append(randConnx)                 #O(1)
14.             vertices[randConnx].append(vertex)                 #O(1)
15.             edge = (vertex, randConnx, 0, rand(5, 20))         #O(1)
16.             edges.append(edge)                                 #O(1)
17.     finalVertices = [1, randomSink]                            #O(1)
```

```
for each key ‘vertex’ in finalVertices :                              #O(2)
    vertices[vertex] = [] #empty array                                #O(1)
    randomB = randomly set to either 2 or 3                           #O(1)
    for i = 0, i < randomB, i++ :                                     #O(randomB)
        randConnx = random choice of vertex (not src. or sink)        #O(1)
        while vertices[vertex] contains randConnx :                   #O(3)
            randConnx = random choice of vertex (not src. or sink)    #O(1)
        vertices[vertex].append(randConnx)                            #O(1)
        vertices[randConnx].append(vertex)                            #O(1)
        if vertex is source :                                         #O(1)
            edge = (vertex, randomConnx, 0, randint(5, 20))           #O(1)
        else : #vertex is sink                                        #O(1)
            edge = (randomConnect, vertex, 0, randint(5, 20))         #O(1)
        edges.append(edge)                                            #O(1)
```

Run-time of test_network_generator constructor:

$$= 3O(1) + O(V\text{-}2) + O(V\text{-}2) * (4 + 5 * randA) + O(1) + 2O(2 + randomB * 12)$$

$$= O(4) + O(V\text{-}2) * (5 + 5 * randomA) + O(4 + randomB * 24)$$

$$= O(8) + 24O(randomB) + (5 + 5 * randomA)O(V\text{-}2)$$

$$= O(8) + 24O(3) + (5 + 5 * 3) * O(V\text{-}2)$$

$$= O(80) + 20 * O(V\text{-}2)$$

$$= 20\ O(V\text{-}2)$$

NOTE: Our implementation limited the number of vertices to be in the range of 7 to 14 for sake of keeping a manageable graph size in the output *dot* and *png* files - thus, the runtime is anywhere from (80 + 20 * 5 =) 180 to (80 + 12 * 20 =) 320. The number of nodes could be easily changed by changing the range of possible vertices; however, if the number of vertices were greatly increased, one would also need to make sure that the number of edges generated was high enough to not leave a sparse graph, and this would change the value of the random numbers in the analysis and thus change the overall runtime. In the above rutime analysis, we used the upper bound on the value of each of the random numbers.

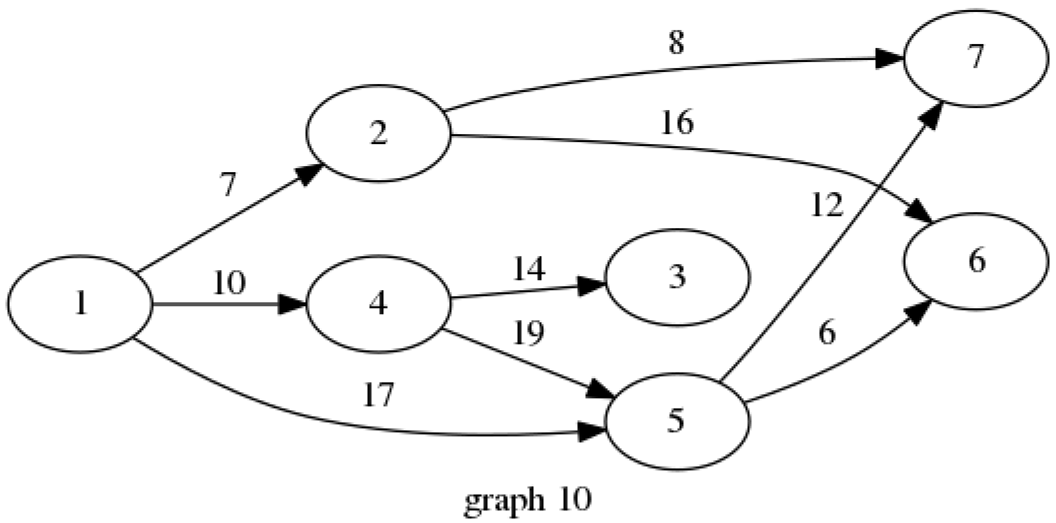
```
class max_flow_generator:
    constructor(network) :
1.      network = network # Holds the test_generator network
2.      sink = findSink()
3.      source = findSource()
4.      max_flow = solveMaxFlow(source, sink, network.edges, infinity)
5.      removeEmpty()

    solveMaxFlow(vertex, end, edgelist, bottleneck, path=[], visited=[]) :
1.      put vertex in visitedList
2.      if sink is reached:
3.          updatePath(path, bottleneck)
4.          return bottleneck
5.      addedFlow = 0
6.      for each edge in edgelist:
7.          if bottleneck == 0 : break
8.          remainingCapacity = getEdgeRemaining(edge)
9.          if edge is outgoing from vertex, the other side has not been visited, and remaining
            edge capacity != 0 :
10.             index = edgelist.getIndex(edge)
11.             put index in path
12.             oldBottleneck = Nil
13.             if remainingCapacity < bottleneck :
14.                 oldBottleneck = bottleneck
15.                 bottleneck = remCap
16.             val = self.solveMaxFlow(edge[1], end, edgelist, bottleneck, path)
17.             if val == 0 and oldBottleneck != Nil: botlleneck = oldBottleneck
18.             addedFlow += val
19.             bottleneck -= val
20.             remove index from path
21.             remove most recently accessed vertex from visited
22.     return addedFlow

    getEdgeRemaining(edge) :
1.      cap = edge[3]
2.      flow = edge[2]
3.      diff = cap - flow
4.      return diff
```

```
    updatePath(path, bottleneck) :
1.      for all edges corresponding to the indices in path:
2.          updateEdge(edge, bottleneck)

    updateEdge(edge, amount) :
1.      change flow of edge by amount

    removeEmpty() :
1.      for each edge in network.edges where flow == 0 :
2.          remove edge from network.edges

    findSource() :
1.      return 1

    findSink() :
1.      return length(network.vertices)
```

Runtime of max_flow_generator fuctions:

- solveMaxFlow will have a runtime of O(E * max_flow).
- getEdgeRemaining will have runtime of O(1).
- removeEmpty will have runtime of O(n), where n is the number of edges with flow == 0 after completing solveMaxFlow.
- updatePath will have runtime of O(n), where n is the length of the path.
- updateEdge with have runtime of O(1).
- findSource and findSink are O(1).

Input/Output Analysis:



(*Figure 1.* Randomly Generated Capacity Network)

Above is one example of a randomly generated flow network. It started by creating a random upper bound on the number of vertices by setting sink to 7. It first created all the internal (ie. not source or sink) nodes and added either 1 or 2 random edges to each so that they are connected with one another. These edges are tuples consisting of the start vertex, the end vertex, the flow, and the capacity, with the capacity being shown for each edge in the graph above. Once the internal vertices were created and linked, it added the source and sink vertices, and randomly picked to add either 2 or 3 random edges outgoing from the source or incoming to the sink, each, respectively. All of this information is stored in an object, test_network_generator.

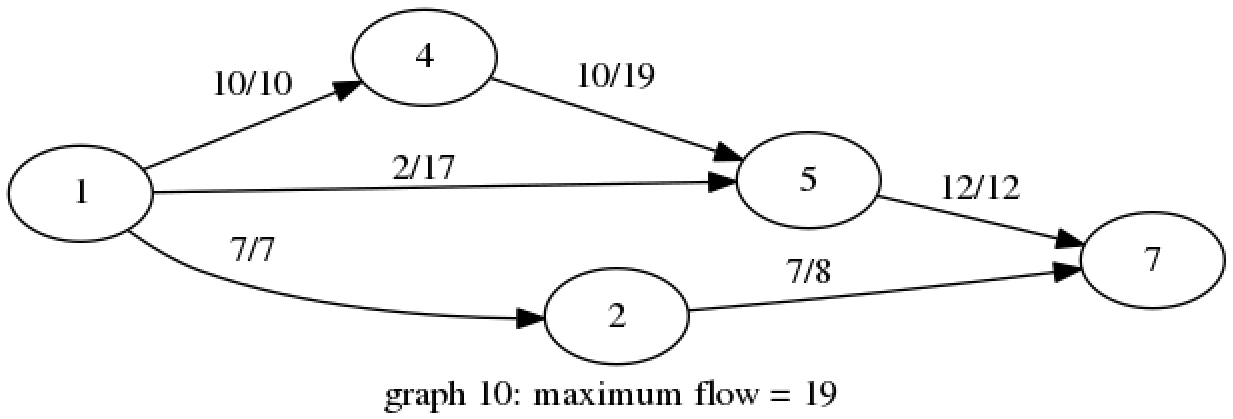
This object is passed into our max_flow_generator. The max_flow_generator object calls solveMaxFlow immediately after setting up all the variables needed for the execution of that function. That function executes as follows:

1. Starting at the source vertex, add the source to the list of visited vertices so that no cycles are formed. (This happens whenever a new vertex is passed, so I will ignore it for sake of brevity moving forward.)
2. Search the list of edges to find an edge outgoing from the source that has not already been encountered, and whose remaining capacity is greater than zero. We

will choose the edge from 1 -> 4. Follow this edge forward, replacing the existing bottleneck with the bottleneck of this edge if its bottleneck is less than the existing bottleneck.The bottleneck when starting is infinity, so we set bottleneck = 10.

3. At vertex 4, we repeat the previous two steps, searching for an outbound edge that does not create a cycle and whose remaining capacity is not zero. We will choose the edge from 4 -> 3 with capacity 14. This capacity is greater than 10, the previous bottleneck, so the bottleneck for the path is unchanged.
4. When we arrive at vertex 3, there are no outbound edges, so we return 0 added flow along this path and move back to vertex 4.
5. We again check for outbound edges, this time encountering the edge going to vertex 5 with capacity of 19, and the bottleneck of 10 is unchanged.
6. We search the list of edges to find an edge outgoing from vertex 5 whose remaining capacity is greater than zero. We will choose the edge from 5 -> 6. The edge capacity of 6 is less than the previous bottleneck of 10, so 6 becomes our new bottleneck.
7. When we arrive at vertex 6, there are no outbound edges, so we return 0 added flow along this path and move back to vertex 5, restoring the old bottleneck.
8. We again check for outbound edges, this time encountering the edge going to vertex 7 with capacity of 12, which is greater than our bottleneck of 10.
9. When we arrive at 7, we find it to be the source - thus we've found a valid path. The current bottleneck of 10 is the bottleneck for this entire path, so we add 12 to the flow of each edge along this path and return a flow of 12 so that it is added to the maximum-flow value eventually returned by the function.
10. Returning brings us back to vertex 5, and we decrement the bottleneck at 5 by the amount we just increased the flow. Vertex 5 has no further outbound edges, so we return the value of flow added up until that point and move back to vertex 4. At vertex 4, we decrement the bottleneck by the amount we just increased the flow. There are also no more outgoing paths, so we return the flow added up until that point again.
11. Returning brings us back to vertex 1, the source. We look for another outbound edge and find one with a capacity of 17 going to vertex 5. That is less than infinity, so 17 becomes our new bottleneck and we continue.
12. At vertex 5, we repeat the process, searching for an outbound edge that does not create a cycle and whose remaining capacity is not zero. (The algorithm would traverse the edge to vertex 6, but we know that's a dead end so we neglect it here.) We choose the edge from 5 -> 7 with a remaining capacity of 14-12 = 2. This capacity is less than 17, the previous bottleneck, so the bottleneck for the path is changed to 2.

13. When we arrive at 7, we find it to be the source - thus we've found a valid path. The current bottleneck of 2 is the bottleneck for this entire path, so we add 2 to the flow of each edge along this path and return a flow of 2 so that it is added to the maximum-flow value eventually returned by the function, and the vertices in the path that led to here will have their bottlenecks decreased.
14. In this manner, we continue recursing up and down the graph in a depth-first search along all possible paths. Anytime we arrive at the sink, we update the path that led us there with the bottleneck along that path, and return that bottleneck to be added to the total flow of the system while decrementing the bottleneck values of the vertices that led us to the sink as we return back up.
15. When all paths have been either followed and updated, or shown to have no remaining capacity, we have finished our algorithm, and we return the sum of all the path bottlenecks in the max-flow network.



(*Figure 2.* Result Network after Flow Computations and Clean-Up)

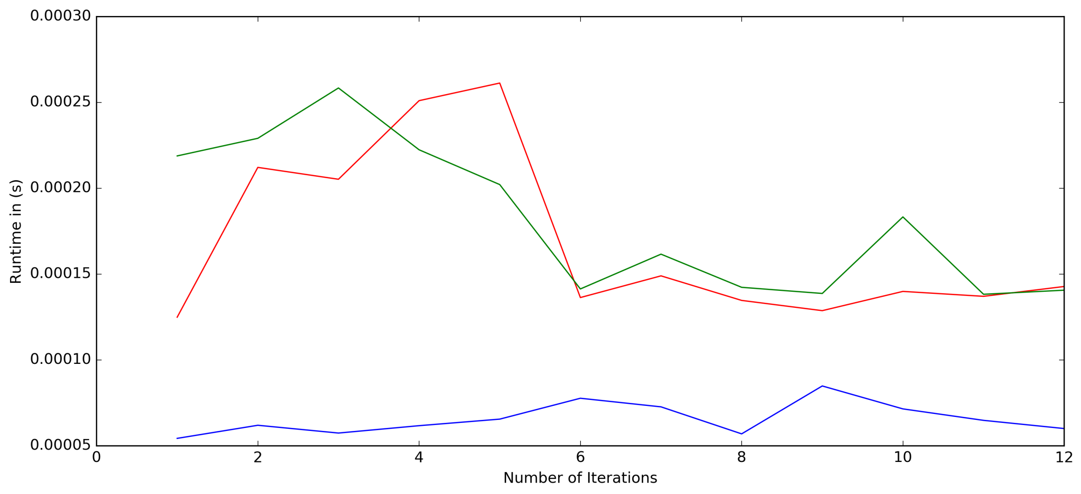
Result Analysis:

We put together 2 separate graph generator classes (not included in our project) that generate graphs for a given number of vertices, and each class can respectively augment by 10% either the number of edges in its graph, or the number of vertices. The generator which adds edges has a fixed number of vertices and thus has a limit to the number of edges that can be added; on the other hand, the generator for vertices simply adds a new vertex and connects it to the existing graph, so there is no limit to how much these graphs can be augmented - the only limit is the practical runtime for a graph with a given number of starting vertices.

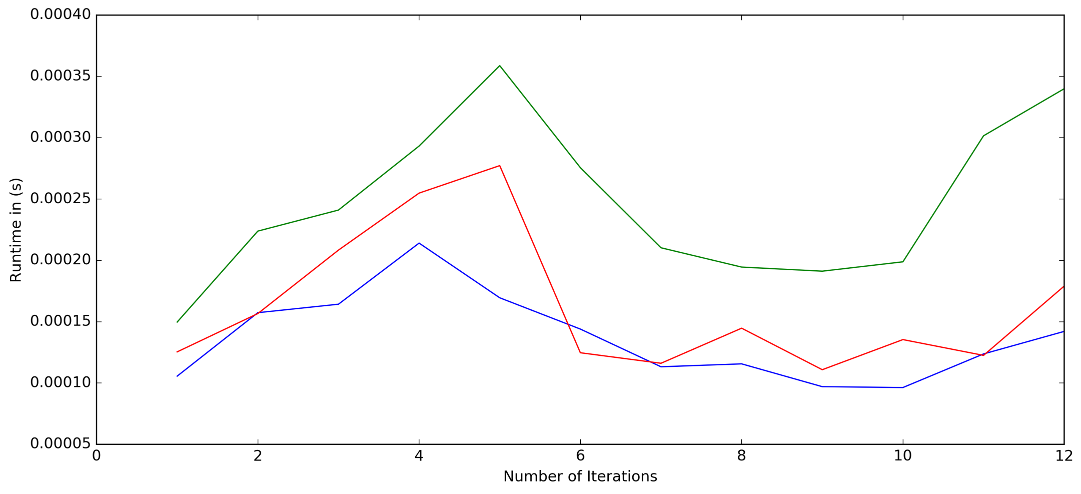
We used system timers to analyze the runtime of our max_generator algorithms when applied to subsequent iterations of augmenting our specialized generator classes. For each type of generator, we created 3 graphs, each with a different number of starting vertices (and thus edges, respectively), and we graphed the results of running our algorithms on each augmented network. The results are below, where *blue* corresponds to 7 vertices, *red* corresponds to 10 vertices, and *green* corresponds to 12 vertices.

NOTE: The simulation and analysis described here was run multiple times, and the diagrams below depict the most frequent characterizations of the relationships in question.

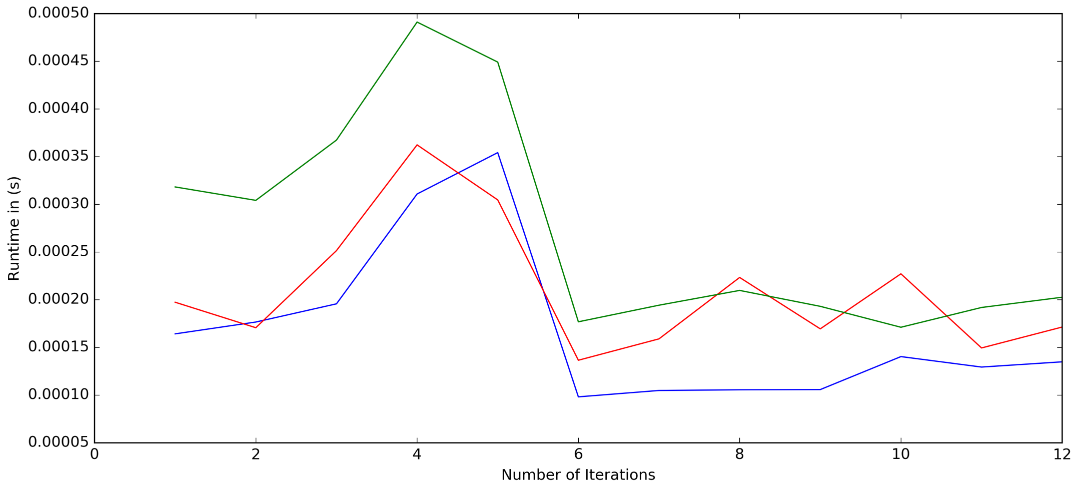
Effect on Runtime of Adding 10% More Edges



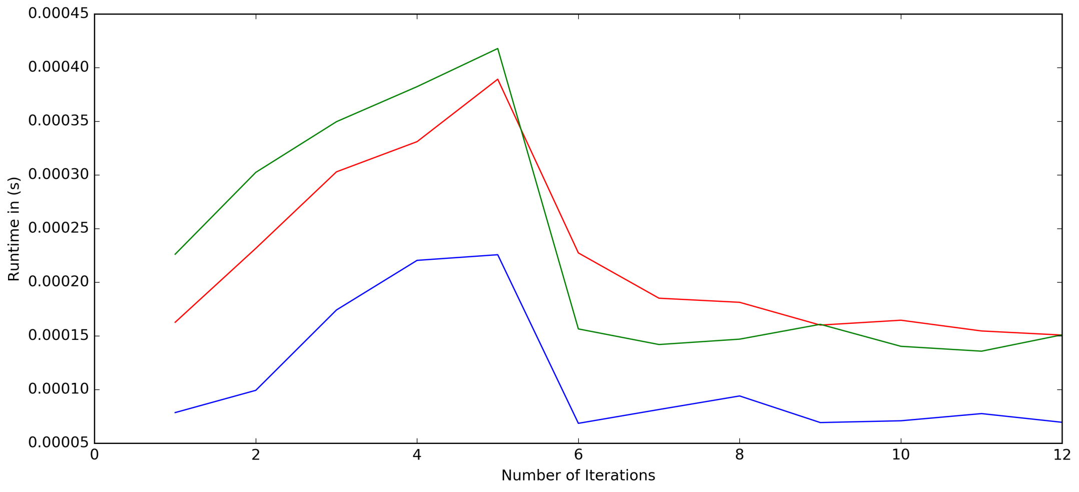
(*Figure 4.*)



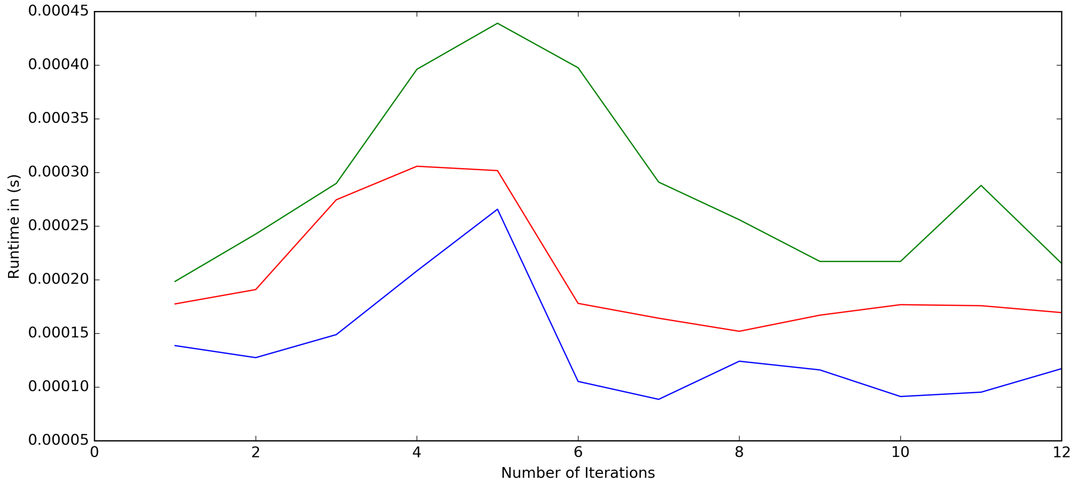
(*Figure 5.*)



(*Figure 6.*)

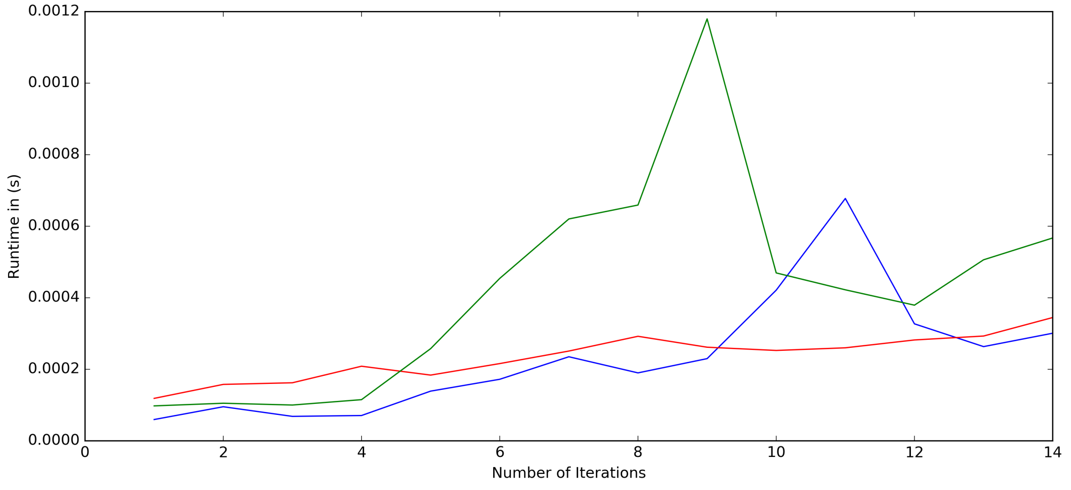


(*Figure 7.*)

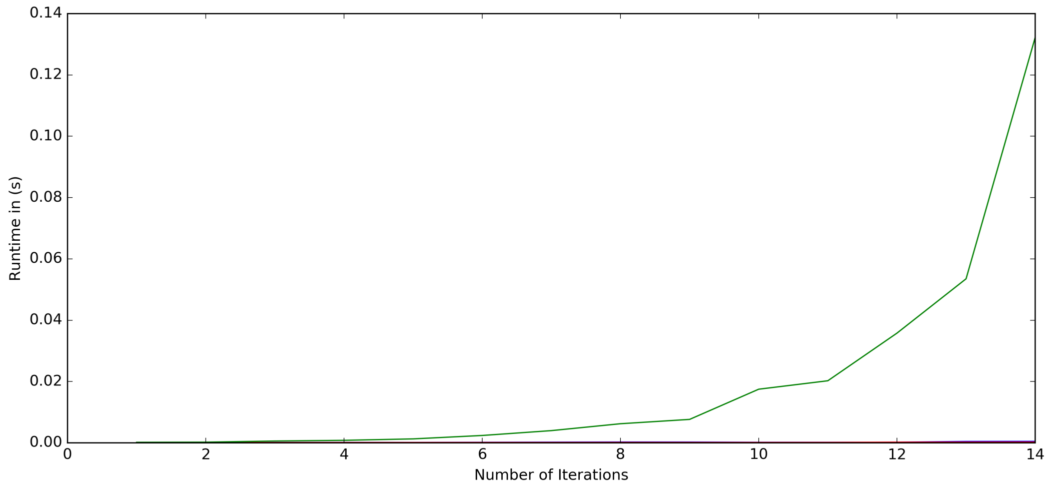


(*Figure 8.*)

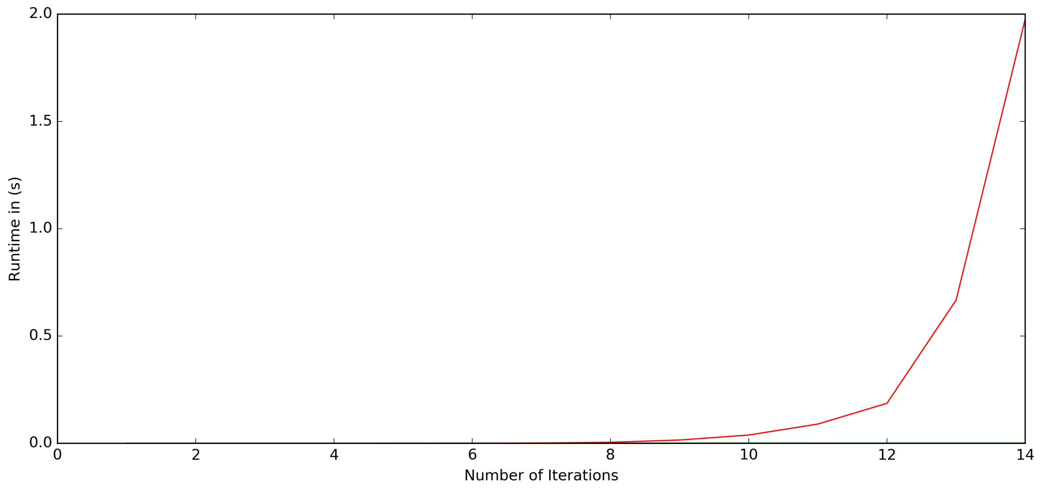
Effect on Runtime of Adding 10% More Vertices



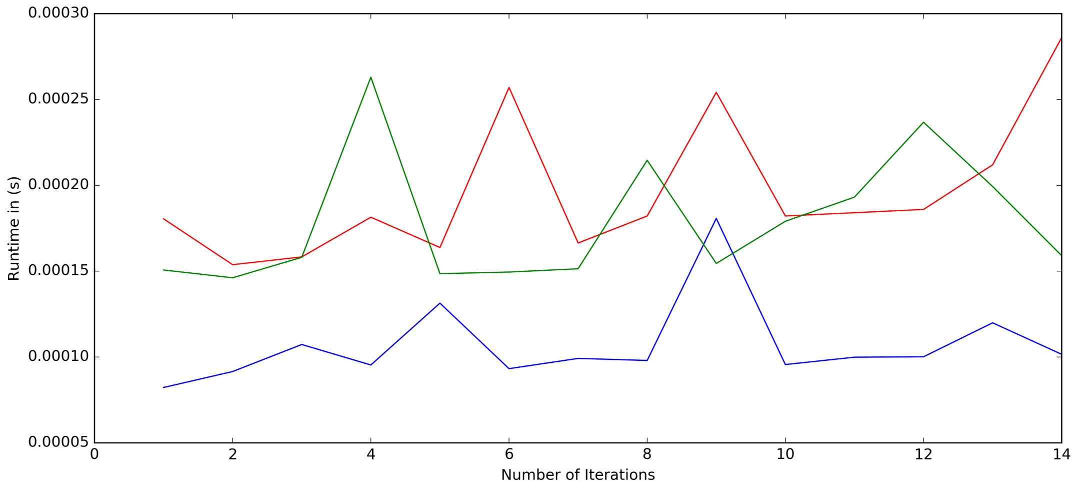
(*Figure 9.*)



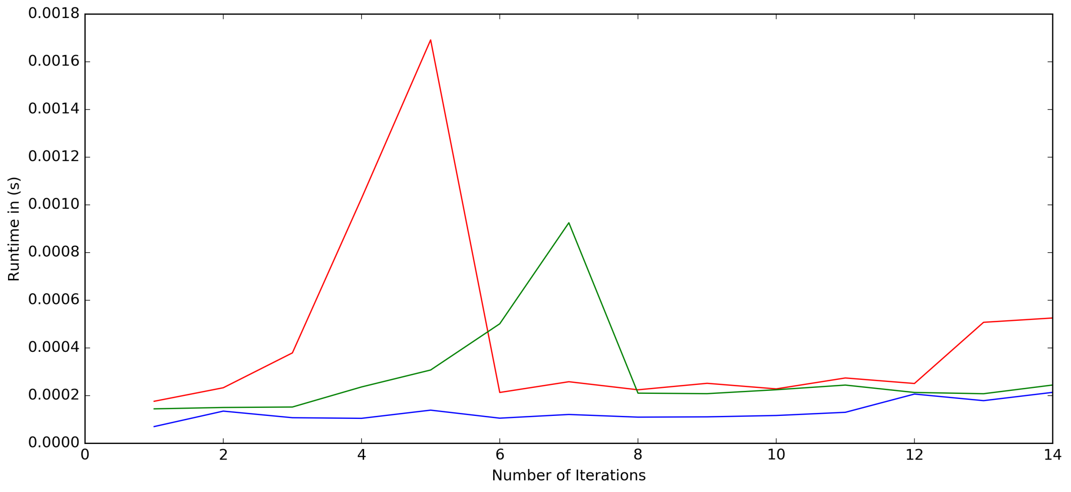
(*Figure 10.*)



(*Figure 11.*)



(*Figure 12.*)



(*Figure 13.*)

From the above graphs, each of which was generated from random sets of data, show clearly that, based on how our graph-generation algorithm augments the graphs with each iteration, adding 10% more edges to graphs with fixed numbers of vertices increases runtime to an extent, but in the long run may actually decrease running time - we believe this is due to the cross-cancellation of flows. When adding edges, the graph with more vertices generally had a higher running time, but this was not always the case, as some graphs could end up generating edges that cannot take flow due to their orientation.

However, when we look at the graphs analyzing the runtime of our max-flow algorithm as we augment the number of vertices in a graph by 10%, there are very few conclusions to be drawn. Generally, adding more vertices to a connected graph is correlated with an increase in runtime, but the total number of vertices in a graph does not seem to always be an indicator of the runtime of max-flow algorithms on that graph.

Conclusion:

The optimization of flow networks is not an easy task. In our attempt to implement a working set of algorithms to generate paths and analyze the max-flow of a network, we tried our best to implement the Ford-Fulkerson method, which is an efficient, polynomial-time method. However, we had many different iterations throughout our development of our codebase, and thus we opted for simplicity over absolute efficiency - this is clear in the many cases where we iterate over the entire set of edges in the graph. I believe, if we had a lot more time, we could have implemented a very efficient max-flow algorithm - provided that we could retool our graph-generators to produce graphs based on adjacency matrices instead of sets, dictionaries, and lists. If we could have used adjacency matrices, a lot of the iterative code used to search for edges would have been O[V] instead of O[E]. However, just creating effective graphs to begin with was a challenge, and so we spent as much time making our graph-generators work as we did getting our max-flow algorithm to work. In the end, it is clear that the Ford-Fulkerson method is actually a very intuitive method for determining the max-flow of a network, but figuring out the best way to implement it was a challenge. Our challenges in completing the tasks set out for us were not only the creation of max-flow-related algorithms, but also determining how to randomly create non-sparse graphs. Overall, we learned a lot from this assignment, and the max-flow problem shows a particularly useful application of greedy algorithms and the beneficial effect on runtime that they have when compared to non-greedy algorithms to accomplish the same task. For example, before we implemented a proper F.F max-flow, we were generating all the posslble paths in the graph and then iterating over all of them; this yielded functions with runtimes of $O(2^{V-2})$, whereas many of our functions now have linear or polynomial runtimes.