CSC 226 SUMMER 2023
ALGORITHMS AND DATA STRUCTURES II
ASSIGNMENT 4
UNIVERSITY OF VICTORIA

1. Show how to compute the shortest paths between all pairs of vertices in the graph attached using the Floyd-Warshall algorithm. It is enough to give the D matrix at the end of each iteration.

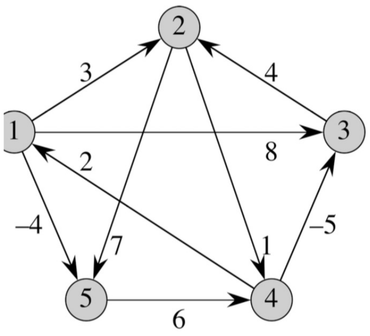
2. Show how to use the Edmunds-Karp algorithm to find a max flow on the graph in slide 3 of Lecture 18 ~~19~~ slides. Show for each step of the algorithm: the residual graph, the augmenting path chosen, and the additional flow.

3. Given a flow f for a graph G, give an algorithm for proving that the flow is maximal which runs in time $O(m)$. The edges are stored in adjacency list form, and with each edge is its capacity and flow across the edge. Give a running time analysis of your algorithm.

4. The edge-connectivity of an undirected graph is the minimum number k of edges that must be removed to disconnect the graph. i.e., 1 for a tree, 2 for a cycle. Give an algorithm for determining the edge connectivity of an undirected graph $G = (V, E)$ which runs a max flow algorithm on $|V| - 1$ different flow networks, each with $|V|$ nodes and $|E|$ edges. What is its running time?

1. Show how to compute the shortest paths between all pairs of vertices in the graph attached using the Floyd-Warshall algorithm. It is enough to give the D matrix at the end of each iteration.

Example: APSP Floyd Warshall



All vertices	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

K=1	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

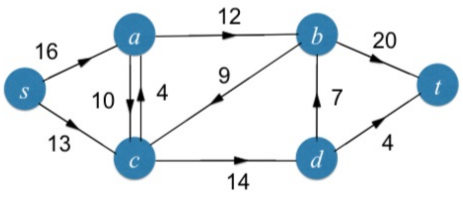
$k=2$	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

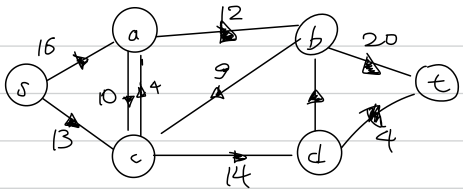
$k=3$	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	-1	-5	0	-2
5	∞	∞	∞	6	0

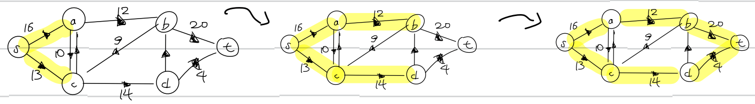
$k=4$	1	2	3	4	5
1	0	3	-1	4	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

$k=5$	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

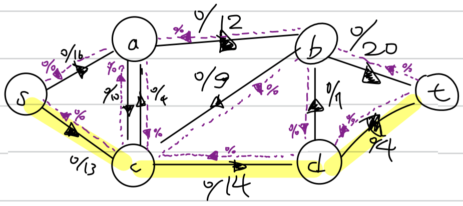
2. Show how to use the Edmunds-Karp algorithm to find a max flow on the graph in slide 3 of Lecture 19 slides. Show for each step of the algorithm: the residual graph, the augmenting path chosen, and the additional flow.







(Now get the argumenting path from s → t

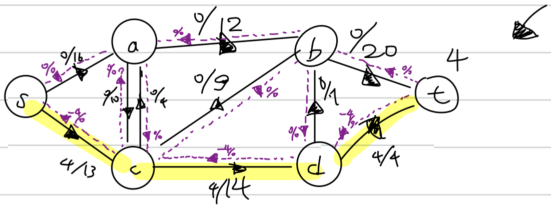


· Find the bottle neck value

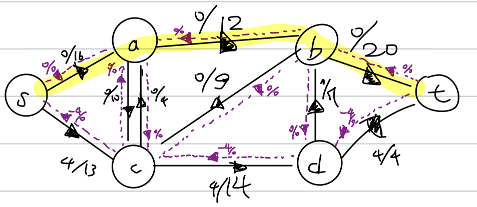
min (13−0, 14−0, 4−0) = 4

↓

Update the flow along the paths by the bottleneck value and repeats until no more argumenting paths can be found.



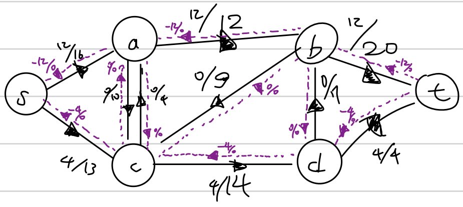
Now, Find the another argumenting path



The bottleneck value is :
$\min(16-0, 12-0, 20-0) = 12$

↓

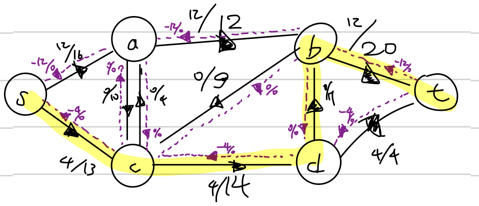
Update the flow along the paths by the bottleneck value and repeats until no more argumenting paths can be found.

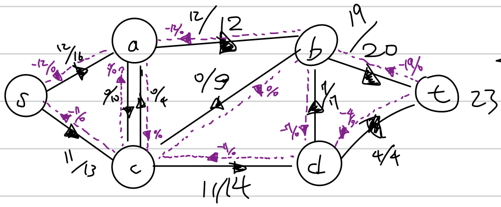


The bottleneck value is :
$\min(13-4, 14-4, 7-0, 20-12) = 7$

↓

Update the flow along the paths by the bottleneck value and repeats until no more argumenting paths can be found.





↓

The maxflow of this graph is 23

3. Given a flow f for a graph G, give an algorithm for proving that the flow is maximal which runs in time $O(m)$. The edges are stored in adjacency list form, and with each edge is its capacity and flow across the edge. Give a running time analysis of your algorithm.

To prove that a given flow f for a graph G is maximal, I will use the concept of residual graphs.

· First, I initialize a residual graph R as a copy of the original graph G.

Secondly, I perform BFS or DFS on the residual graph R which starts from the source node.

· Thirdly, during the traversal, I only consider edges with positive remaining capacity.

· Fourthly, If a path is found from the source to the sink in the residual graph. This means there exists an augmenting path in the original graph G which implies the current flow isn't maximal. So I need to return false.

· Finally, If no augmenting path is found, the flow is maximal and return true.

The running time of this algorithm is denominated by the BFS or DFS which is $O(m)$ where m is the number of edges in the graph. The other steps can be done in $O(m)$ time.

∴ The overall runtime of the algorithm which the flow is maximal is $O(m)$

4. The edge-connectivity of an undirected graph is the minimum number k of edges that must be removed to disconnect the graph. i.e., 1 for a tree, 2 for a cycle. Give an algorithm for determining the edge connectivity of an undirected graph $G = (V, E)$ which runs a max flow algorithm on $|V| - 1$ different flow networks, each with $|V|$ nodes and $|E|$ edges. What is its running time?

1. I construct the directed graph G^* from G by replacing each edge (u,v) in G with two directed edges (u,v) and (v,u) in G^*

2. I let $f^*(u,v)$ be the maximum flow value from u to v through G^* with all edges capacity is 1.

3. I pick an arbitrary node u and compute $f^*(u,v)$ for all $v = u$

4. Now, the algorithm says that edge connectivity equals $c^* = \min\{f^*(u,v)\}$ for all $v \neq u$.

5. Therefore the edge connectivity of G can be computed by running the max-flow algorithm $|V|-1$ times on flow networks, each having $|V|$ vertices and $2|E|$ edges

6. Now I suppose k is the edge connectivity of the graph and "S" is the set of k edges such that removal of S will disconnect the graph into two non-empty subgraphs G_1 and G_2. Let's assume $u \in G_1$, and w be a node in G_2.

7. Since $u = w$, the value $f^*(u,w)$ will be computed by the algorithm
By the max-flow min-cut theorem, $f^*(u,v)$ equals the min-cut size between the pair (u,v), which is at most k since S disconnects u and w. Therefore, we have $C^* \leq f^*(u,w) \leq k$

8 But c^* cannot be smaller than k, as that would imply a cut set of size smaller than k. This is the contradiction of the fact that k is the edge connectivity.

9. Therefore, $C^* = k$ and the algorithm returns the edge connectivity of the graph.

The running time is $O(|E|) + O(|V| \times |E|) = O(|V|^2 \times |E|)$

CSC 226 Assignment #4

1) Orange Highlighting Indicates Changed Values

k = 0	**1**	**2**	**3**	**4**	**5**
1	0	3	8	+∞	-4
2	+∞	0	+∞	1	7
3	+∞	4	0	+∞	+∞
4	2	+∞	-5	0	+∞
5	+∞	+∞	+∞	6	0

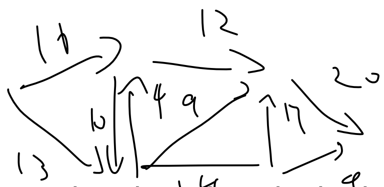
k = 1	**1**	**2**	**3**	**4**	**5**
1	0	3	8	+∞	-4
2	+∞	0	+∞	1	7
3	+∞	4	0	+∞	+∞
4	2	5	-5	0	-2
5	+∞	+∞	+∞	6	0

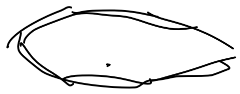
k = 2	**1**	**2**	**3**	**4**	**5**
1	0	3	8	4	-4
2	+∞	0	+∞	1	7
3	+∞	4	0	5	11
4	2	5	-5	0	-2
5	+∞	+∞	+∞	6	0

k = 3	**1**	**2**	**3**	**4**	**5**
1	0	3	8	4	-4
2	+∞	0	+∞	1	7
3	+∞	4	0	5	11
4	2	-1	-5	0	-2
5	+∞	+∞	+∞	6	0

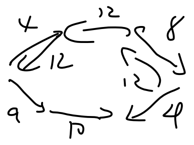
k = 4	**1**	**2**	**3**	**4**	**5**
1	0	3	-1	4	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

k = 5 (Final)	**1**	**2**	**3**	**4**	**5**
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0





$\boxed{12}$

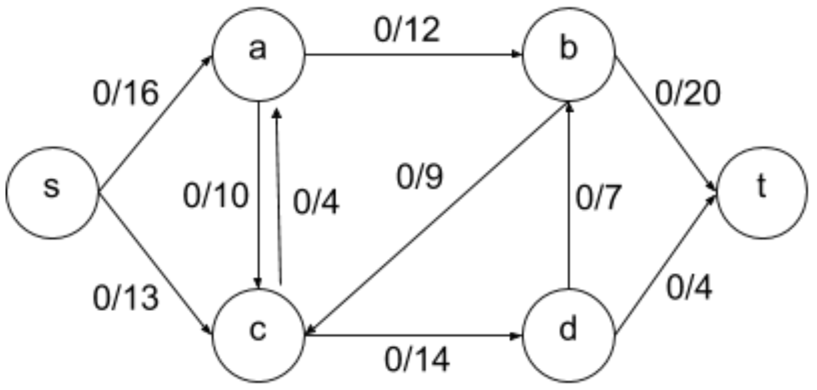


$\boxed{12+4}$

16

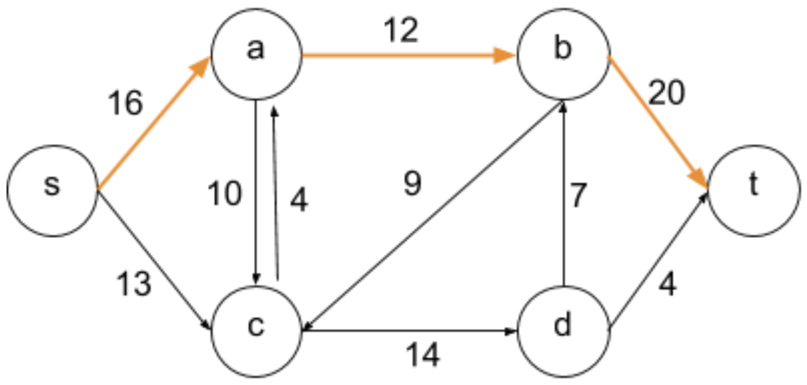
2) Please note that the edges in the initialization and final graphs are written in the form of flow/capacity. Each edge in the residual graphs only display a value for flow.

Initialization:



Total Flow: 0

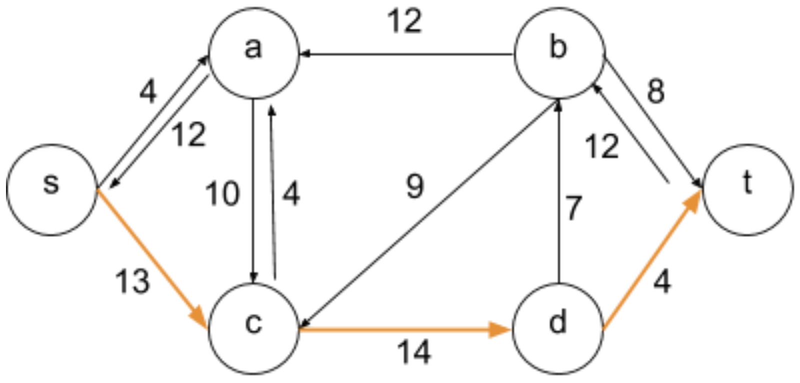
Residual Graph 1:



Additional Flow: +12
Total Flow: 12

-> Indicates Path Taken

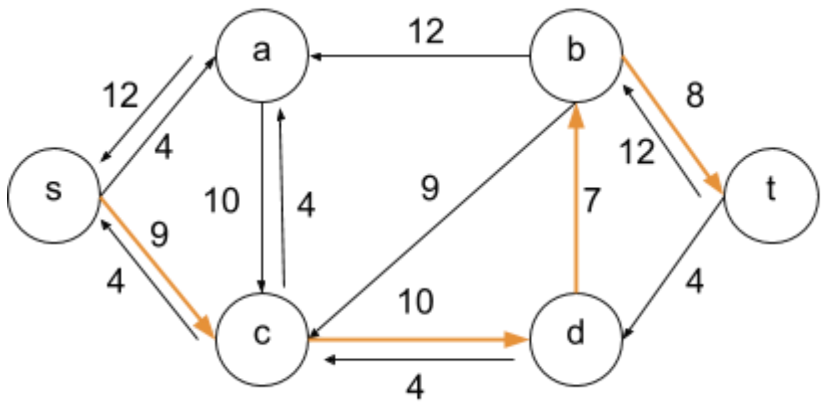
Residual Graph 2:



Additional Flow: +4
Total Flow: 16

-> Indicates Path Taken

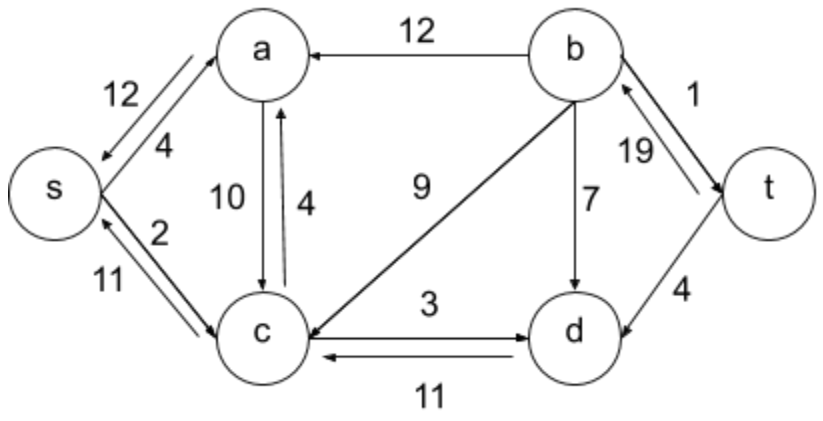
Residual Graph 3:



Additional Flow: +7
Total Flow: 23

-> Indicates Path Taken

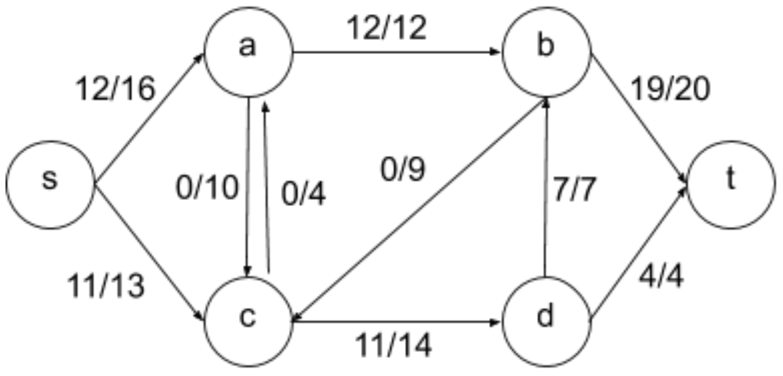
Residual Graph 4:



Total Flow: 23

As there is no augmenting path in the last residual graph, the sink vertex t is not reachable from the source s, we can conclude that a maxflow of **23** has been found.

Final Graph:



Total Flow: 23

3) In order to prove that the provided flow f for a graph G is maximal, we can apply the following algorithm:

1. Initialize an empty adjacency list data structure that will contain a total of n linked lists, one for each vertex in the graph G.

2. Iterate through the existing adjacency list and convert every edge (u,v), into the following format. As we are working with a directed graph, we don't need to worry about repeated edges within the adjacency list:

    2.1. Create a directed edge from the vertices u to v, weighted by taking the edge's capacity and subtracting the current flow amount (c(u,v) - f(u,v))

    2.2. Create a directed edge from the vertices v to u, setting the edge weight to the current flow amount, f(u,v)

3. Run a BFS traversal on the residual graph's adjacency list representation and check to see if a path from source s to sink t exists, referred to as an augmenting path. If an augmenting path is found, we can prove that the given flow, f is not a maxflow, using property C of the Maxflow-Mincut Theorem ('Lecture20.pdf', slide #7). Property C states that the validated claim, there is no augmenting path, is equivalent to property B, that f is a maxflow. Thus, based on the presence of an augmenting path, we can say f is not a maxflow and return false if one such path exists, and true if one does not.

Time Complexity:

- Creating an empty adjacency list data structure will take **O(n)** time, producing one unfilled linked list for each vertex in graph G

- Converting every edge to its appropriate representation in the residual graph will take **O(m)** time. This is because we operate once for each edge, creating at most 2 edges each time.

- A BFS traversal on an adjacency list data structure will take **O(n + m)** time. Although we are operating on a maximum of 2m edges, the 2 is absorbed into the big-Oh notation.

- It will take a constant amount of time, **O(1)** to identify if there is a path from s to t. This is combined into the BFS algorithm, just requiring a check to see if the current node being visited is in fact the sink, t.

Thus the total time complexity is:

⇔ O(n + m + n + m + 1)
⇔ O((m-1) + m + (m-1) + m + 1) Applying the property that the number of edges in a connected graph is ≥ n-1
⇔ O(4m - 1)

Resulting in a final value of **O(m).**

4) To compute the minimum number of edges that must be deleted in order to disconnect the graph G into two components, we can run the following algorithm:

1. First, we need to create a new directed graph from the undirected graph, in order to run the Edmunds & Karp maxflow algorithm. To do this, we create an empty adjacency list data structure with a linked list initialized for each vertice in G. We need to create two directed edges in the new graph for each, previous, undirected edge, one in the forward direction and one in the backward direction.

2. We can apply property #4 of a feasible st-flow network from 'Lecture20.pdf', slide #6 to determine a suitable capacity for each edge in the graph. This property states that taking an st-cut, X, of a feasible st-flow such that the flow f is equal to the capacity of X, implies that f is a maxflow and that X is a mincut. In order to fully leverage this, **the edge weights should all be set to 1**, allowing for **the capacity of the mincut and hence the maxflow value to be equal to the number of edges for removal.**

3. Choose an arbitrary vertex and fix it as the source, s. Choose the sink, t, from the remaining n-1 vertices. Run the Edmunds & Karp maxflow algorithm 1 time, in order to find the initial mincut value. Keep track of this value in a variable called edges.

4. Run the Edmunds & Karp maxflow algorithm a total of n-2 times after the first, choosing the sink from one of the n-2 remaining vertices. Once the sink has been chosen, remove that vertex from the remaining possible choices. After each round, compare the mincut value to the edges variable, and if it is lower than the current value of the variable, set the variable equal to the new mincut.

5. Finally, return the value contained in the edges variable. As stated above, this represents the capacity of the mincut, and since all edges have capacity 1, it also represents the minimum number of edges that must be removed in order to disconnect the graph into two components.

Time Complexity:

- Creating a new, directed representation of the graph G will take **O(n + m)** time. This is because we first need to create n empty linked lists, and then for each existing edge, perform a constant amount of work creating two directed edges. Setting an initial capacity of 1 to each directed edge is also included in the constant time spent on each undirected edge.

- Running the Edmunds & Karp maxflow algorithm a total of n - 1 times, once for each possible sink vertex, will take $O((n - 1)nm^2)$. This is a result of taking the runtime for one round of the Edmunds & Karp algorithm, $O(nm^2)$ and multiplying it by the n - 1 rounds. The constant time operation of comparing the mincut value to the edges variable is absorbed by the big-Oh notation. Further simplifying this results in $O(n^2m^2 - nm^2)$ which is equivalent to $\mathbf{O(n^2m^2)}$.

- Returning the value contained in the edges variable takes a constant amount of time, **O(1)**

Thus, the total time complexity is:

$O(n + m + n^2m^2 + 1)$
which simplifies into the final time of $\mathbf{O(n^2m^2)}$**.**