Analysis of Algorithms

Alvin Lin (axl1439) and William Leuschner (wel2138)

August 2017 - December 2017

Problem 1

Algorithm Description

The Bellman-Ford algorithm for finding the shortest path in a graph with negative edge weights conveniently encounters an error if there is a negative weight cycle. In order to determine if the graph has a negative weight cycle, we simply run Bellman-Ford and return `true` if Bellman-Ford errors out, and `false` if it succeeds.

Running Time Estimate

Since we're merely running Bellman-Ford, plus a very short constant time boolean value inversion at the end, our algorithm runs in the same time bounds as Bellman-Ford, namely $O(n * m)$. When we originally wrote this code, we figured, "Oh, m is a function of n, so it'll all work out fine in the end." Regrettably, we failed to account for the fact that, in the worst case, m is not a *linear* function of n. In a fully-connected graph, the number of edges per node is $(n-1)$, making the number of edges in the graph $(n-1)*n$, and forcing our time complexity up to $n*((n-1)*n))$.

We believe this running time is at least partially justifiable, since fully connected graphs are rare, and we couldn't think of a faster way to do it.

Pseudocode

```
def bellmanFord(edges, vtxCount, source):
  distances = []
  let every element of distances = ∞
  let distances[source] = 0
  for i = 0 to vtxCount:
    for edge in edges:
```

```
      let newDist = distances[edge.src] + edge.weight
      if distances[edge.dst] < newDist:
        distances[edge.dst] = newDist
  for edge in edges:
    if distances[edge.dst] > distances[edge.src] + edge.weight:
      return None
  return distances

let E be the set of input edges
let |V| be the number of input vertices
if bellmanFord(E, |V|, 1) is None:
  output YES
else:
  output NO
```

Problem 2

For these diagrams (Figures 1–5), the residual graphs specify backward edges in red, where the weight of the backward edge is the capacity used on that edge. Completely used-up forward edges are not shown. The augmenting path graphs specify augmenting paths in blue, where the weight of the edge describes the flow of commodity that is passing through that edge. Weights on forward edges depict the remaining usable capacity of that edge.

Problem 3

Algorithm Description

The key to this problem is that the only place an edge can be added is across the minimum s-t cut of the graph. In order to locate the minimum s-t cut, we apply the Edmonds-Karp algorithm to reduce the graph to a state where all of the existing augmenting paths from the server (source) to the client (sink) have been removed. Once the graph is in this state, we can then transpose it and build a breadth-first traversal of the transpose. This tree contains all of the nodes on the t side of the s-t cut. We then search this tree for the node with the lowest index (lexicographically first), and draw a new edge between that node and the source node.

Although we cannot determine why this algorithm fails the sixth test case, we have a hypothesis: Our constraints are wrong and we are excluding nodes that we shouldn't be excluding during the search for a node to connect the source node to. This could be due to the fact that our algorithm for calculating the predecessor nodes

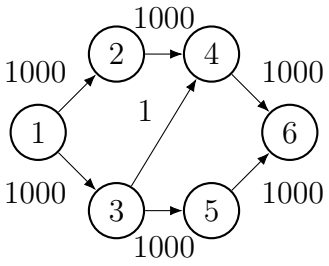
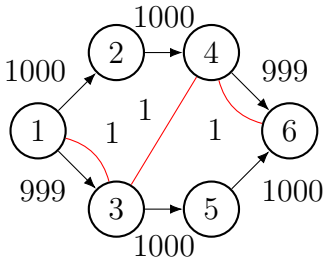
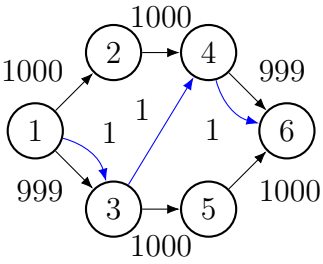


Figure 1: Residual graph before running the algorithm

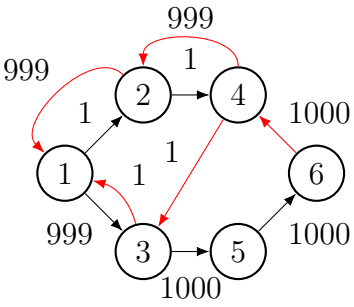


(a) Residual

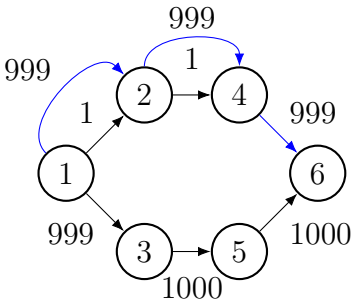


(b) Augmenting

Figure 2: Residual graph and augmenting path after 1 step of the algorithm. Flow $f = 1$.

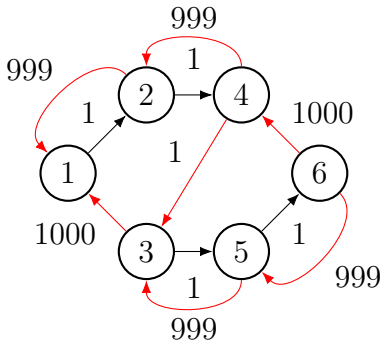


(a) Residual

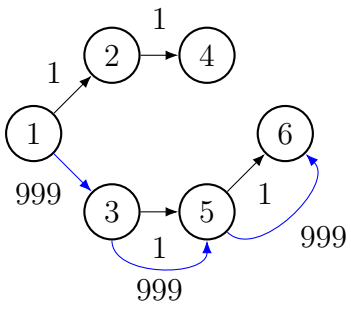


(b) Augmenting

Figure 3: Residual graph and augmenting path after 2 steps of the algorithm. Flow $f = 1000$

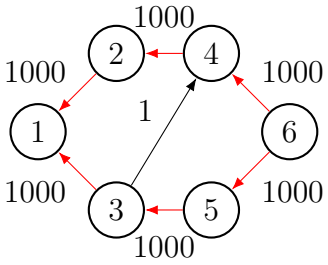


(a) Residual

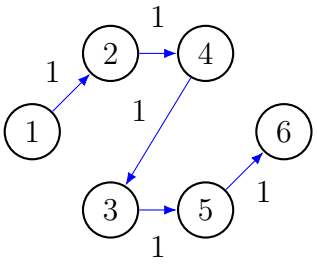


(b) Augmenting

Figure 4: Residual graph and augmenting path after 3 steps of the algorithm. Flow $f = 1999$



(a) Residual



(b) Augmenting

Figure 5: Residual graph and augmenting path after 4 steps of the algorithm. Flow $f = 2000$

in the breadth-first traversal is excluding the leaf node at the end of every branch of the tree, or at least some of them.

Running Time Estimate

This algorithm is just Edmonds-Karp with some extra stuff at the end. Edmonds-Karp runs on $O(m^2n)$ time because finding a single augmenting path requires $O(m)$ time, and at worst, this needs to be done mn times.

Pseudocode

```
def getBandwidthBottleneckEdge(graph, source, sink):
  copy = deepCopy(graph)
  predecessors, visited = bfs(graph, source)
  while visited[sink]:
    minFlow = ∞
    currentNode = sink
    while currentNode != source:
      predecessor = predecessors[currentNode]
      minFlow = min(minFlow, graph[predecessor][currentNode])
      currentNode = predecessor
    currentNode = sink
    while currentNode != source:
      predecessor = predecessors[currentNode]
      graph[predecessor][currentNode] -= minFlow
      graph[currentNode][predecessor] += minFlow
      currentNode = predecessor
    predecessors, visited = b{figure}fs(graph, source)

  predecessors, visited = bfs(transpose(graph), sink)
  nodes = filter(lambda x: x != -1, sorted(predecessors))
  for node in nodes:
    if copy[source][node] == 0:
      return (source, node)
  return "NO"
```

Problem 4

Reducing P to Q_1

Because the longest path algorithm is capped at returning a path through every node, we can simply run our graph through it unchanged and check if the output length is equal to a path through every node.

a. Construct $G_1, s_1, and t_1$ by setting them equal to their subscript-less counterparts.

b. Given ℓ, G will have a Hamiltonian path if and only if ℓ is equal to $n - 1$, where n is the number of vertices in G.

Reducing P to Q_2

This problem is slightly more difficult, but not by much. Becuase the black-box shortest path algorithm accepts negative weights, we can turn it into a longest-path

algorithm by assigning every edge in G a weight of -1.

a. Construct G_1 by assigning every edge a weight of -1. Construct s_1 and t_1 by assigning to them s and t, respectively.

b. Given ℓ, G will have a Hamiltonian path if and only if $\|\ell\| = n - 1$, where n is the number of vertices in G.