CPSC/ECON 365: Algorithms Yale University

Solution to Problem Set 4

Student Name: Anton Melnychuk Due: Monday, Nov 6, 2023 at 2:30 pm ET

1 Your Information

(a) Your name.

Anton Melnychuk

(b) Your SID.

24787491

(c) A list your collaborators and any outside resources you consulted for this problem set.

ULAs: Christian, Ryan.

(d) Copy: "I have followed the academic integrity and collaboration policy as written above."

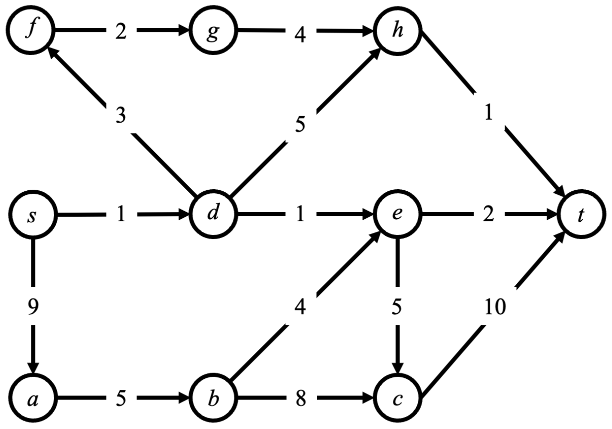
I have followed the academic integrity and collaboration policy as written above.

(e) How many hours did you spend in this problem set?

14 hours

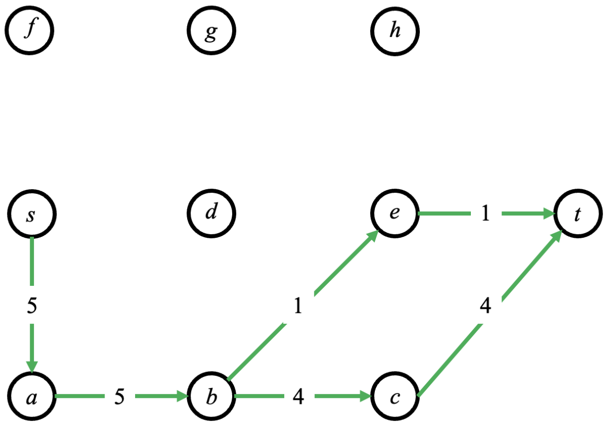
2 Max Flow Practice

Consider the following graph $G = (V, E)$ on $n = 10$ vertices and $m = 14$ edges, with source and sink vertices labeled $s, t \in V$. The capacity of each edge is written in the middle of the edge.



(a) Graph $G = (V, E)$

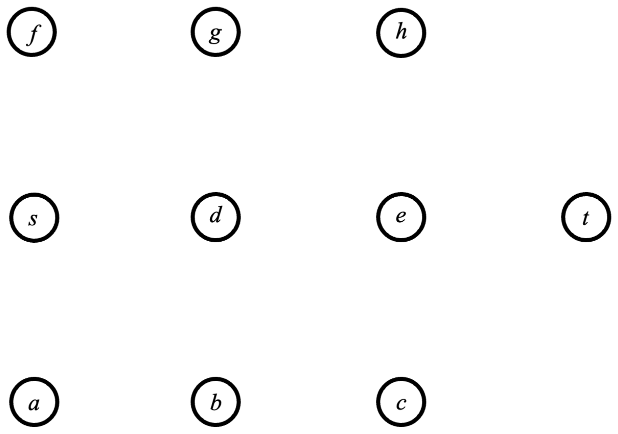
Consider a flow F on G which is drawn in green arrows below. All other edges (not drawn) have zero flow. Note the value of this flow is $5 = 1 + 4$.



(b) Flow F

Problems:

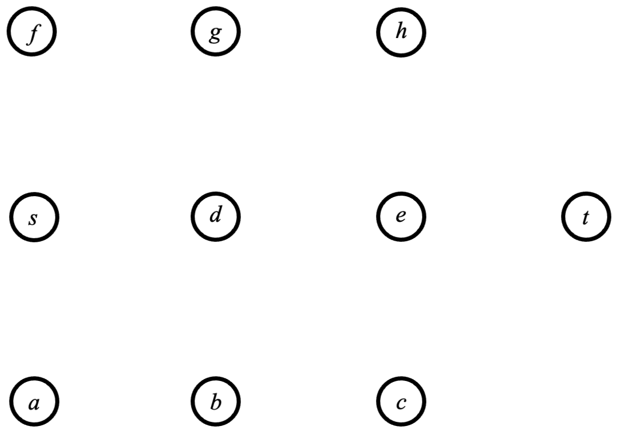
1. Construct the augmenting graph G_F. (Draw it on the graph below, or specify the edges.)



(c) Augmented graph G_F

2. Find an augmented path P in G_F (draw it on the graph above, or specify it as a list of edges). Report the value of its bottleneck β. Is the augmented path unique?

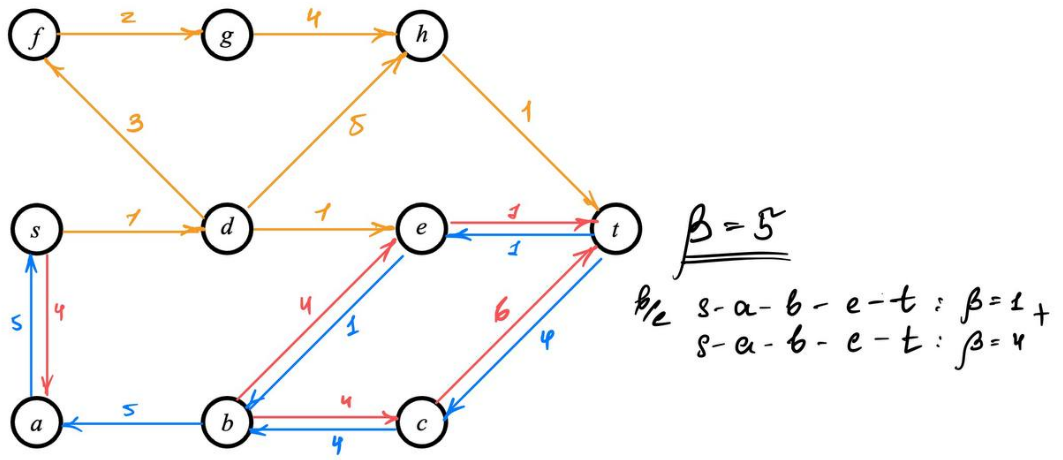
3. Augment the flow using the augmented path you find above to get an augmented flow F' (draw it on the graph below). Report the value of F', and verify it is equal to the sum of the value of the flow F and the bottleneck.



(d) Augmented graph F'

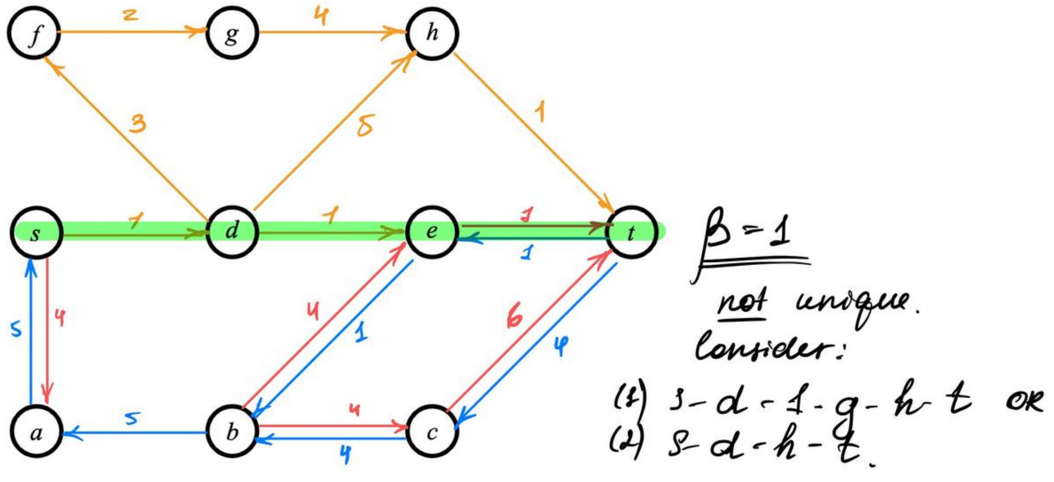
4. Is F' a maximum $s-t$ flow? If yes, then give a certificate (a proof) of optimality. If not, then find the maximum $s-t$ flow.

Problem 1.



(e) Augmenting graph G_F

Problem 2.



(f) Augmented path $P \in G_F$

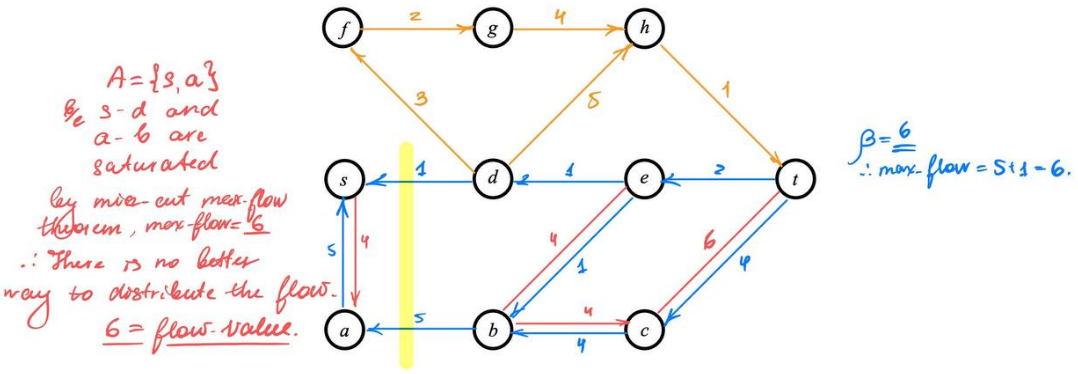
Bottleneck is 1. The augmenting path is not unique. Consider other ways to go:

$$s - d - f - g - h - t$$
$$s - d - h - t$$

.

Problem 3.

Please consider the augmented path highlighted in blue on t-e-d-s vertices. The augmenting flow F' is equal to 6. Since the max flow of the path is determined by bottleneck: $f' = f +$ bottleneck capacity (by def) of augmenting flow, and the previous flow $f = 1+4 = 5$ at $s-a-b-e-t$ and $s-a-b-c-t$ paths, then new $f' = 5 + 1 = 6$, since $min\{1, 2, 3\} = 1$.



f
g
h
s
d
e
t
a
b
c
2
4
3
5
1
1
2
1
2
5
4
4
1
6
4
5
4
4
A={s,a}
b/c s-d and a-b are saturated
by min-cut max-flow theorem, max-flow = 6
∴ There is no better way to distribute the flow.
6 = flow-value.
β = 6
∴ max-flow = 5+1 = 6.

(g) Augmented path $P \in G_F$

Problem 4.

Consider the cut highlighted above. Since by cutting both the $s - d$ and $a - b$ saturated edges in the residual graph, we achieve $k = \text{flow}_{\text{max}}$ (by the Min-Cut Max-Flow Theorem), and we also know that the maximum flow is bounded by $k = 6$, we can conclude that we have reached the optimal distribution and no other augmenting path exists.

3 Water Distribution Problem

In a remote region affected by severe drought, there are n water reservoirs $W_1, W_2, \ldots, W_n$ that supply water to m towns $T_1, T_2, \ldots, T_m$. However, not all reservoirs can supply water to all towns; we are given a list of which towns each reservoir can supply.

Suppose each reservoir W_i has a **supply** of $s_i \in \mathbb{N}$ of water (e.g. in million cubic meters). Suppose each town T_j has a **demand** of $d_j \in \mathbb{N}$ of water (in the same unit). Our goal is to determine whether it is possible to distribute the water from the reservoirs $W_1, \ldots, W_n$ to the towns $T_1, \ldots, T_m$ to satisfy the demand subject to the supply constraint.

Example $n = 2$. Consider for example $n = m = 2$. The supply capabilities are as follows:

Reservoir/Town	T_1	T_2
W_1	✓	✓
W_2	✗	✓

Where ✓ on the (T_i, W_j) entry means reservoir W_j can supply water to town T_i, and ✗ denotes that it cannot. The supply in each reservoir and the demand in each town is as follows.

Reservoir	**Supply** s
W_1	10
W_2	8

Town	**Demand** d
T_1	8
T_2	10

In this case, there is a feasible distribution, given as follows:

Distribution	W_1	W_2
T_1	8	0
T_2	2	8

where the number on the (T_i, W_j) entry denotes the amount of water W_j sends to T_i.

Example $n = 4$. Consider an instance of the problem with $n = m = 4$ as follows.

Reservoir/Town	T_1	T_2	T_3	T_4
W_1	✓	✓	✓	✓
W_2	✗	✓	✗	✓
W_3	✗	✗	✓	✓
W_4	✗	✗	✗	✓

The amount of water available at each reservoir and the demand of each town is as follows.

Reservoir	**Supply** s
W_1	50
W_2	36
W_3	11
W_4	8

Town	**Demand** d
T_1	45
T_2	42
T_3	8
T_4	3

1. **Problem 1:** Is there a feasible distribution in the $n = 4$ example above? If yes, please write down the distribution. If not, please explain why not (or give a certificate of infeasibility).

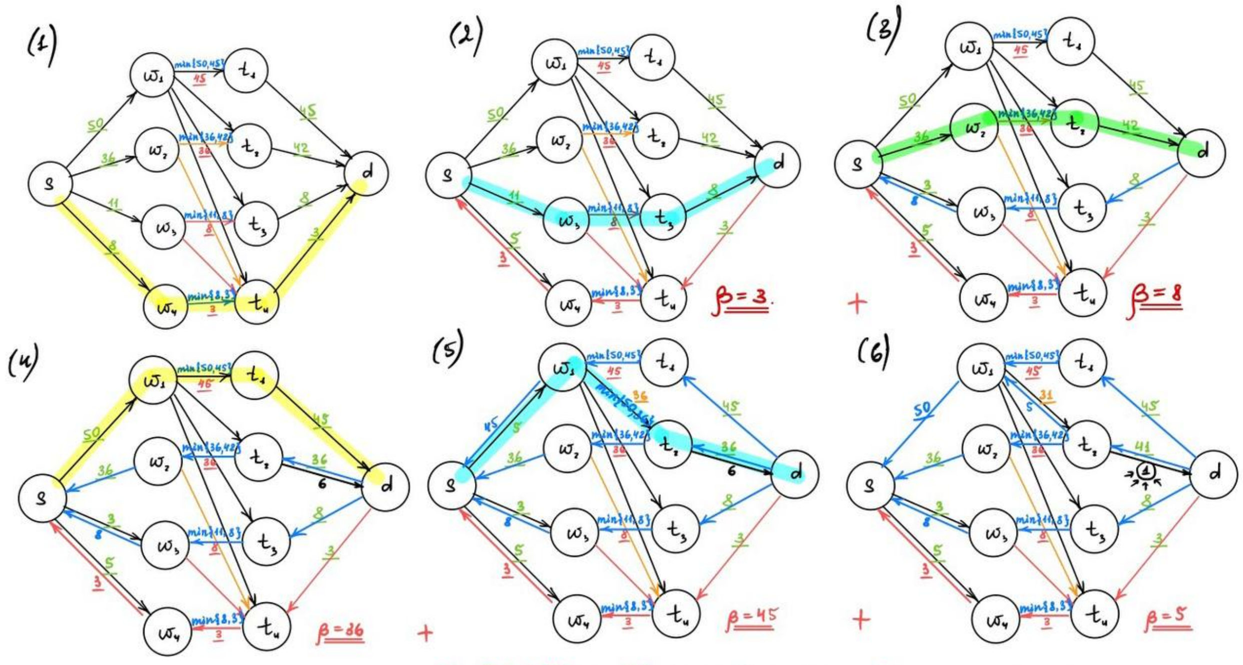
General case. Now consider the general case of n reservoirs and m towns. Suppose for each reservoir W_i and town T_j we know whether W_i can supply water to T_j, which we write as the variable.

$$C_{ij} = \begin{cases} 1 & \text{if } W_i \text{ can supply } T_j \\ 0 & \text{else} \end{cases}$$

for each $i \in \{1, \dots, n\}$ and $j \in \{1, \dots, m\}$. Let $C \in \{0,1\}^{n \times m}$ be a matrix with entries C_{ij}. Suppose the supply of reservoir W_i is $s_i \in \mathbb{N}$, and the demand of town T_j is $d_j \in \mathbb{N}$.

2. **Problem 2:** Design an algorithm to solve the Water Distribution problem above. Your algorithm should take in $s \in \mathbb{N}^n$, $d \in \mathbb{N}^m$, $C \in \{0,1\}^{n \times m}$, and outputs `True` along with a feasible water distribution if one exists, and it outputs `False` otherwise. Prove the correctness of your algorithm, and analyze its running time.

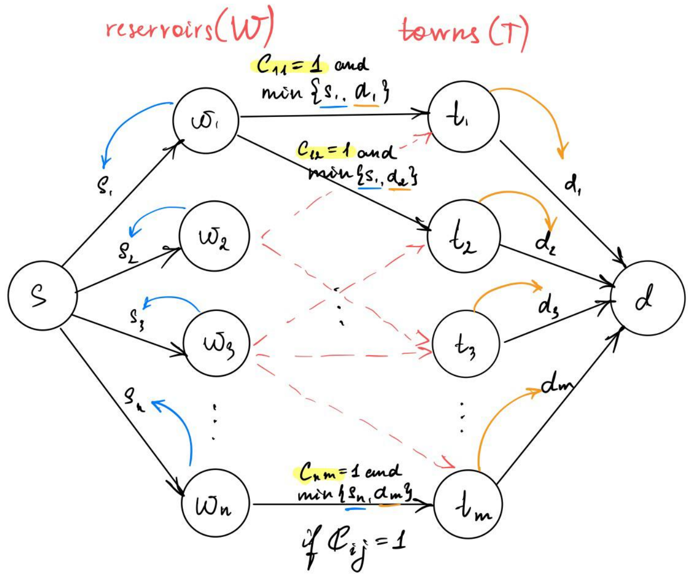
Solution. No, it's <u>not</u> feasible to distribute the water in such example. Consider the possible graph of flow build by applying the algorithm described below. Notice that the sum of outgoing edges at the second town has <u>not</u> reached the demand requested by 1 unit (6). By proof described below in the part 2 of this problem, we conclude that there is no better way to distribute the water between reservoirs and towns than 97 milliom cubic meters in total.



Max Total Flow = 45 + 36 + 8 + 5 + 3 = 97 < Total Demand = 45 + 42 + 8 + 3 = 98

We also know that the algorithm halted by partitioning the final residual graph in two sets divided by the cut going through $s - w_1$, $w_2 - t_2$, $w_3 - t_3$, and $w_4 - t_4$ edges (only incoming for the residual). Since after (6)th iteration of the algorithm, their capacity is fulfilled (only backward edges) and capacities of all edges are non-negative integers, there may not exist greater amount of water that could go through that minimum total capacity by the Min-Cut Max-Flow Theorem explained and proofed during the class. Moreover, the total capacity going through this min-cut is $50 + 36 + 8 + 3 = 97$, which is equal to the values our algorithm has found, so $k = flow_{max}$ (upper bound of the flow).

Algorithm. Let's represent this problem in the flow network. In particular, let's recreate a flow network with a source vertex (s), a sink vertex (d) (notation), and intermediate vertices for each reservoir $(w1, w_2, ..., w_n) \in W$ and each town $(t_1, t_2, ..., t_m) \in T$. Then, connect the source vertex (s) to each reservoir vertex (w_i) with edges whose capacities are equal to the supply of the respective reservoir (represent the maximum amount of water that can be supplied). Second, connect each town vertex (t_j) to the sink vertex (d) with edges whose capacities are equal to the demand of the respective town (represent the amount of water each town needs). Now, for each pair (w_i, t_j) where $C_{ij} = 1$, connect vertex w_i to vertex t_j with an edge whose capacity is $min\{\text{remaining capacity } w_i, \text{demand of } t_j\}$ (ensures that water can only flow from reservoirs to towns that have a supply relationship). Finally, when building edges, set their direction always from left side to right (directed) as visualized here:



(h) Built Graph

- Run the Ford-Fulkerson algorithm on the flow network to find the maximum flow from the source (s) to the sink (d). This represents the maximum possible distribution of water.
- Check if the maximum flow is equal to the sum of demands of all towns. If it is, a feasible $\implies$ return true. If not $\implies$ return false.
- If a feasible distribution is possible, analyze the residual graph to determine the actual distribution of water to each town. Run though each town t and return the sum of all flow-out (reversed) edge capacity values in between the reservoirs (w_i) and towns (t_j) in the residual graph (automatically created during the Ford-Fulkerson execution).

Proof. Consider three distinct parts of the problem:

- If a valid distribution exists, then my algorithm returns $flow_{max} = d_1 + d_2 + d_3 + ... + d_j$ requested (in other words, returns true).
- The sum of all outgoing edges for each town t_k meets the requested demand $d_1, d_2, d_3, ..., d_j$, for $\forall k \in [1, j]$ respectively (means, the distribution will be correct).
- If the distribution is impossible, my algorithm returns maximum flow less than $d_1 + d_2 + d_3 + ... + d_j$ (consequently, returns false).

1.1. If a valid distribution exists, then my algorithm returns $flow_{max} = d_1 + d_2 + d_3 + ... + d_j$ requested.

By contradiction, let's assume the opposite, i.e., there exists a valid distribution, but the algorithm returns a maximum flow (F) that is not equal to $d_1 + d_2 + d_3 + ... + d_j$, $k \in (1, j)$. We know that if the valid distribution exists, it is possible to distribute water from the reservoirs to the towns in the way that satisfies all town demands while respecting supply constraints (from the setup of the algorithm: $min\{$remaining capacity w_i, demand of $t_j\}$). In particular, if $demand > supply$, we can use at most supply $\implies$ capacity. Otherwise, use demand to keep the minimum amount needed to satisfy the problem statement. Also, consider that the reservoir input/ town output edges are limited by their supply and demand capacities by byjection to edges, which states a valid to statement flow in the graph. Now, at that point, if the maximum flow F is not equal to $d_1 + d_2 + d_3 + ... + d_j$, it implies that either F is greater or less:

1. $max_{flow} > d_1 + d_2 + d_3 + ... + d_j$: If the maximum flow F is greater than the sum of town demands, it means that more water is being distributed than the total demand at least at one d_j. This contradicts the fact that there is a valid distribution that respects demand capacity constraints setup by the algorithm (in capacity of edges that connect sink and towns).
2. $max_{flow} < d_1 + d_2 + d_3 + ... + d_j$: (consider the next part as the explanation of this proof).

1.2. The sum of all outgoing edges for each town t_k meets the requested demand $d_1, d_2, d_3, ..., d_j$, for $\forall k \in [1, j]$ respectively (means, the distribution will be correct).

Assume the opposite for the sake of contradiction. Suppose that the algorithm returns a solution, but there exists at least one town t_j where the sum of incoming flow from reservoirs does not meet its demand [WTS: it cannot happen]. We know that (1) the edges in the residual graph are constrained by the supply relationship (minimum) conditions i.e. bottleneck. (2) By the flow value of the cut lemma $V(f) = f^{out}(A) - f^{in}(A)$, then, if demand is not met ($f^{in}(A) > 0$), consider a cut exactly in between town and reservoirs and there should exist path where either capacity of edge connecting them is (a) equal to $t_j - d$ edge or (b) $s - w_n$ (from 1), where the other edge is greater or equal. It implies that the algorithm, by design, could send additional flow from the reservoirs to meet the demand of town t_j, but did not. We have shown that this additional flow would respect both the supply (the remaining capacity of the reservoirs) and demand (the demand of town t_j).

Since both cases lead to contradictions back in part 1, our assumption that a valid distribution exists but the algorithm returns a maximum flow not equal to $d_1+d_2+d_3+...+d_j$ must be also false.

2. Finally, if the distribution is impossible, my algorithm returns maximum flow less than $d_1 + d_2 + d_3 + ... + d_j$ (consequently, returns false).

Analogically, let's consider the situation where the distribution is impossible, but the algorithm returns true i.e. a maximum flow equal to or greater than $d_1 + d_2 + d_3 + ... + d_j$:

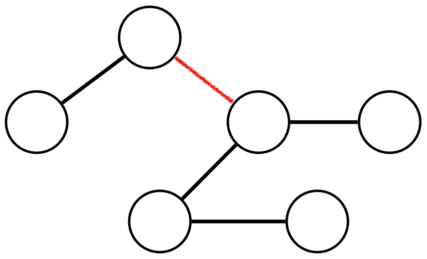
1. If it's greater, consider the proof from lemma 1.1, the flow greater than capacity constraint.
2. If equal, similarly to proof of lemma 1.2, consider the minimum cut that separates the source s from the sink d in the flow network right in between towns and reservoirs as constructed by minimum relationship. By the Min-Cut Max-Flow Theorem, the maximum flow is bounded by the minimum cut capacity. If the distribution is impossible, it means that at least one town t_k will have an unmet demand, while edges in between reservoirs and towns are saturated. This unmet demand represents k max-flow (cut) in the network. By assumption, therefore, if the amount of water reached to towns is greater than k, it contradicts the capacity constraint.

Thus, we have shown by byjection that the algorithm will produce correct value in all possible cases, considering that Ford-Fulkerson Algorithm works correctly. ■

Time Complexity. The total number of edges in the worst case can be expressed as the sum of non-zero elements in the matrix C^1 (notation), at most $O(nm)$ (for each n there is m vertices), augmented by the additional one source and sink edges, which adds $n+m$ to the count. Therefore, by Ford-Furkelson (Lemma 4), the overall time complexity of the algorithm is $O(F(nm+n+m))$, where F is the maximum flow in the network, which could also be represented as the sum of all supply. Thus, overall time complexity is $O(S*(nm+n+m)) = O(S(nm))$, where $S = s_1+s_2+...+s_n$ (the sum of all capacities coming of the source vertex).

4 Edge Connectivity of a Graph

The **edge connectivity** of an undirected graph $G = (V, E)$ is the smallest number of edges k that we have to remove from G such that the resulting graph becomes disconnected. For example, the edge connectivity of a tree is 1, and the edge connectivity of a cycle is 2.


(i) Removing 1 edge disconnects a tree.

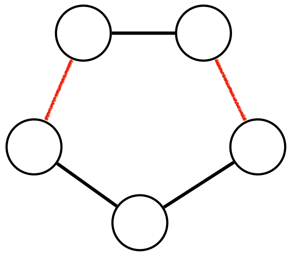

(j) Removing 2 edges disconnect a cycle.

Figure 1: Examples of edge connectivity.

Problem: You are given an undirected graph $G = (V, E)$ with n vertices and m edges as an adjacency list. Design an algorithm that computes the edge connectivity of G. Write your algorithm as a pseudocode. Prove the correctness of your algorithm and analyze its running time.

(*Hint:* Formulate as computing max-flow on a related network, possibly multiple times.)

Solution. Consider the following **algorithm**:

1. Input: Undirected graph $G = (V, E)$, so, first, construct a new bidirectional (two-sided) directed graph G'. Define $c(u, v) = 1$ for every $(u, v) \in E$ for the sake of counting edges and let f be an integral flow in G' of value k s.t. $f(e) \in 0, 1$ for all e.
2. Base Case Exception: For 0 or 1 vertecies in $G' \implies$ return 0.
3. Let s be a vertex in V (choose randomly). For each $t \in V - \{s\}$: Solve the min cut problem in the network (G', s, t, c), and let C be the cut of $\underline{\text{minimum capacity}}$ among all iterations. In particular, update $global = minlength\{global, minCut\}$ with $global = \infty$ initially. Thus, global saves the set of minimum amount of edges we need to cut to get a disconnected graph.
4. Output: The cut C_t with global minimum costs, for some iteration of loop with sink t.

Claim. Algorithm finds $min(C_t)$ (i.e. global min) $= k$ (i.e. edge connectivity output).

Proof. Case 1. To demonstrate that the algorithm identifies the global minimum cut, consider the following. Let the edge-connectivity of the graph be denoted as k. This implies that there exists a set of k edges, denoted as E^*, such that removing these edges disconnects the graph. Then, after removing these edges, by the setup of k, we have two disjoint sets of vertices, and we'll designate the set containing vertex s (source) as S^* for the sake of the proof. So, now, S^* is a global minimum cut of cost at most k (in fact, it's exactly k since it disconnects the graph). Furthermore, it is known that S^* contains vertex s. Since connected, in $\underline{\text{at least one iteration } t}$ of the algorithm, it constructs a network (G, s, t, c) in which vertex t is not in S^*, but on any other side of the cut. This implies that S^* is a valid cut for the network with a $\underline{\text{capacity}}$ of k, since vertex t is not reachable anymore from s during the execution of the Ford-Fulkerson Algorithm.

Case 2. As the algorithm proceeds to find the minimum capacity cut in this network, it must discover a cut with a capacity of at most k (indeed, exactly k) because the capacity of the cut cannot be greater than the capacity of the global minimum cut among all possible t. This implies that for at least one value of t, the cut C_t is also an optimal global minimum cut, with a capacity of k by the Min-Cut Max-Flow. Therefore, the algorithm identifies $\underline{\text{at least one}}$ global minimum cut during its execution, as it examines different values of t, one of which corresponds to the global minimum cut. Finally, in the base case of lower than 1 (including) vertices, return 0 (no cut exist). ■

Time Complexity. The algorithm involves $|V| - 1$ minimum cut calculations in networks. Each of these cuts can be determined through a maximum flow computation. As the maximum flow in each network has a cost at most $|V| - 1$, and given that all capacities are integers (Lemma 3), the Ford-Fulkerson algorithm computes each maximum flow in $O(|E| * F^{max}) = O(|E| * |V|)$ time (from the presentation). Consequently, the total running time is $O(|E| * |V|^2)$ in total.

5 Completing Latin Squares

A **Latin square** is an n-by-n grid in which each cell has been filled with an integer between 1 and n so that every row and every column contains all n integers. Here are two examples for $n = 4$:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \\ 3 & 4 & 1 & 2 \\ 4 & 1 & 2 & 3 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \\ 3 & 4 & 1 & 2 \\ 4 & 3 & 2 & 1 \end{bmatrix}$$

How do we find Latin squares?

We can try to fill in each row one by one. The first row is easy: we can fill it with any permutation of $1, \ldots, n$, but the subsequent rows are challenging due to the constraint on each column. Nevertheless, it turns out this strategy works: we can always extend a valid partial filling to the next row. In this problem, your goal is to produce an algorithm to do so.

Concretely, suppose we are given a *partially filled Latin square* which is an n-by-n grid in which the first k rows (for some $0 \le k < n$) have been filled in. We will say that a partially filled Latin square is *valid* if the entries that have been filled in every row and column are distinct numbers.

Problem: Design an algorithm that takes as input a *valid* partially filled square (with k rows filled in, for some $0 \le k < n$), and outputs another valid partially filled square with $k + 1$ rows filled in. Prove the correctness of your algorithm and analyze its running time.

Example: The algorithm takes as input a valid partially filled square, such as:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 3 & 1 & 4 & 2 \\ ? & ? & ? & ? \\ ? & ? & ? & ? \end{bmatrix}$$

and outputs another valid partially filled square with one additional row filled in, such as:

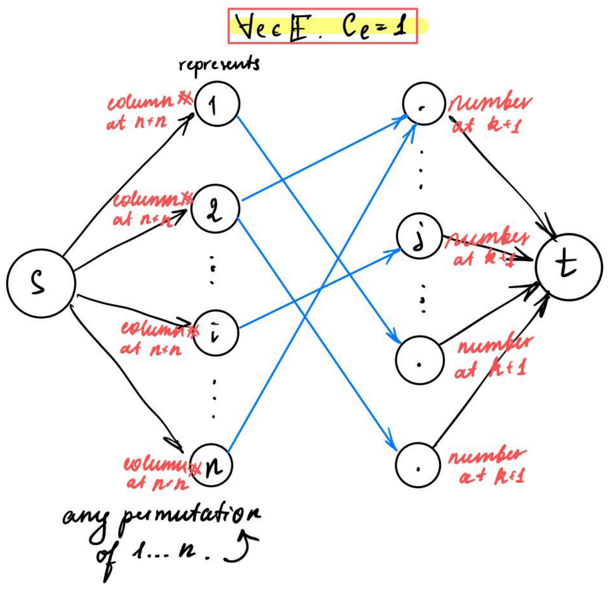
$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 3 & 1 & 4 & 2 \\ 2 & 4 & 1 & 3 \\ ? & ? & ? & ? \end{bmatrix}.$$

Note there may be multiple possible outputs, and it does not matter which one you output as long as it is valid.

(*Hint:* Consider a bipartite graph with n vertices on each side. What should the vertices and edges be to encode the input Latin square configuration?)

Solution. Algorithm:

First, create a bipartite graph with two sets of vertices: numbers 1 to n and empty cells in the current row (distinct values). [Want to match numbers to their possible positions]. Second, create edges between numbers and empty cells if the number does not appear in the same column as the empty cell. Now, to find a maximum cardinality matching in the bipartite graph, we can convert this problem into network flow and run the Ford-Fulkerson Algorithm. To do that, add extra source vertex being connected to all verticies of the first column. Similarly, add sink vertex being connected to all verticies in the last (empty values) column (consider such graph below). Define $c(u,v) = 1$ for every $(u,v) \in E$ for the sake of byjective distribution and let f be an integral flow in G' of value k s.t. $f(e) \in 0,1$ for all e. Want $|M| = k$ by the flow-value lemma to cut (A,B) with $A = X \cup \{s\}, B = Y \cup \{t\}$ during the Ford-Fulkerson execution. After execution of the algorithm, if perfect matching is found, assign back the matched numbers from bypartily to the empty cells in the current row of table and returned the $k+1$ filled table.



Time Complexity.

Filling the one row takes $O(n)$ time. Creating the bipartite graph with $O(n)$ vertices and $O(n^2)$ edges at most takes $O(n^2)$ time. Finding a maximum cardinality matching in a bipartite graph can be done in $O(|E| * |V|) = O(n^3)$ time. Thus, **overall**, the running time of the algorithm is $O(n + n^3 + n^3) = O(n^3)$, where n is the size of the Latin square.

Proof. Claim: If there exist a matching, my algorithm finds it ($\Longrightarrow$):

- The algorithm creates a bipartite graph to match the remaining numbers (1 to n) with the empty cells in the $(k+1)$th row. Each edge is created only if the number does not appear in the same column as the empty cell. Therefore, at this stage, we ensure that each number appears at most once in each column and row, and all entries are distinct.
- We also know, in the bipartite matching, if there is a solution, the output is a perfect matching since the amount of flow in the input and output is limited to 1 and only one augmenting path could be mapped by the setup of the algorithm above (either 0 flow or 1). Otherwise, if there exist more than one augmenting path that adds to the same empty cell in the graph G' that would violate the capacity constraint previously determined. Thus, if there is a correct flow in the built flow-network graph, then by Reduction to Max-Flow Theorem (class), we have a perfect matching of the same size as the value of the flow.

Thus, if there exist a matching, since the input is a valid partially filled Latin square, the partial filling ensures that no number repeats in the same column or row, and each number is used only once in the first k rows. Therefore, the algorithm then $\underline{\text{finds}}$ a maximum cardinality matching in the bipartite graph, ensuring that each empty cell in the row is assigned a unique number.

Claim: The solution found by the algorithm is a perfect matching ($\Longleftarrow$):

Assume for the sake of contradiction that the solution found by the algorithm is $\underline{\text{not}}$ a perfect matching. This means that there must be at least one of the following scenarios:

- Some number in the range 1 to n is left unmatched in the bipartite graph. This would imply that at least one empty cell in the $(k+1)$th row remains unassigned.
- There is a pair of numbers that are both assigned to the same empty cell in the $k+1$ row.

In either scenario, the Latin square $\underline{\text{properties would be violated}}$. If there is an unmatched number or an unassigned empty cell, the Latin square would not have all distinct integers from 1 to n in the $k+1$ row or column, which is a fundamental property of a Latin square. If two numbers are assigned to the same empty cell, this would result in a repetition of numbers in the $k+1$ row, violating the rule that each entry must be distinct. ■