Introduction to Algorithms, T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Prentice Hall India, Chapter 9, 26

9.2-4. Suppose we use RANDOMIZED-SELECT to select the minimum element of the array A = (3, 2, 9, 0, 7, 5, 4, 8, 6, 1). Describe a sequence of partitions that results in a worst-case performance of RANDOMIZED-SELECT.

Solution:

We will have the worst case if the maximum number is chosen as pivot in every iteration.

Partition 1:

Suppose maximum element 9 is chosen as pivot.

3 2 **9** 0 7 5 4 8 6 1

Exchange elements 9 and 1 as initially we assume the pivot is present at the last position. So, we have,

3 2 **1** 0 7 5 4 8 6 **9**

A partition should be chosen such that all the elements to the right of it must be greater that it and left of it is smaller than or equal to it. Now we observe that 9 is that position. There are no elements right of 9 but all elements to left of 9 are smaller than it. So, 9 is the partitioning point and 9 is placed in its proper position. No, we apply recursive call to the left of the partition only as right side contain no elements.

Partition 2:

Now maximum element 8 is chosen as pivot from the elements passed recursively in the previous partition

3 2 1 0 7 5 4 **8** 6

Exchange elements 8 and 6. So we have,

3 2 1 0 7 5 4 **6** **8**

We observe all the elements to left of 8 is less than or equal to 8 and all to the right of it is greater than it (there is no element to right of 8, so this condition holds trivially). So, 8 is the next correct point of portioning. We apply recursive call to left side of the partitioning element.

Partition 3:

Now maximum element 7 is chosen as pivot from the elements passed recursively in the previous partition

3 2 1 0 **7** 5 4 6

Exchange elements 7 and 6. So we have,

3 2 1 0 **6** 5 4 **7**

We observe all the elements to left of 7 is less than or equal to 7 and all to the right of it is greater than it (there is no element to right of 7, so this condition holds trivially). So, 7 is the next correct point of portioning. We apply recursive call to left side of the partitioning element.

Continuing in the same way we have,

Partition 4:

3 2 1 0 **6** 5 4

3 2 1 0 **4** 5 **6**

Partition 5:

3 2 1 0 4 **5**

Here randomly chosen pivot 5 is present at last position only, so no exchange is required.

Partition 6:

3 2 1 0 **4**

Similarly, randomly chosen pivot 4 is present at last position only, so no exchange is required.

Partition 7:

3 2 1 0

0 2 1 **3**

Partition 8:

0 **2** 1

0 **1** **2**

Partition 9:

0 **1**

Partition 10:

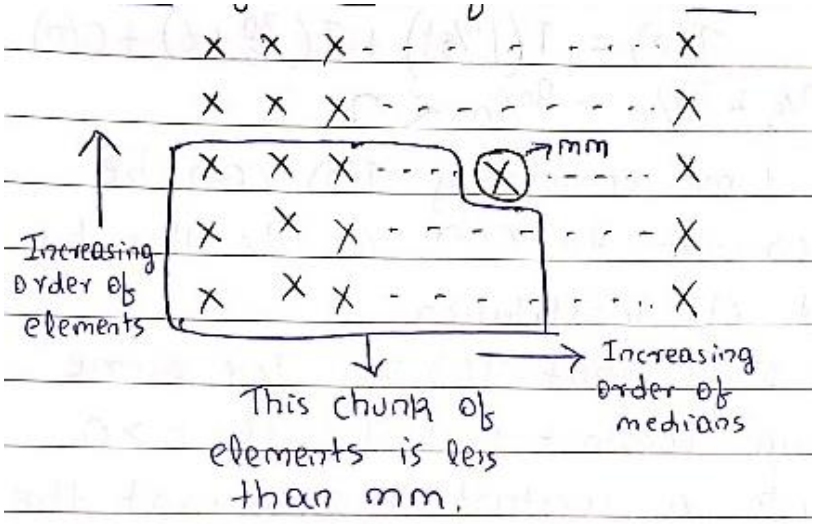
0

The minimum element is found. Number of remaining elements in the partition is 1, i.e. the minimum itself. We know cost of each partitioning in $O(n)$ and is worst case the minimum is found after checking all the elements. So, we have, $n * O(n) = O(n^2)$

9.3-1. In the algorithm SELECT, the input elements are divided into groups of 5. Will the algorithm work in linear time if they are divided into groups of 7? Argue that SELECT does not run in linear time if groups of 3 are used.

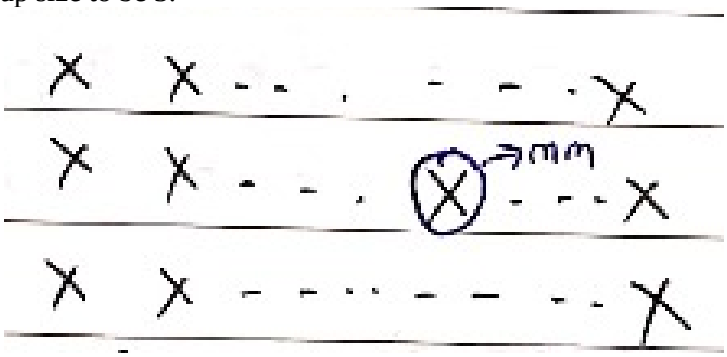
Solution:

We visualize the input elements of an array A to be arranged in a 2D matrix with each column consisting of elements from the same group. The group elements are sorted in increasing order and also the also the groups are sorted in increasing order of the medians.



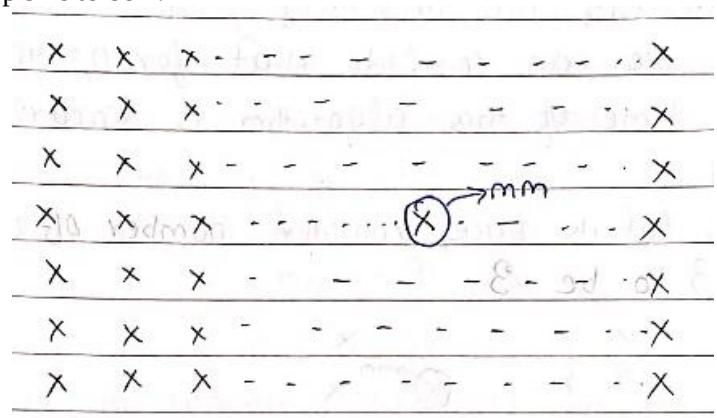
There are $\lfloor \frac{n}{5} \rfloor$ groups in A. Number of groups in $A_<$ = $\left\lfloor \lceil \frac{n}{5} \rceil \times \frac{1}{2} \right\rfloor - 2$. We subtract the 2 to reduce overestimation. Hence, total elements in $A_< = 3 \times \left(\left\lfloor \lceil \frac{n}{5} \rceil \times \frac{1}{2} \right\rfloor - 2 \right) = \frac{3n}{10} - 6$ and total elements in $A_> = n - \left(\frac{3n}{10} - 6 \right) = \frac{7n}{10} + 6$. Now, $Max(|A_<|, |A_>|) = \frac{7n}{10} + 6$. Finally, we can write $T(n) = T\left(\lfloor \frac{n}{5} \rfloor \right) + T\left(\frac{7n}{10} + 6 \right) + O(n)$. Now, $\frac{n}{5} + \frac{7n}{10} = \frac{9n}{10} < n$. Thus, the complexity $T(n)$ can be shown to converge to O(n).

We now consider the group size to be 3.



Hence, we have $|A_<| = \left(\left\lfloor \lceil \frac{n}{3} \rceil \times \frac{1}{2} \right\rfloor - 2 \right) \times 2 = \frac{n}{3} - 4$ and $|A_>| = n - \left(\frac{n}{3} - 4 \right) = \frac{2n}{3} + 4$. The time complexity will thus be $T(n) = T\left(\lfloor \frac{n}{3} \rfloor \right) + T\left(\frac{2n}{3} + 4 \right) + O(n)$. Also, $\frac{n}{3} + \frac{2n}{3} = n \not< n$. Hence, during recursion we cannot discard any elements.

Finally, we consider group size to be 7.



Hence, we have $|A_<| = \left(\left\lfloor\left\lfloor\frac{n}{7}\right\rfloor \times \frac{1}{2}\right\rfloor - 2\right) \times 4 = \frac{4n}{14} - 8 = \frac{2n}{7} - 8$ and $|A_>| = n - \left(\frac{2n}{7} - 8\right) = \frac{5n}{7} + 8$. The time complexity will thus be $T(n) = T\left(\left\lfloor\frac{n}{7}\right\rfloor\right) + T\left(\frac{5n}{7} + 8\right) + O(n)$. Also, $\frac{n}{7} + \frac{5n}{7} = \frac{6n}{7} < n$. Hence, during recursion we can discard elements.

9.3-8. Let X [1…n] and Y [1…n] be two arrays, each containing n numbers already in sorted order. Give an $O(logn)$ time algorithm to find the median of all 2n elements in arrays X and Y.

Solution:

Input: Two sorted arrays A [1…. n] and B [1……m]
Output: Median of the n + m elements, median
Assumption: The two arrays must be sorted in increasing order.
Let n ≤ m, i.e. A is shorter.

Steps:

1. Start
2. Binary Search on shorter array. Set startA = 0 and endA = n
3. Partition both array at some point.

$$\text{partitionA} = \frac{(\text{startA} + \text{endA})}{2}$$

$$\text{partitionB} = \frac{(\text{n+m-1})}{2} - \text{partitionA}.$$

The partition must be such that number of elements in combined array of A and B must be equal on both sides.

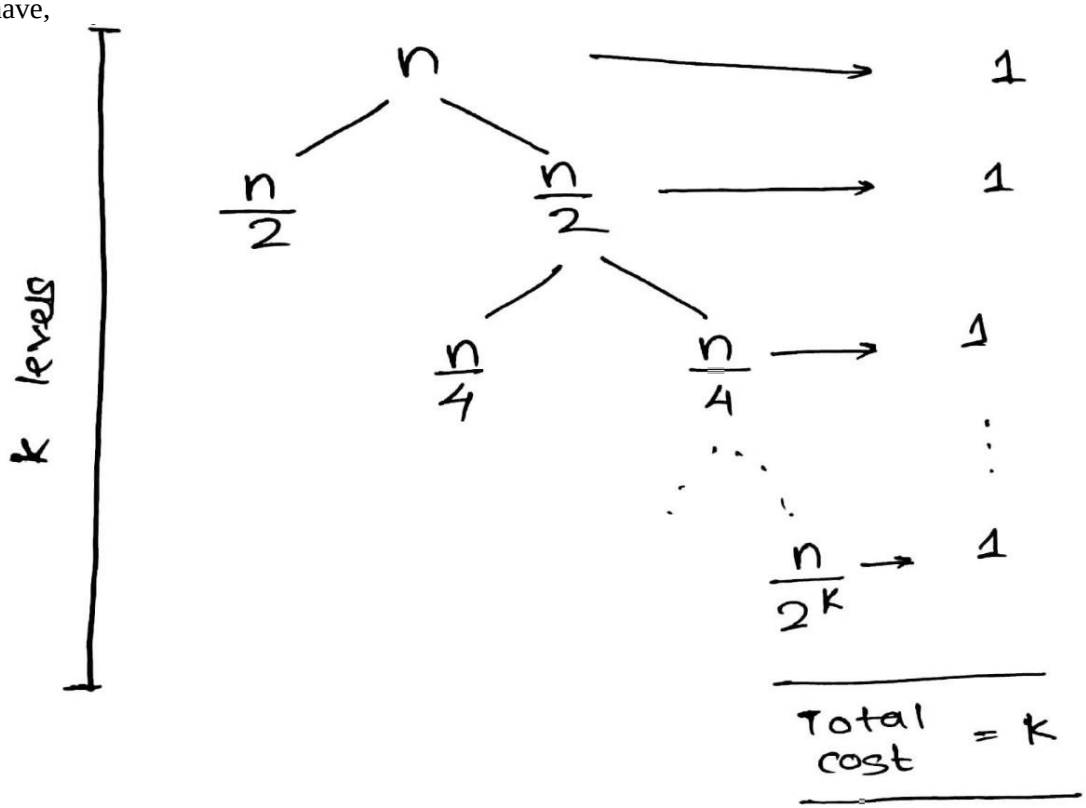
4. A_1 is the maximum element on left partition of A and A_2 is the minimum element on the right partition of A. B_1 is the maximum element on the left partition of B and B_2 is the minimum element on the right partition of B.
Note: If the left side of any partition contain no element, i.e. partitioning point is along the edge then we assume the element to be −INF and if same situation arises on the right side then take it to be +INF.
5. If $A_1 \leq B_2$ and $B_1 \leq A_2$, then partition is correct, i.e. all elements to left of partitioning point of A is smaller than all the elements to right of partitioning point of B and all elements to the left of the partitioning point of B is less than all the elements to the right of partitioning point of A.
6. A_1, A_2, B_1, B_2 are on the border of the partition. If partitioning is correct, Median must be among these elements.
if(n + m == even)
median = avg $[\max(A_1, B_1), \min(A_2, B_2)]$
otherwise,
median = max (A_1, B_1)

[This is because left partitioning contains one extra element than the right which must be the median. GOTO Step 8.]

7. If the partition is not correct i.e. condition in Step 5 does not hold. Then we need to find whether partitioning point in A need to move left or right.
   If $A_1 > B_2$ i.e. all elements to the left of A is not less than all elements to right of B. Then partitionA needs to move left. Hence, we search for median in left side of current partitioning point of A. Thus, the array reduces to $\text{end}A = \text{partition}A - 1$; otherwise, partition has to shift right. We search for partitioning point in the right side of current partition of A. Thus, the array reduces to $\text{start}A = \text{partition}A + 1$. GOTO Step 3.
8. PRINT median
9. Stop

Analysis:

At every iteration we look for proper partitioning in any one side of the array.
Thus, the recursion is $T(n) = T\left(\frac{n}{2}\right) + 1$; where $T\left(\frac{n}{2}\right)$ assumes each time, partitioning occurs at the middle and 1 refers to the cost of finding partition after sub-dividing is constant. Thus, we have,



Now, $\frac{n}{2^k} = 1$ implying$k = \log_2 n$. Therefore, $T(n) = \log_2 n = O(\log n)$
Now we have assumed$n \leq m$, but if we remove this restriction, we always perform binary search on $\min(m, n)$. Thus, running time becomes, $T(n) = O(\log(\min(m, n)))$

26.1-3. Suppose that a flow network G = (V, E) violates the assumption that the network contains a path s →v→ t for all vertices v ϵ V. Let u be a vertex for which there is no path s → u → t. Show that there must exist a maximum flow f in G such that f (u, v) = f (v, u) = 0 for all vertices v ϵ V.

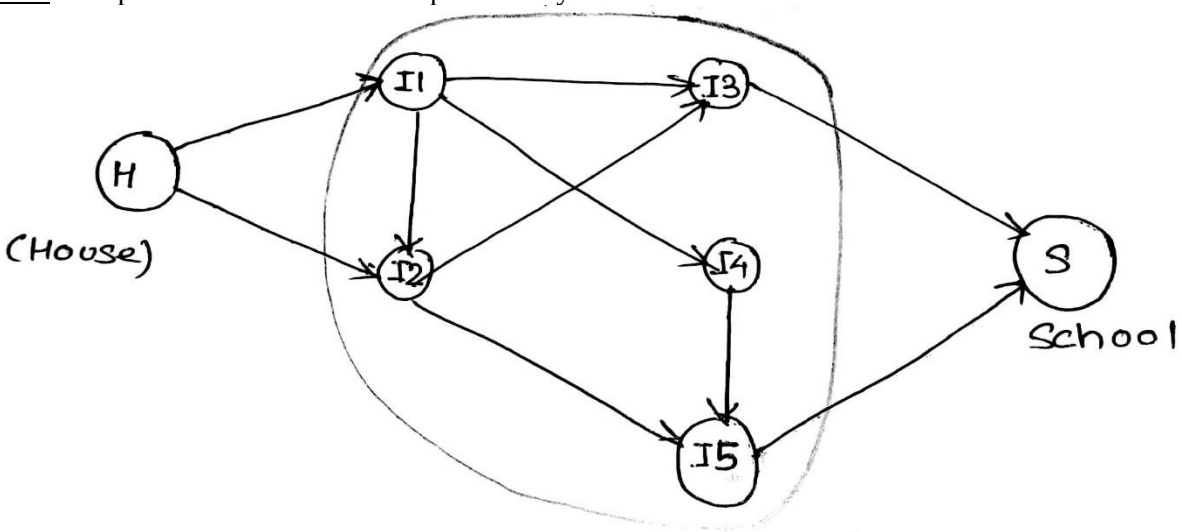
Solution:

We are given a flow network G = (V, E) which violates the assumption that the network contains a path s →v→ t for all vertices v. Let u be a vertex for which there is no paths s →u→ t. So, there can occur two conditions, either no path exists i) from s to u or ii) from u to t. Hence, u is such a vertex in which if condition (i) occurs then inflow to u implies there will be an outflow from u. So, to maintain the law of conservation, there will only be zero flow from any other vertex to u. Again, if condition (ii) occurs, an outflow from u implies an inflow to u. Therefore, to maintain the conservation property, there should be zero flow out of u. So, we have established the fact that f (u, v) = 0 and f (v, u) = 0. Now, because there is no path, no more flow can be conducted in the network. So, from the above fact which is established there can exist a maximum flow f in G of zero only.

26.1-6. Professor Adam has two children who, unfortunately, dislike each other. The problem is so severe that not only do they refuse to walk to school together, but in fact each one refuses to walk on any block that the other child has stepped on that day. The children have no problem with their paths crossing at a corner. Fortunately, both the professor's house and the school are on corners, but beyond that he is not sure if it is going to be possible to send both of his children to the same school. The professor has a map of his town. Show how to formulate the problem of determining whether both his children can go to the same school as a maximum-flow problem.

Solution:

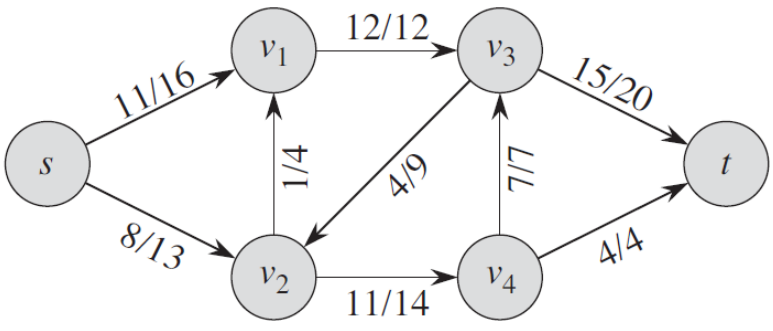
Given: The professor has the street map of the city.



Using this street map, we construct an undirected graph G = (V, E) where each vertex $I_i \in V$. Each vertex I_i represents intersection between two streets whereas each edge represents a street. The professor has two children, say c_1 and c_2 and the problem is to find if there exist two distinct paths so that the two children can travel from home to school without intersecting each other's path. We introduce two new nodes H and S, representing home and school respectively and connect them to those vertices that are reachable, i.e. those intersecting points that are directly reachable from H or S. Now we make the edges directed so that we can reach school from home. We are to determine capacity of each edge. Now, if an edge is traversed by c_1 then it cannot be traversed by c_2. Thus, the capacity of each edge is 1.

We have a few assumptions. The house is a source vertex and school is the sink vertex. Thus, there is no edges incoming in H and outgoing from S. This is well justified, because from the school we consider that the children are not to go anywhere and we also do not consider what happens when both reach home. We can easily map the given problem into a network flow problem. Now, we have to find the maximum flow. If two distinct paths are found such that they do not intersect at any point then the flow, f must be greater than 2, i.e. $|f| \geq 2$. Otherwise, if they intersect at a point it means, there exists an edge which is common and removal of that edge makes the graph disconnected. Thus, only one can go to school if a path exists, or else none can go to school. Hence, the flow f is less than 1, i.e. $|f| \leq 1$. We can conclude that the professor using network flow can determine if the children can go to school or not.

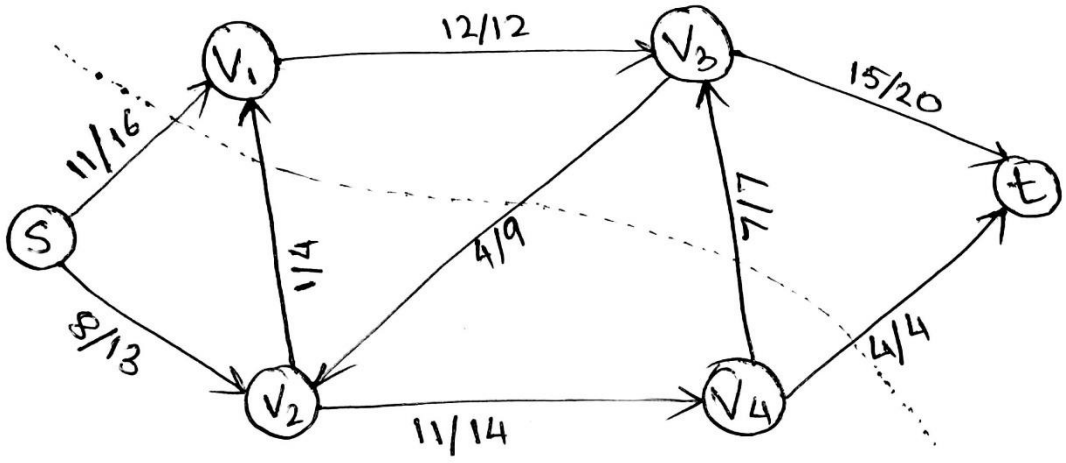
26.2-2.



In Figure the, what is the flow across the cut $(\{s, v_2, v_4\}, \{v_1, v_3, t\})$? What is the capacity of this cut?

Solution:

We have to find $\text{CUT}(\{s, v_1, v_2\}, \{v_1, v_3, t\})$. Let A be $\{s, v_1, v_2\}$ and B be $\{v_1, v_3, t\}$. We know the flow across $\text{CUT}(A, B) = f^{out}(A) - f^{in}(A)$. Flow out of A: $s \rightarrow v_1 = 11$; $v_2 \rightarrow v_1 = 1$; $v_4 \rightarrow v_3 = 7$; $v_4 \rightarrow t = 7$. Flow into A: $v_1 \rightarrow v_2 = 4$. So the flow across the $\text{CUT}(A, B) = 23 - 4 = 19$ and the Capacity of the $\text{CUT}(A, B) = \sum_e$ Cut Edges out of $A = 16 + 4 + 7 + 4 = 31$

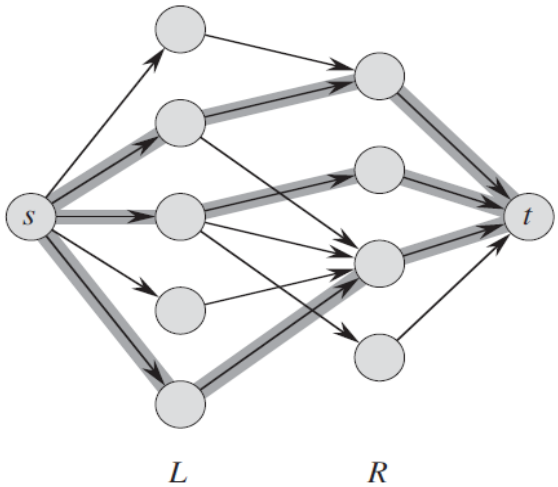


26.2-12. Suppose that you are given a flow network G, and G has edges entering the source s. Let f be a flow in G in which one of the edges (v, s) entering the source has f (v, s) = 1. Prove that there must exist another flow f' with f' (v, s) = 0 such that |f| = |f'|. Give an O(E) time algorithm to compute f', given f, and assuming that all edge capacities are integers.

Solution:

There exists a path for all vertices starting from s. So, there will be a cycle and an edge (v, s) will also exist. To satisfy conservation of flow, a cycle will exist with no edges of zero flow. Such a cycle is found using Depth First Search. Since the graph is connected, this takes O(E) time. Then the bottleneck value '1' is documented from every edge on the cycle. This preserves the value of the flow so it is still maximal. Since all edges had non-zero flows previously, the capacity constraint is maintained. Again, the conservation constrain is maintained, since the values are documented at both the in-edge and out-edge for each vertex on the cycle by some amount. So, we can say the given statements are proven to be true.

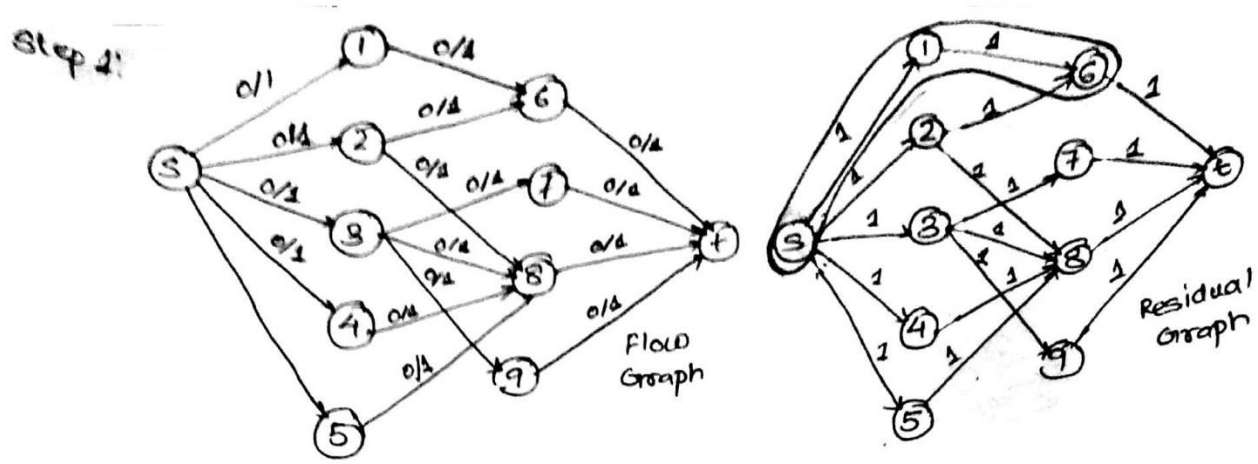
26.3-1.



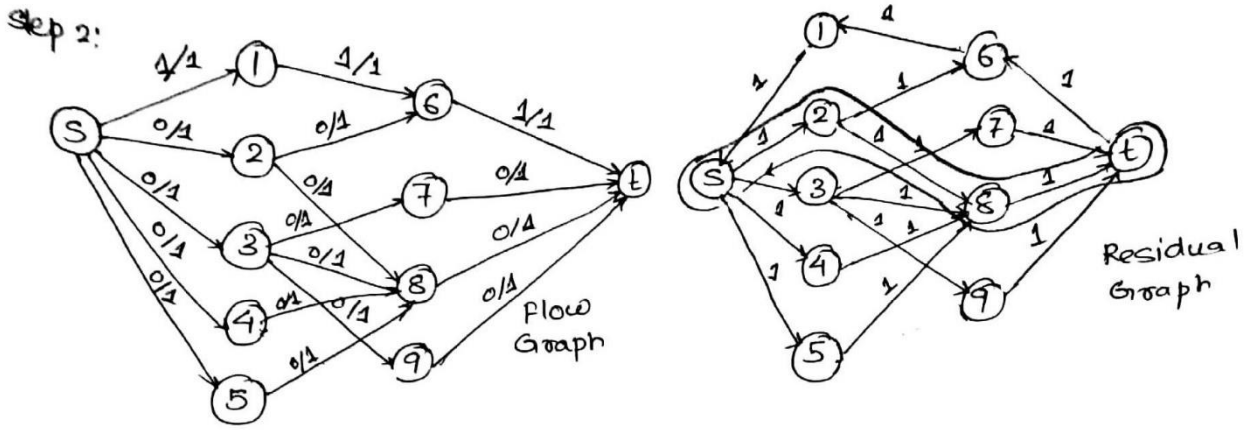
Run the Ford-Fulkerson algorithm on the flow network in the Figure and show the residual network after each flow augmentation. Number the vertices in L top to bottom from 1 to 5 and in R top to bottom from 6 to 9. For each iteration, pick the augmenting path that is lexicographically smallest.

Solution:

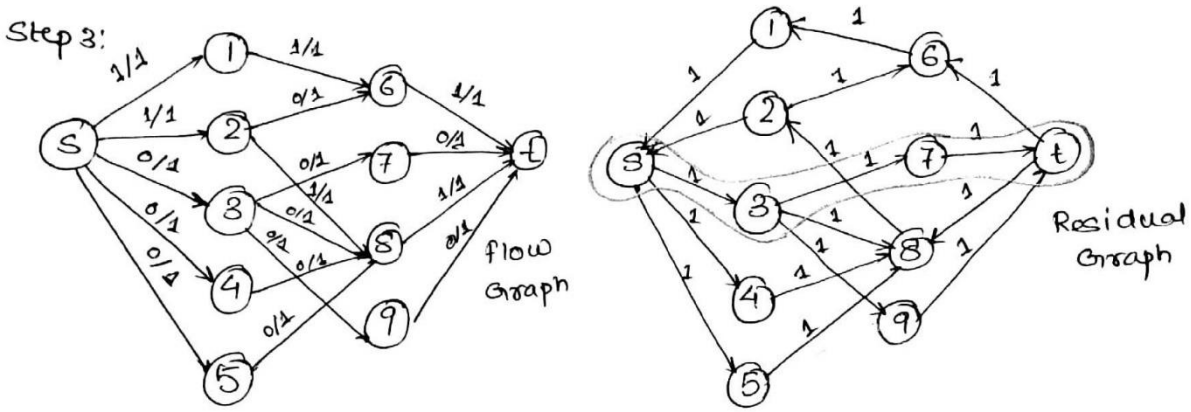
We label the above graph in the following manner.



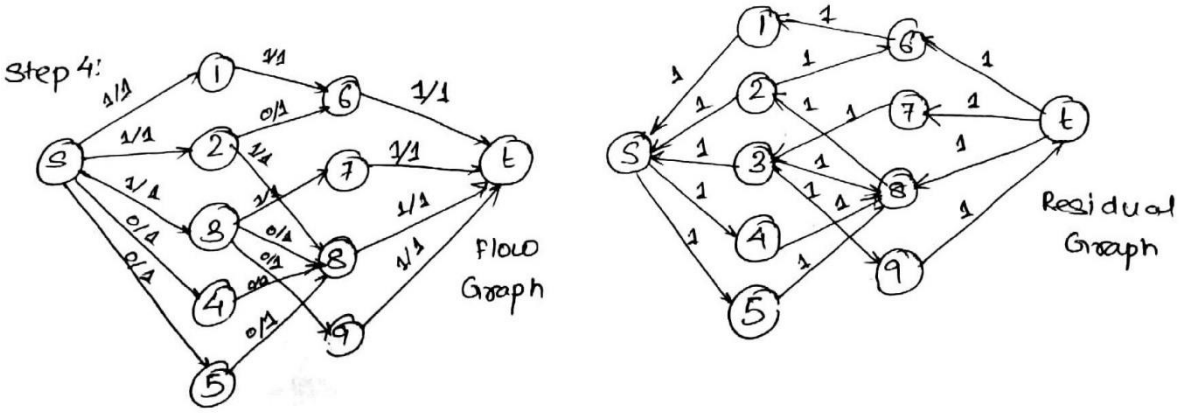
Now it is given for every iteration we need to choose the lexicographically smallest path. So, we choose the path s →1→ 6→t from the residual graph with bottleneck 1. Hence, the new flow graph and the corresponding residual graph is as follows.



The next lexicographically smallest path we observe is s →2→ 8→t from the residual graph with bottleneck 1. The new flow graph and the corresponding residual graph is as follows.



The next lexicographically smallest path we observe is s →3→ 7→t from the residual graph with bottleneck 1. The new flow graph and the corresponding residual graph is as follows.



In this residual graph we observe that there is no more augmenting path from s to t in the residual graph. Thus, the maximum flow has been reached. We chose (1, 6), (2, 8) and (3, 7) as matched pairs.

Algorithms: Design Techniques and analysis, M. H. Alsuwaiyel, Chapter 17

17.8. Prove or disprove the following statement. Using an algorithm for finding a maximum matching by finding augmenting paths and augmenting using these paths, whenever a vertex becomes matched, then it will remain matched throughout the algorithm.

Solution:

We start by some arbitrary matching. Now every time we start finding a augmenting path from a free vertex and end at some other free vertex and end at some other free vertex (if augmenting path exists). Now, to add this augmenting path we XOR it with the graph and all matched and unmatched edges' roles are reversed for all the edges in the path. Each path (except where we start and end) has at least one incoming edge and one outgoing edge (each edge has opposite roles). Now after augmentation the roles are reversed (for the edges in the path) but the vertex still has one matched edge associated with it. Moreover, after augmentation two new vertices at either end of the path become matched. Thus, vertices after becoming matched, never become unmatched.

Algorithm Design, J. Kleinberg and Eva Tardos, Pearson Education (Indian edition), Chapter 7

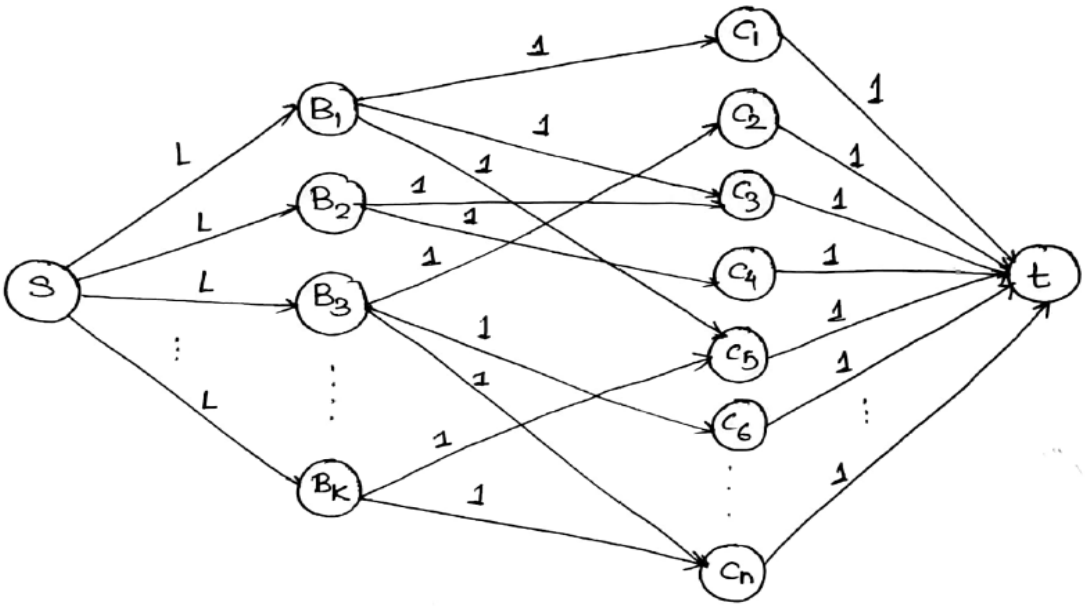
7. Consider a set of mobile computing clients in a certain town who each need to be connected to one of several possible base stations. We'll suppose there are n clients, with the position of each client specified by its (x, y) coordinates in the plane. There are also k base stations; the position of each of these is specified by (x, y) coordinates as well. For each client, we wish to connect it to exactly one of the base stations. Our choice of connections is constrained in the following ways. There is a range parameter r - a client can only be connected to a base station that is within distance r. There is also a load parameter L - no more than L clients can be connected to any single base station. Your goal is to design a polynomial-time algorithm for the following problem. Given the positions of a set of clients and a set of base stations, as well

as the range and load parameters, decide whether every client can be connected simultaneously to a base station, subject to the range and load conditions in the previous paragraph.

Solution:

Let B denote the set of base stations, $B = \{B_1, B_2, \ldots\ldots, B_k\}$ and C denote the set of clients, $C = \{C_1, C_2, \ldots\ldots, C_n\}$. Given: Maximum allowable distance between a base station and client for a connection to exist is r. We have to construct a digraph, G = (V, E) such that $V = B \cup C$ and $E = \{\text{directed edge e between } B_i \text{ and } C_j \mid d(B_i, C_j) \leq r \; \forall i, j\}$. The weights associated with each edge or the capacity of each edge is 1, since, each client can be connected to at most one base station, indicating load 1 by each client.

To map the given problem into a network flow problem, we introduce two new vertices, s and t where s is the source vertex that is connected to all vertices in B and t is the sink vertex and all vertices in C are connected to t. Edge weight for all edges (s, B_i) $\forall i$ is L, where L is maximum number of clients a Base Station can support at an instance of time. Edge weights for all edges (C_j, t) $\forall j$ is 1, because each client can be connected to at least one base station. Now, it forms a network flow problem. We apply Ford Fulkerson Algorithm to find maximum flow, i.e. maximum number of connections possible with given load and location of a base station and clients and radius of communication. We can say that a client C_x can be connected to a base station B_y if we have a s-t path of capacity or value L+2.



8. Statistically, the arrival of spring typically results in increased accidents and increased need for emergency medical treatment, which often requires blood transfusions. Consider the problem faced by a hospital that is trying to evaluate whether its blood supply is sufficient. The basic rule for blood donation is the following. A person's own blood supply has certain antigens present (we can think of antigens as a kind of molecular signature); and a person cannot receive blood with a particular antigen if their own blood does not have this antigen present. Concretely, this principle underpins the division of blood into four types: A, B, AB, and O. Blood of type A has the A antigen, blood of type B has the B antigen, blood of type AB has both, and blood of type O has neither. Thus, patients with type A can receive only blood types A or O in a transfusion, patients with type B can receive only B or O, patients with type O can receive only O, and patients with type AB can receive any of the four types.

(a) Let S_O, S_A, S_B and S_{AB} denote the supply in whole units of the different blood types on hand. Assume that the hospital knows the projected demand for each blood type D_O, D_A, D_B andD_{AB} for the coming week. Give a polynomial-time algorithm to evaluate if the blood on hand would suffice for the projected need.

(b) Consider the following example. Over the next week, they expect to need at most 100 units of blood. The typical distribution of blood types in U.S. patients is roughly 45 percent type O, 42 percent type A, 10 percent type B, and 3 percent type AB. The hospital wants to know if the

blood supply it has on hand would be enough if 100 patients arrive with the expected type distribution. There is a total of 105 units of blood on hand. The table below gives these demands, and the supply on hand.

Blood Type	Supply	Demand
O	50	45
A	36	42
B	11	8
AB	8	3

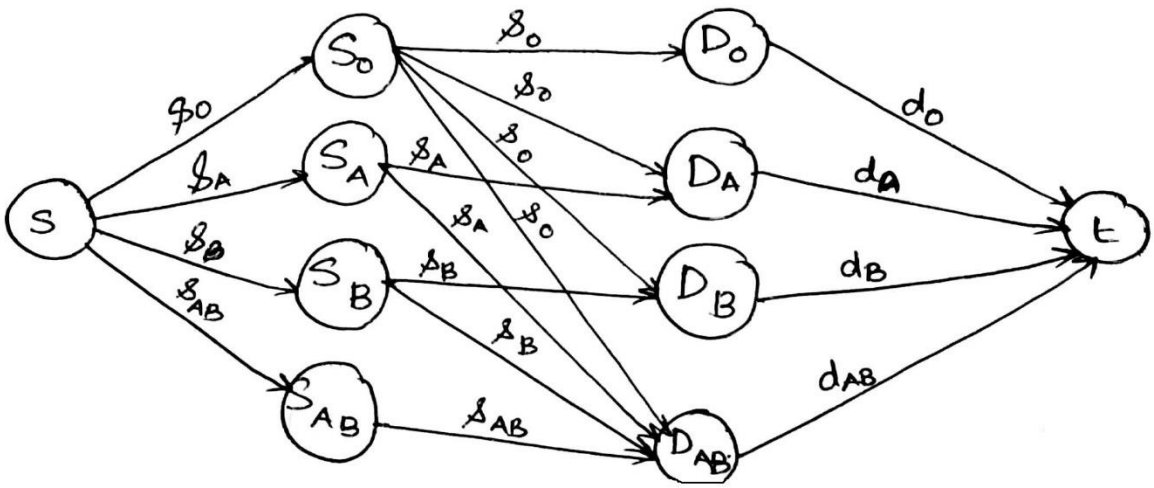
Is the 105 units of blood on hand enough to satisfy the 100 units of demand? Find an allocation that satisfies the maximum possible number of patients. Use an argument based on a minimum-capacity cut to show why not all patients can receive blood. Also, provide an explanation for this fact that would be understandable to the clinic administrators, who have not taken a course on algorithms. (So, for example, this explanation should not involve the words flow, cut, or graph in the sense we use them in this book.)

Solution:

Let $S = \{S_O, S_A, S_B, S_{AB}\}$ denote the availability of the corresponding blood groups. Let $D = \{D_O, D_A, D_B, D_{AB}\}$ denote the demand of the corresponding blood groups. The rules of blood transfusions are as follows:

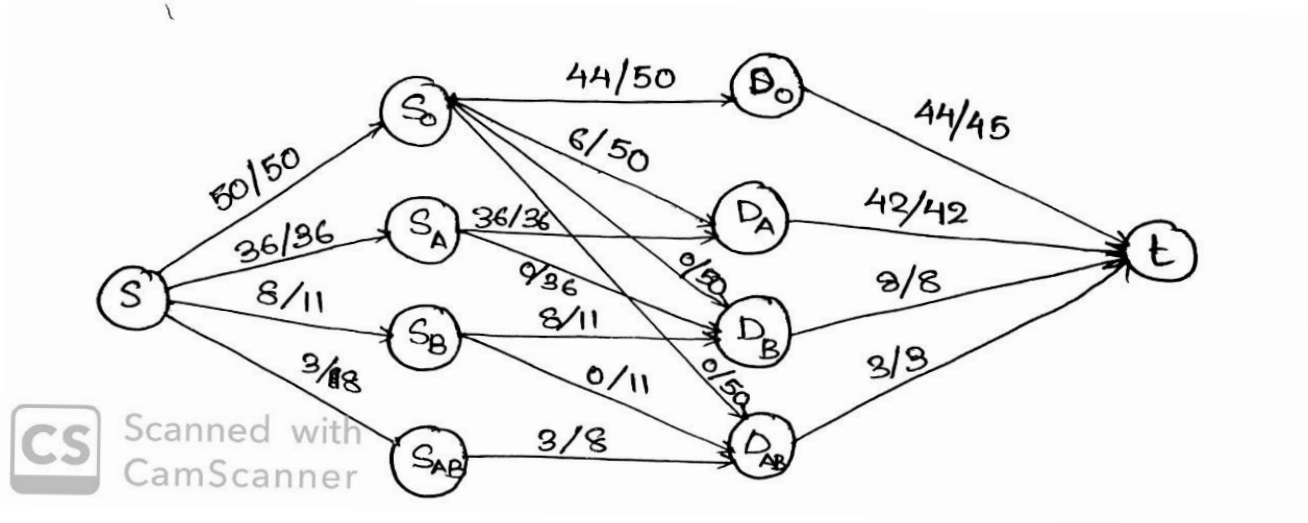
1. O can give blood to any group
2. A can give blood to A and AB
3. B can give blood to B and AB
4. AB can give blood to only AB

Now, let us construct a graph $G = (V, E)$ where $V = S \cup D$ and the edges exists obeying the above rules. Hence, we have,

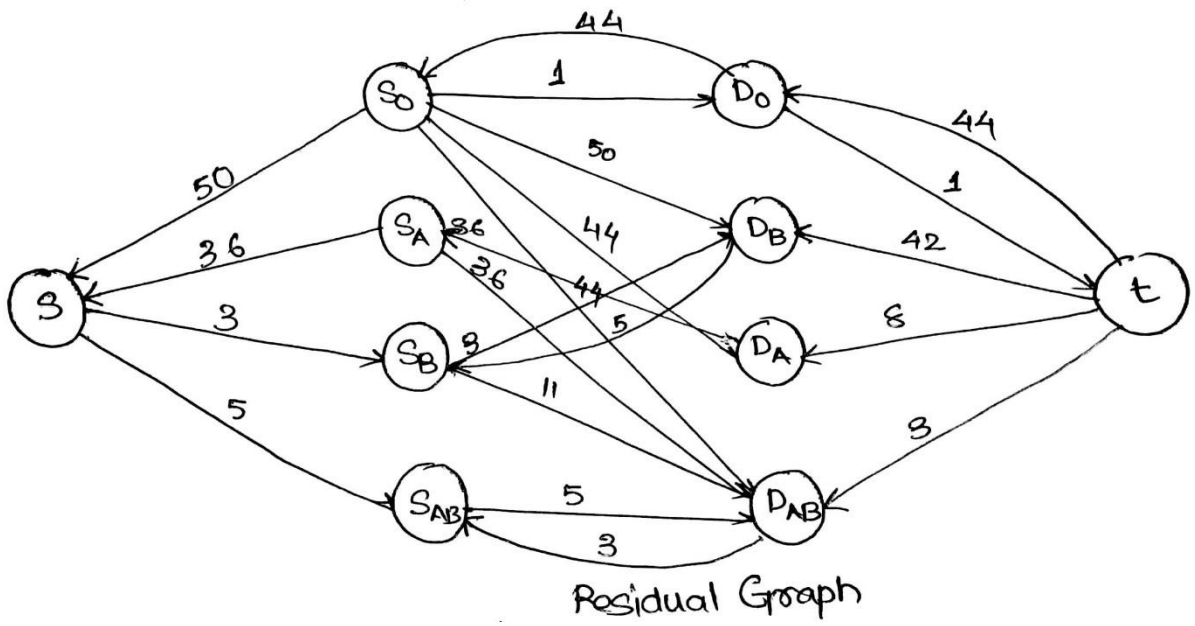


We introduce two vertices s and t as source and sink vertices. The source vertex connects to all the vertices in s and all vertices in D connects to sink vertex t. Now, we are to find the edge weights. For every edge (s, s_i), the weight is equal to maximum supply of the i^{th} blood group. For every edge (D_j, t), the edge weight is equal to maximum demand of j^{th} blood group. For every edge $(S_i, D_j) \forall i, j$, edge weight is equal to maximum supply of the i^{th} blood group. This is because maximum of that amount of blood can be supplied regardless of the demand. Each edge represents capacity of the edge. We have represented the given problem as a network flow problem. Now we can apply the Ford Fulkerson algorithm to find the maximum flow, i.e. the maximum demand that can

be fulfilled on the basis of blood requirements due to road accidents.



Only one person cannot be served with the given supply. We draw the residual graph of the above allocation and observe that there is no augmenting path from s to t. Hence the flow cannot be increased, i.e. maximum flow has been reached.



Thus, maximum flow has been reached. When we obtain maximum flow, we get minimum cut. Now, if an augmenting path from s to t would have existed then we could increase the flow. But no such path exists. So, we have reached maximum flow and all edges are not saturated. Hence, we can conclude that all demands cannot be fulfilled. We now observe that total requested is (45+42+8+3) and supplied is (50+36+3+5). Thus (45+42+8+3)-(50+36+3+5) = 1, cannot be served. Hence, all demands cannot be met.

9. Network flow issues come up in dealing with natural disasters and other crises, since major unexpected events often require the movement and evacuation of large numbers of people in a short amount of time. Consider the following scenario. Due to large-scale flooding in a region, paramedics have identified a set of n injured people distributed across the region who need to be rushed to hospitals. There are k hospitals in the region, and each of the n people needs to be brought to a hospital that is within a half-hour's driving time of their current location (so different people will have different options for hospitals, depending on where they are right now). At the same time, one doesn't want to overload any one of the hospitals by sending too many patients its way. The paramedics are in touch by cell phone, and they want to collectively work out whether they can choose a hospital for each of the injured people in such a way that the load on the hospitals is balanced: Each hospital receives at most $\lceil n/k \rceil$ people. Give a

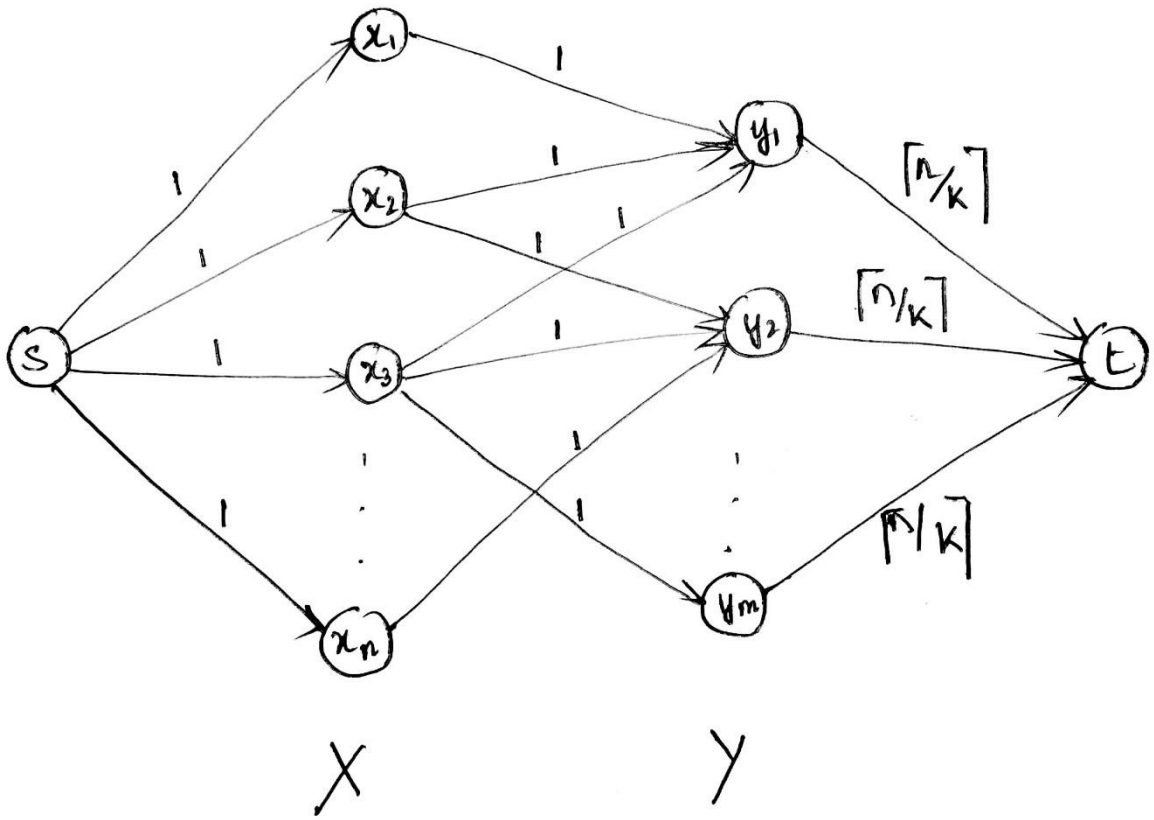
polynomial-time algorithm that takes the given information about the people's locations and determines whether this is possible.

Solution:

Each patient needs to be sent to a nearby hospital. So, the set of patients and hospitals form a bipartite graph G = (V, E). $V = X \cup Y$ and $X \cap Y = \emptyset$. Here, X represents the set of patients and Y represents the set of hospitals. E is the set of directed edges from X to Y, having weight 1 each. Each patient node in X has outgoing edges to hospital nodes in set Y that is within half an hour of their location. Now, we form a $G' = (V', E')$ where G' has all vertices and edges as in G with two vertices source s and sink t and edges joining them. The edges start from s and end at all the nodes in X and each has weight one. Another set of edges starts from all the vertices in Y and ends at t and each has weight $\left\lceil \frac{n}{k} \right\rceil$, since each hospital can be populated with at most $\left\lceil \frac{n}{k} \right\rceil$ people. Now G' forms a network flow model. Now on G' we apply the Ford-Fulkerson Algorithm to get the matching of patient to hospital.

1. Find a s-t path
2. Find the bottleneck of path
3. Increase the flow by bottleneck
4. Terminate if no path exists otherwise GOTO Step 1.

The construction of the graph takes $O(nK)$ time and solving the network flow by Ford-Fulkerson takes $O(mC^*)$ where m is the number of edges, and C^* is the sum of edge capacities coming out of s.



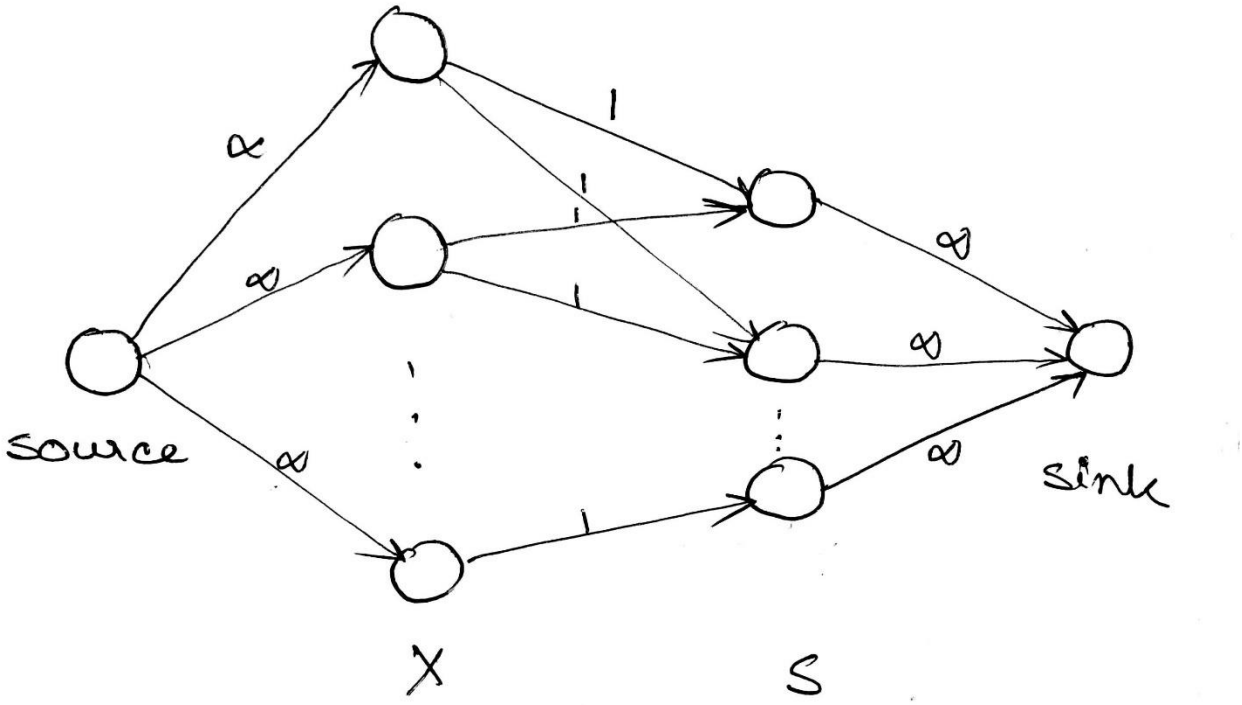
14. We define the Escape Problem as follows. We are given a directed graph G = (V, E) (picture a network of roads). A certain collection of nodes $X \subset V$ are designated as populated nodes, and a certain other collection $S \subset V$ are designated as safe nodes. (Assume that X and S are disjoint.) In case of an emergency, we want evacuation routes from the populated nodes to the safe nodes. A set of evacuation routes is defined as a set of paths in G so that

(i) Each node in X is the tail of one path,

(ii) The last node on each path lies in S, and
(iii) The paths do not share any edges. Such a set of paths gives a way for the occupants of the populated nodes to "escape" to S, without overly congesting any edge in G.
(a) Given G, X, and S, show how to decide in polynomial time whether such a set of evacuation routes exists.
(b) Suppose we have exactly the same problem as in (a), but we want to enforce an even stronger version of the "no congestion" condition (iii). Thus, we change (iii) to say "the paths do not share any nodes." With this new condition, show how to decide in polynomial time whether such a set of evacuation routes exists. Also, provide an example with the same G, X, and S, in which the answer is yes to the question in (a) but no to the question in (b).

Solution:

Given: G = (V, E) is a directed graph, $X \subset V$ is a set of populated nodes and $S \subset V$ is a set of safe nodes and $X \cap S = \emptyset$.



Evacuation routes are defined as a set of paths in G such that

i) Each node in X is the tail of one path
ii) The last node on each path lies in S
iii) The paths do not share edges

(a) Given G, X and S we have to determine if such a evacuation route exists or not. X and S are disjoint sets and edges starting at X ends in S. Now we have to match nodes from X to nodes in S. Multiple nodes in X can be matched to a single node in S. But no single edge can be used more than once. So, each edge has weight one. Now, X and S forms a bipartite graph. Hence, for each node in X, we declare a source node, 'source' and for each node in S, we declare a sink, 'sink'. From 'source' edges start and end in nodes at X, for all nodes in X. For all nodes in S, an edge starts at a node and ends in a 'sink'. The edges starting from 'source' and ending at a 'sink' has infinite weight.

i) From the graph thus formed, we select a path from source to sink.
ii) We find the bottleneck capacity of the path.
iii) We then increase the bottleneck capacity on the forward edges and decrease the bottleneck capacity on the backward edges.
iv) We then form the Residual Graph. A residual graph is one having forward edges and backward edges. Forward edges show the capacity that still can be pushed through the edge from vertex u to v. Backward edges show the capacity of the edge that have already been pushed through the edge from vertex u to v.
v) We choose a source-sink path and continue from step (ii). We stop if no such path exists.

(b) To enforce an even tighter condition of 'no congestion', the third condition changes to (iii) the paths do not share any nodes. Given G, X and S we need to decide whether such evacuation routes exist, enforcing an even stronger version of the "no congestion" condition. A node from X needs a matched pair at S. No two nodes in X can be matched to a single node in S. So, we assume for all nodes to be assigned to some node in S, $|S| \geq |X|$. X and S form a bipartite graph with edges directed towards S. Now we use the Hungarian tree method for solving the matching problem.

Let M be the maximum matching in G.

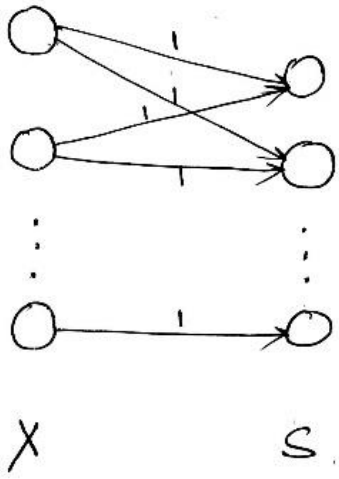
i) We initialize M to an arbitrary matching

ii) While $\exists$ a free vertex $x \in X$ and $s \in S$

a) Let r be the free x vertex. Using Breadth First Search we grow an alternating path tree T rooted at r.

b) If T is a Hungarian Tree $G = G - \{T\}$

c) Else we find any augmenting path p in T and let Matching$M = M \oplus p$, where $\oplus$ is the augmenting operator.



15. Suppose you and your friend Alanis live, together with n - 2 other people, at a popular off-campus cooperative apartment, the Upson Collective. Over the next n nights, each of you is supposed to cook dinner for the co-op exactly once, so that someone cooks on each of the nights. Of course, everyone has scheduling conflicts with some of the nights (e.g., exams, concerts, etc.), so deciding who should cook on which night becomes a tricky task. For concreteness, let's label the people $\{p_1, \ldots, p_n\}$, the nights $\{d_1, \ldots, d_n\}$; and for person p_i, there's a set of nights $S_i \subset \{d_1, \ldots, d_n\}$ when they are not able to cook. A feasible dinner schedule is an assignment of each person in the coopto a different night, so that each person cooks on exactly one night, there is someone cooking on each night, and if p_i cooks on night d_j, then $d_j \in$ Si.

(a) Describe a bipartite graph G so that G has a perfect matching if and only if there is a feasible dinner schedule for the co-op.

(b) Your friend Alanis takes on the task of trying to construct a feasible dinner schedule. After great effort, she constructs what she claims is a feasible schedule and then heads off to class for the day. Unfortunately, when you look at the schedule she created, you notice a big problem. n - 2 of the people at the co-op are assigned to different nights on which they are available: no problem there. But for the other two people, p_i and p_j, and the other two days, d_k and d_l, you discover that she has accidentally assigned both p_i and p_j to cook on night d_k, and assigned no one to cook on night d_l. You want to fix Alanis's mistake but without having to recompute everything from scratch. Show that it's possible, using her "almost correct" schedule, to decide in only $O(n^2)$ time whether there exists a feasible dinner schedule for the co-op. (If one exists, you should also output it.)

Solution:

We have labelled the people $p_1, p_2, \ldots\ldots\ldots, p_n$ and the nights are labelled as $d_1, d_2, \ldots\ldots, d_n$.

a) We assign an edge for each person p_i to all the nights d_j in which a person is available to cook. The edge starts at p_i and ends at d_j. Each edge has weight one since each person needs to be assigned a night. If there is a perfect matching, then and only then each person can be assigned a unique night to cook and vice-versa.

b) All people are assigned a night to cook in which they are available, but by mistake two people p_i and p_j have been assigned to night d_k, instead of being assigned to d_k and d_l. An approach to fix this problem is as follows:
We can surely believe that both p_i and p_j have been assigned to day d_k. So, with this belief, we assume that p_i will be allotted to some other night and p_j is kept allotted to d_k.

1. Check if p_i can be allotted to d_l. If the person can be allotted, then we terminate here with the previous matching and new allotments of p_i to d_l and p_j to d_k.
2. If the allotment is not possible, then we search for all the persons. If $\exists\ p_t$ who can be allotted to d_l and p_i can be allotted to d_t; where d_t is the night to which p_t had previously been assigned. If such an assignment is possible, then we terminate here with the schedule where p_t being allotted to d_l and p_i to d_t and set remaining as previous.
3. If again no such assignment could be done, perform the same task with keeping p_i assigned to d_k and trying to find an allotment for p_j.
4. If p_j can be assigned to d_l, then terminate.
5. Otherwise we have to repeat Step 2 to find a p_t whose allotted night can be exchanged with d_l. If no such allotment could be found we terminate with the allotment of p_i to d_k and p_t to d_l and rest remaining the same.
6. If no such assignment is done, we conclude that a matching could not be found.

The complexity of the process is $O(n^2)$ since $O(n)$ time is required to find d_l from n different points and then if this fails, then again $O(n)$ is required to find p_t from again n different points. So, we get a time complexity of $O(n^2)$.

16. Back in the euphoric early days of the Web, people liked to claim that much of the enormous potential in a company like Yahoo! was in the "eyeballs"—the simple fact that millions of people look at its pages every day. Further, by convincing people to register personal data with the site, a site like Yahoo! can show each user an extremely targeted advertisement whenever he or she visits the site, in a way that TV networks or magazines couldn't hope to match. So if a user has told Yahoo! that he or she is a 20-year-old computer science major from Cornell University, the site can present a banner ad for apartments in Ithaca, New York; on the other hand, if he or she is a 50-year-old investment banker from Greenwich, Connecticut, the site can display a banner ad pitching Lincoln Town Cars instead. But deciding on which ads to show to which people involves some serious computation behind the scenes. Suppose that the managers of a popular Web site have identified k distinct demographic groups $G_1, G_2, \ldots\ldots, G_k$. (These groups can overlap; for example, G_1 can be equal to all residents of New York State, and G_2 can be equal to all people with a degree in computer science.) The site has contracts with m different advertisers, to show a certain number of copies of their ads to users of the site. Here's what the contract with the i^{th} advertiser looks like. For a subset $X_i \subseteq \{G_1, G_2, \ldots\ldots, G_k\}$ of the demographic groups, advertiser i wants its ads shown only to users who belong to at least one of the demographic groups in the set X_i. For a number r_i, advertiser i wants its ads shown to at least r_i users each minute. Now consider the problem of designing a good advertising policy— a way to show a single ad to each user of the site. Suppose at a given minute, there are n users visiting the site. Because we have registration information on each of these users, we know that user j (for j = 1, 2, ….., n) belongs to a subset $U_j \subseteq \{G_1, G_2, \ldots\ldots, G_k\}$ of the demographic groups. The problem is: Is there a way to show a single ad to each user so that the site's contracts with each of the m advertisers is satisfied for this minute? (That is, for each i = 1, 2, ……, m, can at least r_i of the n users, each belonging to at least one demographic group in X_i, be shown an ad provided by

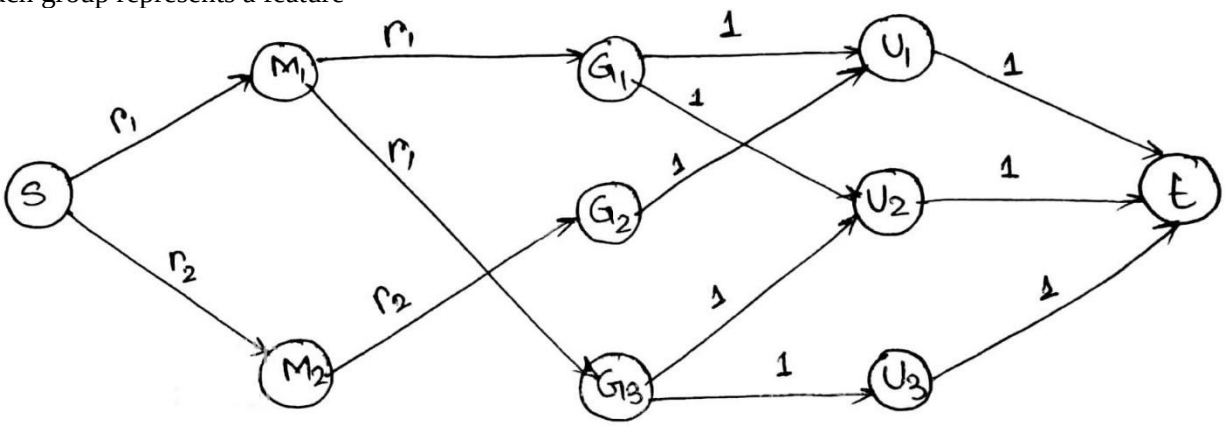
advertiser i?) Give an efficient algorithm to decide if this is possible, and if so, to actually choose an ad to show each user.

Solution:

Let X = Set of all groups, M = Set of advertisements and U = Set of users

Constraints:

1. At least r_i advertisements of M_i th company are shown
2. Each user can see at most one advertisement at a time
3. $X_i \subseteq X$ are the groups that i th company will show an advertisement
4. Each user may belong to more than one group
5. Each group represents a feature



Let s and t be the source and destination vertex respectively. It is a maximization problem where companies need to maximise advertisements. Thus, all companies are connected to source vertex s with capacity, $c(s, U_i) \geq r_i$. All the companies M_i are connected to multiple groups to which it wants to display its advertisement with each edge having capacity r_i because each of the groups can be shown all the advertisements of the company. Now, all the users U_j are connected to destination t with capacity one as each of them can see only one advertisement. Now, each group G_k is connected to multiple users U_j with capacity one, as each user can see one advertisement only. Thus, we can map the given problem into a network flow problem and obtain maximum flow in the graph using Ford Fulkerson Algorithm.