CS412 Algorithms: Design & Analysis

Spring 2024

Dhanani School of Science and Engineering

Habib University

Practice Problems

Week 9

1. Show that splitting an edge in a flow network yields an equivalent network. More formally, suppose that flow network G contains edge (u, v), and we create a new flow network G' by creating a new vertex x and replacing (u, v) by new edges (u, x) and (x, v) with $c(u, x) = c(x, v) = c(u, v)$. Show that a maximum flow in G' has the same value as a maximum flow in G.

   Solution: To see that the networks have the same maximum flow, we will show that every flow through one of the networks corresponds to a flow through the other. First, suppose that we have some flow through the network before applying the splitting procedure to the anti-symmetric edges. Since we are only changing one of any pair of anti-symmetric edges, for any edge that is unchanged by the splitting, we just have an identical flow going through those edges. Suppose that there was some edge (u, v) that was split because it had an anti-symmetric edge, and we had some flow, $f(u, v)$ in the original graph. Since the capacity of both of the two edges that are introduced by the splitting of that edge have the same capacity, we can set $f'(u, v) = f'(u, x) = f'(x, v)$. By constructing the new flow in this manor, we have an identical total flow, and we also still have a valid flow.

   Similarly, suppose that we had some flow f' on the graph with split edges, then, for any triple of vertices u, x, v that correspond to a split edge, we must have that $f'(u, x) = f'(x, v)$ because the only edge into x is (u, x) and the only edge out of x is (x, v), and the net flow into and out of each vertex must be zero. We can then just set the flow on the unsplit edge equal to the common value that the flows on (u, x) and (x, v) have. Again, since we handle this on an edge by edge basis, and each substitution of edges maintains the fact that it is a flow of the same total, we have that the end result is also a valid flow of the same total value as the original.

   Since we have shown that any flow in one digraph can be translated into a flow of the same value in the other, we can translate the maximum value flow for one of them to get that it's max value flow is $\leq$ to that of the other, and do it in the reverse direction as well to achieve equality.

2. The Ford-Fulkerson algorithm is a widely used algorithm to solve the maximum flow problem in a flow network. The maximum flow problem involves determining the maximum amount of flow that can be sent from a source vertex to a sink vertex in a directed weighted graph, subject to capacity constraints on the edges.

   The algorithm works by iteratively finding an augmenting path, which is a path from the source to the sink in the residual graph, i.e., the graph obtained by subtracting the current flow from the capacity of each edge. The algorithm then increases the flow along this path by the maximum possible amount, which is the minimum capacity of the edges along the

path.

Given a graph which represents a flow network where every edge has a capacity. Also, given two vertices source s and sink t in the graph, find the maximum possible flow from s to t with the following constraints:

1. Flow on an edge doesn't exceed the given capacity of the edge.
2. Incoming flow is equal to outgoing flow for every vertex except s and t.

Solution:

```
# Python program for implementation
# of Ford Fulkerson algorithm
from collections import defaultdict

# This class represents a directed graph
# using adjacency matrix representation
class Graph:

  def __init__(self, graph):
    self.graph = graph # residual graph
    self. ROW = len(graph)
    # self.COL = len(gr[0])

  '''Returns true if there is a path from source 's' to sink 't' in
  residual graph. Also fills parent[] to store the path '''

  def BFS(self, s, t, parent):

    # Mark all the vertices as not visited
    visited = [False]*(self.ROW)

    # Create a queue for BFS
    queue = []

    # Mark the source node as visited and enqueue it
    queue.append(s)
    visited[s] = True

    # Standard BFS Loop
    while queue:

      # Dequeue a vertex from queue and print it
      u = queue.pop(0)

      # Get all adjacent vertices of the dequeued vertex u
      # If a adjacent has not been visited, then mark it
      # visited and enqueue it
      for ind, val in enumerate(self.graph[u]):
        if visited[ind] == False and val > 0:
          # If we find a connection to the sink node,
          # then there is no point in BFS anymore
          # We just have to set its parent and can return true
          queue.append(ind)
          visited[ind] = True
          parent[ind] = u
          if ind == t:
            return True
```

```
48
49      # We didn't reach sink in BFS starting
50      # from source, so return false
51      return False
52
53
54    # Returns the maximum flow from s to t in the given graph
55    def FordFulkerson(self, source, sink):
56
57      # This array is filled by BFS and to store path
58      parent = [-1]*(self.ROW)
59
60      max_flow = 0 # There is no flow initially
61
62      # Augment the flow while there is path from source to sink
63      while self.BFS(source, sink, parent) :
64
65        # Find minimum residual capacity of the edges along the
66        # path filled by BFS. Or we can say find the maximum flow
67        # through the path found.
68        path_flow = float("Inf")
69        s = sink
70        while(s != source):
71          path_flow = min (path_flow, self.graph[parent[s]][s])
72          s = parent[s]
73
74        # Add path flow to overall flow
75        max_flow += path_flow
76
77        # update residual capacities of the edges and reverse edges
78        # along the path
79        v = sink
80        while(v != source):
81          u = parent[v]
82          self.graph[u][v] -= path_flow
83          self.graph[v][u] += path_flow
84          v = parent[v]
85
86      return max_flow
87
88
89  # Create a graph given in the above diagram
90
91  graph = [[0, 16, 13, 0, 0, 0],
92      [0, 0, 10, 12, 0, 0],
93      [0, 4, 0, 0, 14, 0],
94      [0, 0, 9, 0, 0, 20],
95      [0, 0, 0, 7, 0, 4],
96      [0, 0, 0, 0, 0, 0]]
97
98  g = Graph(graph)
99
100 source = 0; sink = 5
101
102 print ("The maximum possible flow is %d " % g.FordFulkerson(source, sink)
      )
103
104 # This code is contributed by Neelam Yadav
```

3. Given a weighted, directed graph $G = (V, E)$ with no negative-weight cycles, let m be the maximum over all vertices $v \in V$ of the minimum number of edges in a shortest path from the source s to v. (Here, the shortest path is by weight, not the number of edges.) Suggest a simple change to the Bellman-Ford algorithm that allows it to terminate in $m + 1$ passes, even if m is not known in advance.

Solution: Before each iteration of the for loop on line 2, we make a backup copy of the current d values for all the vertices. Then, after each iteration, we check to see if any of the d values changed. If none did, then we immediately terminate the for loop. This clearly works because if one iteration didn't change the values of d, nothing will of changed on later iterations, and so they would all proceed to not change any of the d values.

4. Suppose that a weighted, directed graph $G = (V, E)$ has a negative-weight cycle. Give an efficient algorithm to list the vertices of one such cycle. Prove that your algorithm is correct.

Solution: Begin by calling a slightly modified version of DFS, where we maintain the attribute v.d at each vertex which gives the weight of the unique simple path from s to v in the DFS tree. However, once $v.d$ is set for the first time we will never modify it. It is easy to update DFS to keep track of this without changing its runtime. At first sight of a back edge (u, v), if $v.d > u.d + w(u, v)$ then we must have a negative-weight cycle because $u.d + w(u, v) - v.d$ represents the weight of the cycle which the back edge completes in the DFS tree. To print out the vertices print $v, u, u.\pi, u.\pi.\pi$, and so on until v is reached. This has runtime $O(V + E)$.