

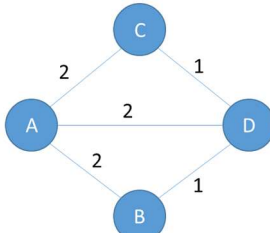
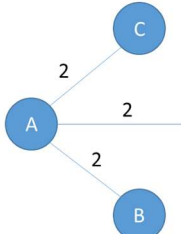
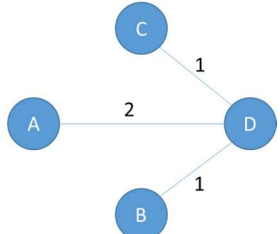
## CSC373 Assignment 2

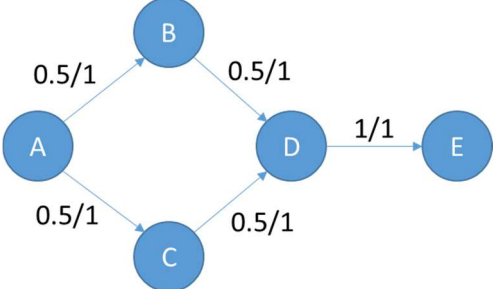
Note: all pseudo-code will be written using Python-style semantics

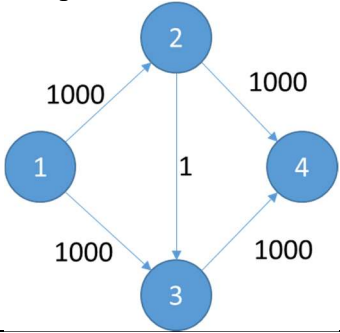
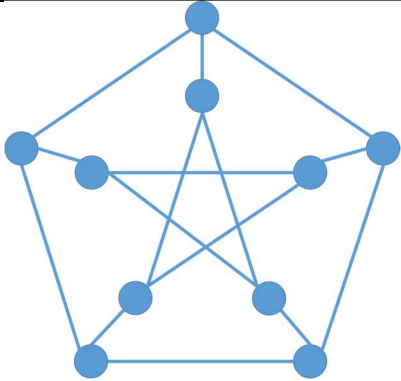
### Terminology and Notation:

Term / Notation	Meaning
$s \rightarrow t$	Directed edge that goes from node $s$ to node $t$
$s \leftrightarrow t$	Undirected edge that connects node $s$ and node $t$
$s \rightarrow\!\!\rightarrow t$	The shortest path from node $s$ to node $t$
$s \rightsquigarrow t$	A path from node $s$ to $t$
$\{J\}$	If statement $J$ has multiple correct possibilities, this is the set of all correct possibilities that satisfy $J$ (e.g. $\{s \rightsquigarrow t\}$ is the set of all possible paths from node $s$ to node $t$ )
Constrictor	A constricting edge
Augmentation	Augmenting path
CC	Connected component
$\kappa(G)$	Set of CCs in graph $G$
$\sigma(G)$	A spanning tree of graph $G$
$\mathbb{P}$	Set of irrational numbers ( $\mathbb{R} - \mathbb{Q}$ )
FF	The Ford-Fulkerson method
$O(f) * O(g) = O(fg)$	Denotes that an $O(f)$ method is being run $O(g)$ times for a total runtime of $O(fg)$
Parent Node	Node $s$ is a parent of $t$ if $(s \rightarrow t)$

1. True / False Short Answer

Claim	Verdict	Justification / Counter Example
An undirected graph with $n$ vertices and at most $n - k$ edges has at least $k$ connected components.	True	<p>Consider a graph with <math>n</math> nodes and zero edges – the number of CCs is <math>n</math> (the number of nodes).</p> <p>If we add an edge to connect two CCs into one CC, we reduce the number of CCs by 1 (from <math>k</math> to <math>k - 1</math>, where <math>k =  \kappa(G) </math>).</p> <p>Thus, if we add <math>n - k</math> edges to graph <math>G</math> where <math>\kappa(G) =  V  = n</math>, we reduce <math> \kappa(G) </math> from <math>n</math> to <math>n - (n - k) = k</math>.</p>
The shortest-paths tree computed by Dijkstra is necessarily an MST.	False	<p>Input Graph</p>  <p>Dijkstra (source is A)</p>  <p>Minimum Spanning Tree</p>  <p>We note that the total weight of the Dijkstra tree is 6, while the total weight of the MST is 4. Clearly the Dijkstra tree is not minimal.</p>
Suppose that we have computed an MST. If the weight of each edge in the graph is increased by 1, the computed spanning tree remains minimum with respect to the new weights.	True	<p>Let the graph in question be <math>G</math>, the MST be <math>m</math> and the set of spanning trees be <math>S = \{\sigma(G)\}, m \in S</math>.</p> <p>We know that the number of edges in a tree with <math>n</math> nodes is <math>n - 1</math>, so if we increase the weight of every edge in a tree by 1, the total weight increases by <math>n - 1</math>.</p> <p>Thus, the relative total weights of the trees in <math>S</math> remain the same under the new weights, therefore <math>m</math> is still minimal in <math>S</math>, with respect to the weights.</p>

Claim	Verdict	Justification / Counter Example
In a weighted directed graph with positive weights, Dijkstra might call the <i>update()</i> procedure (aka <i>relax()</i> procedure, see CLRS 24.3) on the same edge more than once.	False	<p>We inspect the pseudo-code given in CLRS 24.3 – clearly, Dijkstra only considers the leaving edges of every vertex only once for relaxation (lines 7-8).</p> <p>Since edges can only leave one vertex, Dijkstra’s algorithm must only consider one edge for relaxation.</p>
Maximum flow in a network with integral capacities is necessarily integral.	False	
We are given a weighted graph and $s \rightarrow t$ . If all edge weights in the graph are multiplied by a positive constant, $s \rightarrow t$ remains shortest from s to t with respect to the new weights.	True	<p>Let the graph in question be <math>G</math>, and <math>P = \{s \rightsquigarrow t\}</math>, where <math>s \rightarrow t \in P</math>.</p> <p>If we multiply all the weights of <math>G</math> by some <math>c \in \mathbb{R}^+</math>, the total weights of the paths in <math>P</math> are also multiplied by <math>c</math>.</p> <p>Since <math>s \rightarrow t</math> was minimal in <math>P</math> with respect to total weight (by definition), <math>c * (s \rightarrow t)</math> must also be minimal (the shortest path) in <math>c * P</math>.</p>
Suppose we run DFS on a directed graph $G = (V, E)$ and we find a vertex with discovery time 1 and finishing time $2 V $ . Then the entire graph must be strongly connected.	True	<p>We note that the (relative) finishing time of the head of the DFS tree is equal to double the number of nodes connected to the head of the tree (DFS spends 2 “turns” on each node, one going “forward” and one “backward”)</p> <p>Since <math>F = 2 V </math>, we can say that the number of nodes connected to the head of the tree is <math> V </math>.</p> <p>If <math> V </math> nodes are connected to the head, we can say there is a way of getting from any node to another.</p>

Claim	Verdict	Justification / Counter Example
Ford-Fulkerson method runs in polynomial time assuming that capacities are positive integers.	False	<p>We note that FF runs in <math>O( E f)</math> time, where <math>f</math> is the maximum flow of the target network <math>G = (V, E)</math> (CLRS 26.2). This is clearly not polynomial with respect to <math> V </math> or <math> E </math>.</p> <p>Example: the following takes 1000 iterations to complete with traditional FF</p> 
Ford-Fulkerson method terminates on all input flow networks with capacities that are positive real numbers.	False	<p>Consider graph <math>G = (V, E)</math> and capacity function <math>c: E \rightarrow \mathbb{R}^+</math>, where <math>\exists e \in E, c(e) \in \mathbb{P}^+</math> (there is an edge with positive irrational weight).</p> <p>By CLRS 26.2, footnote 2, FF can fail to terminate when given irrational edge capacities.</p> <p>Thus we have found a network with capacities in <math>\mathbb{R}^+</math> (since <math>\mathbb{P}^+ \subset \mathbb{R}^+</math>) that causes FF to run indefinitely.</p>
Undirected graph $G = (V, E)$ is called $k$ -regular if degree of every vertex is exactly $k$ . Girth of the graph is the length of a shortest cycle in $G$ . For example, the simple cycle with 5 vertices is a 2-regular graph of girth 5. We claim that there is no 3-regular graph of girth 5 on 10 vertices.	False	 <p>This graph is 3-regular (each node is connected to 3 other nodes), has 10 nodes and has girth 5.</p>

## 2. Number of Distinct Shortest Paths

### a. English Description of Algorithm

- We will use a modified breadth first search, where we will also maintain (as a node attributes *num\_ways*, *dist\_from\_start*) the number of ways to get through the node, and the distance from the starting node (number of edges)
- We initialize this modified BFS at  $s$ , setting  $s.num\_ways = 1$ ,  $s.dist\_from\_start = 0$ , and  $k.num\_ways = 0$ ,  $k.dist\_from\_start = \infty$  for all other nodes  $k \neq s$
- Every time we encounter a node  $p$  that is currently on the priority queue, we increment  $p.num\_ways$  by  $y.num\_ways$ , where  $y$  is the node that has “discovered”  $p$  (i.e. a node with an edge leading to  $p$ )
- When we finish the BFS, we return  $t.num\_ways$  as the answer

b. Pseudocode

```
def algorithm(graph, s, t):
    # Set num_ways and dist_from_start of all nodes except s to 0
    for node in graph.nodes:
        node.num_ways = 0
    s.num_ways = 1

    # Maintain bit-vectors that keep track of nodes:
    future = [1] * len(graph.nodes) # To be explored
    present = [0] * len(graph.nodes) # Currently getting explored
    past = [0] * len(graph.nodes) # Explored already

    # Helpers for setting the bitvectors appropriately
    def make_future(node):
        future[node], present[node], past[node] = 1, 0, 0
    def make_present(node):
        present[node], future [node], past[node] = 1, 0, 0
    def make_past(node):
        past[node], present[node], future[node] = 1, 0, 0

    # Set s's values in the bitvector appropriately (we're starting from here)
    make_present(s)

    # Maintain a priority queue of nodes for exploration
    queue = PriorityQueueHeap()
    queue.push(s) # s is the first node to explore from

    while len(queue) > 0:
        cur_node = queue.pop()
        adjacent_nodes = graph.adjacent_to(cur_node)
        for nxt_node in adjacent_nodes:
            if present[nxt_node]:
                nxt_node.num_ways += cur_node.num_ways
            elif future[nxt_node]:
                nxt_node.num_ways = cur_node.num_ways
                queue.push(nxt_node)
                make_present(nxt_node)
            make_past(nxt_node)

    return t.num_ways
```

c. Proof of Correctness

- i. We note that the number of ways of going through a node is equal to the sum of the number of ways of going through its parent nodes
- ii. Our algorithm modifies BFS such that:
  1. When we see a completely novel node, we set its *num\_ways* value to the *num\_ways* of its “discoverer” node – since we know that:  
 $|\{s \rightarrow a\}| = x \wedge (a \rightarrow b) \Rightarrow |\{s \rightarrow b\}| \geq x$  (as we can just concatenate  $(a \rightarrow b)$  onto  $(s \rightarrow a)$  to get  $(s \rightarrow b)$ )
  2. When we see a node that has been explored by another node, we just increment its *num\_ways* value by the *num\_ways* value of the current (parent) node. This is because:  
 $\exists (a \rightarrow c), (b \rightarrow c) \Rightarrow \{s \rightarrow c\} = (\{s \rightarrow a\} + (a \rightarrow c)) \cup (\{s \rightarrow b\} + (b \rightarrow c))$   
Thus:  
 $\exists (a \rightarrow c), (b \rightarrow c) \Rightarrow |\{s \rightarrow c\}| = |\{s \rightarrow a\}| + |\{s \rightarrow b\}|$
  3. Since we are correctly modifying the *num\_ways* attribute, the algorithm must be returning the correct result when we query  $t$  for its *num\_ways* value

d. Runtime Justification

- i. This algorithm should have the same runtime as BFS:  $O(|V| + |E|)$  (CLRS 22.2), as the code is very similar, just with the added activity of manipulating a node attribute, which is done in constant time

3. Server Movement Challenge

*Let  $l$  be the length of the given cable (I can't figure out how to do curly  $L$ 's in Word)*

a. English Description of Algorithm

- i. Construct a *complete* (every node has an edge to every other node), weighted graph  $G$  of the following form:
  1. Nodes: electrical outlets
  2. Weights: distance from one electrical outlet to the other
- ii. Find the minimum spanning tree of  $G$ , call it  $m$
- iii. Find the **shortest path in  $m$**  from the starting outlet to the target outlet. Call this  $P$
- iv. Find the longest distance between outlets in  $P$ , call this  $D$ .
- v. Return  $L = D - l$ , which is the length of cable needed to move the server without outages

## b. Pseudocode

```
def dist(pt1, pt2):
    x_diff = pt1.x - pt2.x
    y_diff = pt1.y - pt2.y
    return ((x_diff ** 2) + (y_diff ** 2)) ** 0.5

def build_graph(outlets):

    # Make a graph where the nodes are the outlets
    graph = Graph()
    for outlet in outlets:
        graph.add_node(outlet)

    # Connect the graph completely (each edge weight is the distance between nodes)
    for n1 in graph.nodes:
        for n2 in graph.nodes:
            if n1 != n2 and graph.not_connected(n1, n2): # no self/duplicate edges!
                graph.connect(n1, n2, weight=dist(n1, n2))

    return graph

def dijkstra(graph, source, target):
    return (the shortest path from source to target in list
            form:[source, ..., target])

def mst_as_graph(graph):
    # Use Prim to find the MST of graph
    mst = Prim(graph)

    # Return the MST of graph in graph form
    return Graph(nodes = mst.nodes,
                  edges = mst.edges)

def find_largest_gap(points):
    # Finds the largest gap between two consecutive points in points

    # Setting a high score that can be beaten easily
    high_score = -1 * infinity

    # Look at two nodes at a time (sliding window size 2)
    for [p1, p2] in path_nodes:

        # Set a new high-score if it is beaten
        if dist(p1, p2) > high_score:
            high_score = dist(p1, p2)

    # Return the distance between the two furthest consecutive pts in points
    return high_score

def algorithm(outlets, given_cable_length, src_outlet, dst_outlet):
    graph = build_graph(outlets) # Rep as graph
    mst_graph = mst_as_graph(graph) # Find MST
    short_path = dijkstra(mst_graph, src_outlet, dst_outlet) # Find min path in MST
    largest_gap = find_largest_gap(short_path) # Find longest edge
    return largest_gap - given_cable_length # Return what we want
```

c. Proof of Correctness

i. Proof of Helper Function Correctness

1. *dist()*

- a. This is just a simple calculation and it is trivially correct

2. *build\_graph()*

- a. This algorithm is supposed to convert the list of outlet locations into a connected graph of the following form:
- Nodes: locations (Cartesian coordinates)
  - Weights: distances between locations
- b. We can clearly see that the semantics of the first part clearly add all the locations as nodes in the graph, and the semantics of the second part clearly connect distinct pairs of nodes together with edges with weights equal to the distance between the Cartesian coordinates of the nodes

3. *dijkstra()*

- a. This just returns the results of Dijkstra's algorithm in a list format – CLRS 24.3 proves Dijkstra's algorithm to be correct

4. *mst\_as\_graph()*

- a. This just returns the result of Prim's algorithm in graph format – CLRS 23.2 proves Prim's algorithm to be correct

5. *find\_largest\_gap()*

- a. This is just a simple 2-window, linear search for the pair of consecutive nodes with the largest distance between them – proving the correctness of a linear search is trivial

ii. Proof of Main Algorithm:

1. Definitions / Declarations

- a. Let  $E_X, V_X, c_X$  be the edges, vertices/nodes and cost function of graph  $X$ , respectively
- b. Let  $M(G)$  be the minimum spanning tree of graph  $G$
- c. Let the *mass* of a path  $p$  be the **maximum weight** of an edge in  $p = \phi_1, \dots, \phi_k$ , i.e.  $\text{mass}(p) = \max\{c(\phi_1), \dots, c(\phi_k)\}$
- d. Let  $\mu(a, b, G) \in \{(a \rightsquigarrow b)\}$  be the path of **minimum mass** between nodes  $a, b$  in graph  $G$ .
- e. Let  $m(a, b, G)$  be the **shortest path** between nodes  $a, b$  in  $M(G)$  – note that since  $M(G)$  is a tree,  $m(a, b, G)$  must be unique.
- f. Lemma A:  $\mu(a, b, G) = m(a, b, G)$ . Proof:
- Contained in page 3 of the following document – for maximum spanning trees, but logic can be adapted for minimum spanning trees: <http://suraj.lums.edu.pk/~te/cspf/cspf34.pdf>

2. Proof of Correctness:

- a. The goal of our algorithm is to find the minimum length of cable needed to get the cart from starting outlet  $s$  to destination outlet  $t$ .
- b. In order to find this length of cable, we first need to find a path where the largest gap between nodes is minimized (because this will minimize the amount of cable that we need to “leapfrog” the cart)
- c. By A, we know that the shortest path between nodes in an MST also happens to be the path with the lowest maximum weight – this is what we need to satisfy the above requirement



- d. Since our algorithm correctly finds an MST and then correctly finds a shortest path on that MST, it is also correct in finding the maximum distance between two-nodes on this minimum-mass path.
- e. Since the algorithm is correct in finding the total distance  $D$  of cable needed, it is also correct in returning the amount of cable  $L$  needed to compensate for  $l$  (since  $D = L + l \Rightarrow L = D - l$ )

d. Runtime Justification

- i. Assume we use the following implementations (runtimes sourced from CLRS):

Data Structure / Algorithm	Implementation	Runtime(s)
Graph	Adjacency List (hash-map of nodes to linked lists of edges)	Adding nodes $\in O(n)$ (hash-map insertion worst case) (CLRS 11.2)  Connecting nodes $\in O(1)$ (if we maintain a pointer to the end of each linked list)
Prim's Algorithm	Binary Heap (priority queue) Adjacency List (graph)	$O( E \log  V )$ (CLRS 23.2)
Dijkstra's Algorithm	Binary Heap (priority queue) Adjacency List (graph)	$O( E \log  V )$ (if all nodes are reachable from source, CLRS 24.3)

- ii. These are the non-trivial steps to the algorithm:

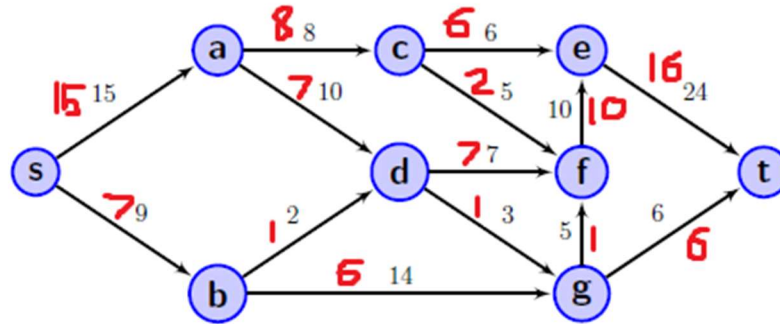
1. Building the graph –  $build\_graph() \in O(n^2)$ 
  - a. First step is to add all the outlets as nodes to the graph (can be done with a simple loop over all outlets, will take  $O(n)$  time for each insertion in the worst case, so  $O(n^2)$  total)
  - b. Second step is to connect all the nodes together according to Euclidean distance (since there are  $n$  many nodes, there are  $\Theta(n^2)$  many pairs of different nodes, thus the double-loop takes  $O(n^2)$  time (if node connection is  $O(1)$  time)
  - c. The dominant term of the above is  $O(n^2)$ , thus this is the upper-bound for this step's runtime
2. Finding the MST of the graph -  $mst\_as\_graph() \in O(n^2 \log n)$ 
  - a. We note that for the initial graph representation of the outlets,  $|V| = n, |E| \in O(n^2)$
  - b. Thus, finding the MST using Prim's algorithm will take  $O(n^2 \log n)$  time
  - c. We also note that the MST has  $n$  nodes and  $n - 1$  edges (by virtue of being a tree with  $n$  nodes)
  - d. Converting this MST to a graph will take  $O(n) * O(n) = O(n^2)$  time to add the vertices and  $O(n)$  time to add the edges
  - e. The dominant term of the above is  $O(n^2 \log n)$
3. Finding the shortest path in the MST to get from the start to the end –  $dijkstra() \in O(n \log n)$ 
  - a. We note that the graph we are running Dijkstra's on has  $|E|, |V| \in O(n)$ , so the runtime of this step is  $O(n \log n)$

4. Finding the longest gap in the shortest path -  $find\_largest\_gap() \in O(n)$ 
  - a. This is just a simple linear search for the longest length between points, with a sliding window of size 2
  - b. With a window size of 2, this step will take  $(n - 1) \in O(n)$  iterations
- iii. We note that the dominant term out of all of the above steps is  $O(n^2 \log n)$ , and this is the upper-bound for the runtime of our algorithm

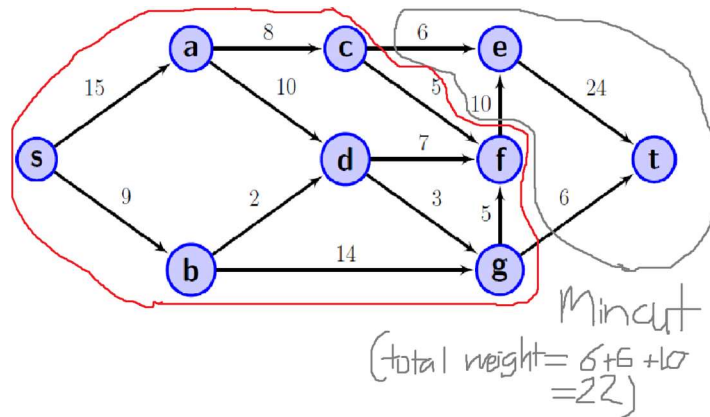
#### 4. Network Flow

##### a. Computations

- i. Max Flow: 22 (16 + 6 or 15+7)

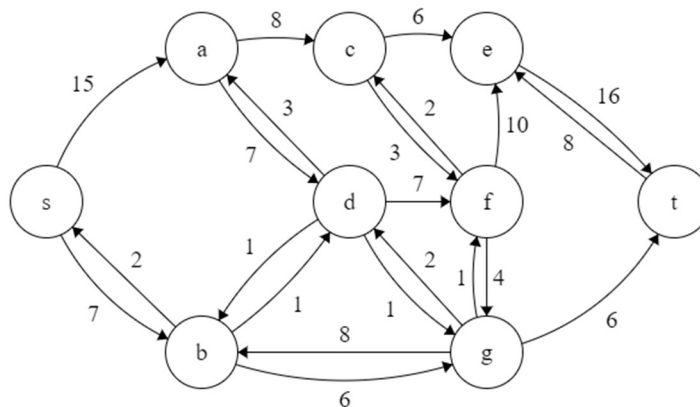


- ii. Min Cut

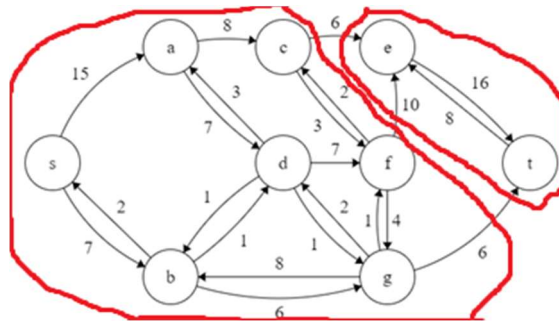


##### b. Residual Graph

- i. Diagram



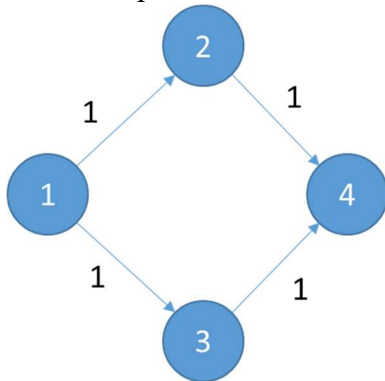
ii. Min Cut



c. List of Constrictors

- i.  $\{(c \rightarrow e), (f \rightarrow e), (g \rightarrow t)\}$

d. Small Graph with No Constrictors



e. Constrictor-Finding Algorithm

i. English Description:

1. We are going to find the edges that belong on the min-cut – these are constrictors
2. Construct the residual graph using Edmonds-Karp, a variant of FF
3. Find all nodes that can be reached from the source in the residual graph using DFS, mark the rest of the nodes as unreachable
4. In the original flow network, look at the edges leaving the reachable set of nodes – filter these by the edges that connect reachable nodes to unreachable nodes
5. Return the set of edges that link the nodes marked as reachable and unreachable together

ii. Proof of Correctness

1. Note that edge  $e = (u \rightarrow v)$  is a constrictor  $\Leftrightarrow$  increasing  $c(e)$  results in the creation of a new  $(s \rightsquigarrow t) \in G_{\text{residual}}$
2. If increasing  $c(e)$  results in the creation of a new  $(s \rightsquigarrow t) \in G_{\text{residual}}$ , then more flow can be pushed along this path to increase max-flow, which means that  $e$  must be a constrictor – if this increase in capacity does not result in the creation of a new augmenting path, then  $e$  is not a constrictor
3. Thus, the edges that cross between the reachable and unreachable sets of nodes are constrictors – since the max-flow of a network is equal to its minimum cut, increasing the capacity of any of the edges on the minimum cut must also increase the max-flow

iii. Runtime Justification

1. Step Analysis:

- a. We know that Edmonds-Karp runs in time  $O(|V||E|^2)$  (CLRS 26.2)

- b. Finding the set of reachable and unreachable nodes can be done using DFS and looping (runtime  $O(|V| + |E|)$ )
    - c. Finding the set of edges that link the reachable and unreachable node sets together takes  $O(E)$  time (to loop over all the edges)
  - 2. The dominant term of the above is  $O(|V||E|^2)$ , which means it is the upper-bound for the running time of this algorithm
- 5. Algorithm to Update Max-Flow on Capacity Changes
  - a. English Description of Algorithm
    - i. Look at the residual graph of the flow network and the flow network itself
    - ii. Create a new graph with the original nodes with the same edge directions as the original flow network
    - iii. We mark unsaturated edges with weight 0 (as there is spare room to push flow through), and saturated edges with weight  $p(e)$  (as we need to upgrade these to push flow through), where  $e$  is the current edge.
    - iv. We then find the minimum-weight path needed to get from the source to the terminal nodes – if the total weight is 0, we return the empty list – we don't need to upgrade any edges
    - v. Otherwise, this shortest-path will be the collection of edges that we need to upgrade to increase our max-flow

## b. Pseudocode

```
def make_upgrade_cost_graph(flow_graph, original_network):
    # original_network is the graph of the flow network
    # flow_graph is the "implementation" of the max_flow

    # Graph to be returned at the end of the algorithm
    ret_graph = DirectedGraph()

    # Add the nodes to the graph
    for node in original_network:
        ret_graph.add_node(node)

    # Assume that edges are identified identically between the two inputs,
    # and that weights in flow_graph do not exceed the weights of original_network
    for edge in original_network.edges:

        # If the current edge is saturated, set weight of this edge to p(e)
        if flow_graph.weight(edge) == original_network.weight(edge):
            ret_graph.connect_with_weight(edge.source, edge.end, p(edge))

        # Otherwise set the weight of this edge to 0
        else:
            ret_graph.connect_with_weight(edge.source, edge.end, 0)

    # Return the upgrade-cost graph
    return ret_graph

def dijkstra(graph, source, target):
    return (the shortest path from source to target in list
            form:[source, ..., target])

def consecutive_nodes_to_edges(nodes):
    ret_lst = []
    for [n1, n2] in nodes:
        ret_lst += WeightedDirectedEdge(n1, n2, weight=p(n1, n2))
    return ret_lst

def algorithm(flow_graph, original_network):
    # Make the graph of upgrade costs
    upgrade_graph = make_upgrade_cost_graph(flow_graph, original_network)

    # Find the shortest path along the upgrade-cost graph
    cheapest_path_to_upgrade_nodes = dijkstra(upgrade_graph,
                                                flow_graph.source,
                                                flow_graph.terminal)

    cheapest_path_to_upgrade =
        consecutive_nodes_to_edges(cheapest_path_to_upgrade_nodes)

    # Return the empty list if we don't have to upgrade anything
    if sum(cheapest_path_to_upgrade, key=lambda(x)->x.weight) == 0:
        return []

    # Otherwise return the shortest path as the collection of edges to upgrade
    else:
        return cheapest_path_to_upgrade
```

c. Proof of Correctness

**I DO NOT KNOW HOW TO ANSWER THIS QUESTION**

d. Runtime Justification

Data Structure / Algorithm	Implementation	Runtime(s)
Graph	Adjacency List (hash-map of nodes to linked lists of edges)	Adding nodes $\in O(n)$ (hash-map insertion worst case) (CLRS 11.2)  Connecting nodes $\in O(1)$ (if we maintain a pointer to the end of each linked list)
Prim's Algorithm	Binary Heap (priority queue) Adjacency List (graph)	$O( E \log  V )$ (CLRS 23.2)
Dijkstra's Algorithm	Binary Heap (priority queue) Adjacency List (graph)	$O( E \log  V )$ (if all nodes are reachable from source, CLRS 24.3)

i. Steps of the Algorithm:

1. First step is to create the upgrade cost graph:

- $make\_upgrade\_cost\_graph() \in O(|V|^2 + |E|)$  (if hashing is done stupidly)  
 $make\_upgrade\_cost\_graph() \in O(|V| + |E|)$  (if hashing is done competently)
- Adding nodes to the graph takes  $O(V) * O(V) = O(V^2)$  time in the worst case if hashing is done stupidly.
- If the hashing is competent, this takes  $O(1) * O(V) = O(V)$  time to do
- Adding edges to the graph takes  $O(1) * O(E) = O(E)$  time

2. Second step is to find the shortest path along the upgrade-cost graph:  $dijkstra() \in O(|E|\log|V|)$

3. Third step is to convert the shortest path from node form into edge from:  $consecutive\_nodes\_to\_edges \in O(|V|)$

- This is just a single sliding window, will obviously run in time proportional to the input size

4. Fourth step is to check whether the shortest path has total weight of 0:  $sum() \in O(|V|)$

- Summing a collection of numbers using a for-loop will take time proportional to the size of the collection

ii. Total Runtime (dominant term analysis):

- Dumb hashing:  $O(|V|^2 + |E| + |E|\log|V| + |V|) \in O(|V|^2 + |E|\log|V|)$
- Normal hashing:  $O(|V| + |E| + |E|\log|V| + |V|) \in O(|V| + |E|\log|V|)$