CS 218 : Design and Analysis of Algorithms

Lecture - 01

Topic: Introduction to Network Flow

Bipartite Matching

Remember the last problem in the midsem ?? To be honest, I don't :) . But the solution is something which uses Bipartite (Yes, that same stuff we studied long sem ago.) Just a recap :A bipartite graph is a graph where the vertices can be divided into two disjoint sets such that all edges connect a vertex in one set to a vertex in another set. There are no edges between vertices in the disjoint sets. AH, now I remember the question, it asks us to find an assignment of days to the requests. Motivation behind the usage of Bipartite is completely because of the typical constraints. Other example (which uses Bipartite) is to assign jobs to people where a person can do multiple jobs (c'mon, not simultaneously).Note that in such a valid assignment, all the vertexes need not be included in the matching inorder to meet all the constraints. (Why ? what if all the people are only for a single job ! skill issue right ..!!) .

However, the midsem problem (Sorry, I couldn't think of another name to the question so I named it " The midsem Problem") isn't trivial and greedy doesn't work. { Atleast Prof Gurjar didnt find a greedy solution.} The goal in our Bipartite matching is to maximise the no. of edges in our matching, Whereas the last part of the problem which has " prices " involved has a greedy solution !! Define a boolean function which returns true iff we can assign distinct days to all intervals in S. This function takes List S as an argument and sorts S with increasing order of ending days and assigns the first available day. If no day is available for some iterator in S return false. Now this serves as the subroutine to the Price variation. Sort with decreasing order of prices and initialise set S (*null set*) . and for each $r \in L$, if is_Schedulable(S+r) is true then add r to S. We're done ! with the midsem as well as with the solution UFFFF!

Network flow problem

The typical Network flow problems involve Traffic flow, Electric Network, Waterflow Network. The objective is to get the maximum flow.

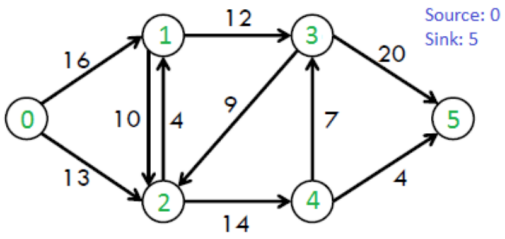


Figure 1: Netork Flow Problem

The problem statement takes following as input
(i) Directed Graph G (V,E)
(ii) Capacities on edges { $C_e : e \in E$ }
(iii) a Source vertex (S)

(iv) S Sink vertex (t) { derived from target }
Flow for a network is simply the directed graph that shows any valid assignment of flow.

$$f_e : e \in \text{E and } \forall e \in \text{E and } 0 \leq f_e \leq C_e$$

f_e can't be negative since reverse flow isn't accepted.

Flow conservation constraint: No flow is generated at any vertex other than Source (s) and sink (t) { target actually holds a negative generation }

$$\text{i.e } \forall v \in V - \{S, t\}$$

$$\sum_{e \in in(v)} f_e = \sum_{e \in out(v)} f_e$$

Objective: To maximise the outward flow.
Assumptions: No incoming edges in S and outgoing edges in t.
Claim: $f^{out}(S) = f^{in}(t)$. This can be generalised, for any subset of V ,say U, s.t $U \subseteq V$ and $S \in U$
,

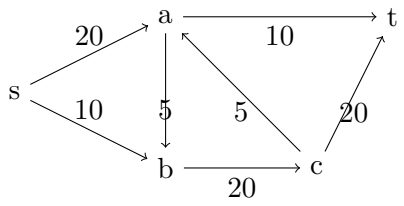
$$\sum_{v \in U} f^{out}(v) - \sum_{v \in U} f^{in}(v) = f^{out}(S)$$

Lecture - 02

Topic: Continuing Network flow problem

Example

Consider the below graph. Here, s is the start point(or source), and t is the sink, and the edges show the maximum flow that can be transfered through that pipe in respective direction.



Capacities

Definition

For any $U \subseteq V$ s.t $s \in U, t \notin U$, we define total capacity of traffic that can go out of U as $cap(U) = \sum_{v \in U, w \notin U} C_{vw}$

Claim

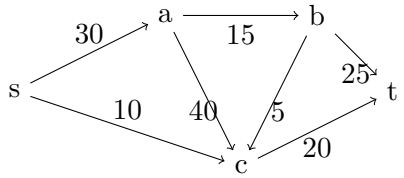
Max (s,t) cut flow $\leq$ min of all (s,t) cut's out capacity.
I.e, we say a (s,t) cut as a subset like $U \subseteq V$ s.t $s \in U, t \notin U$, i.e, some cut between nodes s and t. So, the net flow that can flow out(flow is equal for any such cut as seen is last class) is obviously less equals the individual cut's out capacity $cap(U)$. Thus, the max flow that can happen is less equals the $\min_{U \subseteq S}\{cap(U)\}$

Theorem

Max of (s,t) cut flow = min of (s,t) cut's out capacity.
We have seen its less equals in the above claim which was easier to get intuitively. But this is a bit hard and we'll prove it later.

Algorithm

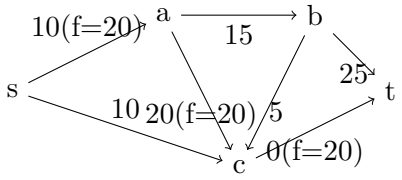
We see all the paths in the following case, and greedily choose the path as shown.



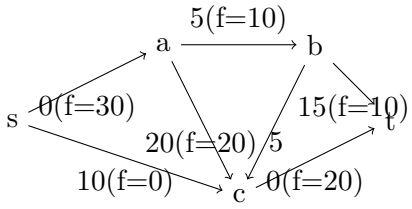
Lets greedily choose the path s,a,c,t with bottle neck flow of 20 (i.e, if network flows in that path, only a max of 15 can flow which is bottle necked by c-t edge). Now, we construct the residual graph

paths	bottle neck flow
s,a,b,c,t	5
s,a,b,t	15
s,a,c,t	20
s,c,t	10

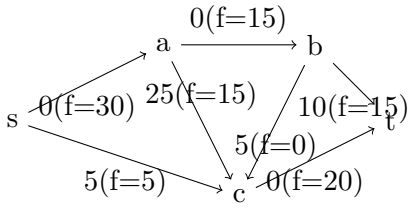
with the remining capacities that can be filled. Corresponding edge's current flow is depicted by (f=").



Now, repeating same as before, the max bottle neck is s,a,b,t path with bottleneck of flow 10.

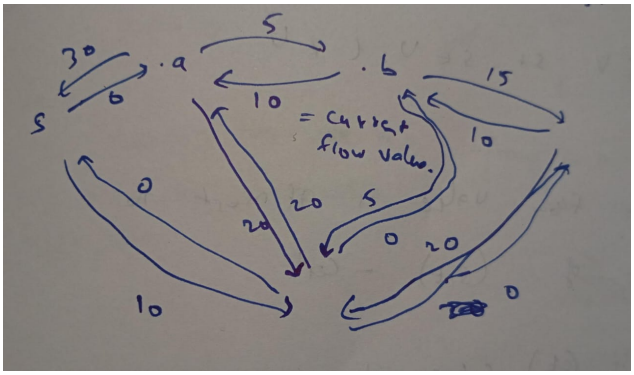


Now, there is no path between s and t and now, we stop the algorithm. We see that greedily we got the max flow as 30. But consider the following flow diagram where flow is 35.



So clearly this greedy method doesn't work, and we observe that, as compared to our solution, actual solution has extra iteration of path s,c,a,b,t with a back edge in a-c. i.e, we subtract a flow of 5 from edge a-c. So, from here, we get the idea of adding back edges in the residual graph to make our algo correct.

Here, we have the same residual graph as before, and just add back edges of max capacity values as the current flow value through that edge. It can also be viewed as *original capacity - current capacity*. So for the above residual graph, the modified one will look as follows.



Final Algorithm

- start with zero flow.
- while(you can [1])
  - find a path in the residual graph
  - push max possible flow along this path
  - update residual path

Here, we note that we stoped caring about the path with maximum bottle neck. From now on, we just find a path and proceed as thats a general case of the algorithm.

[1] till there are no path from s to t

Lecture - 03

Topic: Continuing Network flow problem

Claim 1

After every valid update, we have a valid flow i.e flow $\leq$ capacity and ≥ 0. This is easy to see as every time, the back flow is less equals the flow that is already flowing, and so net flow is always in the intended direction only. So its positive and also less equals capacity.

Claim 2

If the residual graph G_f has no (s,t) path then f is the max flow, i.e, if algorithm terminates, the attained flow is the maximum.
We know Max of (s,t) cut flow $\leq$ min of (s,t) cut's out capacity. So we'll show $f^{out}(s) = cap(U)$ and tell the maximum is attained.
The algo terminates when there is no path between s and t, i.e, there exists an s-t cut with no outgoing paths in the residual graph. Lets define U to be set of reachable nodes from S. This can be calculated by BFS(Breadth first search) or DFS(depth first search) methods. If $t \notin U$ at any point of time, then we terminate the algo. Clearly, U is an s-t cut. Thus,

$$f^{out}(s) = f^{out}(U) - f^{in}(U)$$

Claim 2.1: $f^{in}(U) = 0$. Assume its non-zero. Then we would have an out edge of the same value in the residual graph by our algorithm. But we know that there is no outgoing edge. Hence it can't be non-zero.

Claim 2.2: $f_e^{out} = C_e$. If not, there would have been an outgoing edge of capacity $C_e - f_e^{out}$. Again contradiction that there is no outgoing edge.

Hence, we can tell that,

$$\begin{aligned} f^{out}(s) &= f^{out}(U) - f^{in}(U) \\ &= \sum f_e^{out} - 0 \\ &= \sum C_e \\ &= cap(U) \end{aligned}$$

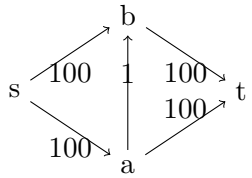
Hence proved the claim.

Claim 3

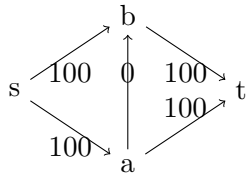
The algorithm terminates in finite steps.
If the capacities are integers, then $f^{out}(s)$ increases by at least 1 everytime, and thus we can say it terminated at some point. But with real valued capacities, its not required for it to terminate. In fact, there are such examples where the algo runs infinitely.

Running Time

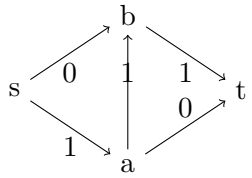
In each iteration, we need to perform DFS or BFS which is of order $O(|E|)$. We also need to modify the edges which is again of same order. Assuming the graph is of integral capacities, total number of iterations is $O(F_{max})$. Thus the total order is $O(|E|F_{max})$. By scaling the capacities, running time can be brought down to $O(|E||V|log(C))$ (This is polynomial, but not strongly polynomial as it depends on capacities). Strongly polynomial time was acheived by Edmonds Karp by choosing s-t path everytime with fewest number of edges and then proved that it takes only $O(|E||V|)$ iterations. So final order is $O(|E|.|E||V|)$.
See the following graph of capacities.



Here, clever choosing of s-t cut would be the following showing the flow.



But if its chosen inefficiently, like



flow

So here, the we increase the flow 1 by 1 and it would take much time before we get the maximum flow. So choosing the correct path matters.

Lecture - 05
Topic: P vs NP problems

NP and NP-completeness

For any algorithm, we generally try ideas and see if it works. If yes, success. If not, then try new ideas. But when to stop trying? Maybe we can check for NP-completeness. Eg- shortest path with negative weights is NP-complete problem and so we can give up on polynomial time algorithm.

History of polynomial time algorithm

Finding GCD using Euclid's algorithm is very efficient(polynomial time in its bit size) while trying out every factor is not polynomial. In 1950s some famous algorithms like greedy algos, Dynamic Programming, Simplex method, etc came up. These were found to be far better than brute force exponential time. Mathematicians at these times were focussed on whether the algorithm works or not, and whether it runs in finite number of steps or infinite. But Edmond emphasised on whether it takes polynomial steps or exponential. There might be different computatinal models like in one model, to go to 100th index in an array, it takes 100 units of time whereas in modern models, it takes the same time irrespective of the index. But polynomial time algos are polynomial irrespective of the computational model.

Roomate Allocation Problem

There are n students, and some like each other and some don't (Assume liking is symmetric like if A like B, B also likes A). So, we want to allocate such that they like each other. This was solved by Edmond in polynomial time algorithm. But, the same problem with triple occupency [2] is not solved yet.

Given a graph, is there a path of lenght k? (Not known to be in P time)
Given a graph with edges colored red or blue, is there an s-t path with alternating red and blue edges? (in P) . But same in directed graph is not know in P.
The underlying problem is that, a path should not have repeated veritces, and we don't have efficient algo to ensure that. Finding minimum lenght path by default takes care of this greedily. But here, greedy approaches don't work.

Given a number(in binary), is it factorisable? It can be done in P (solved by Sir's friends in IITK)

Travelling Salesperson problem(TSP)

Given a list of cities, we have to visit every city and come back to the same city with minimum cost - Not known in P.

Types of problems

- Decision problem: just we need to answer yes or no.

[2] There are pairs of people liking each other and we allot 3 in same room if everyone likes each other

- Optimization problem: max network flow problem. This comes with a Decision problem like is there any flow with value $\geq k$. If k is polynomial in size of input, we can binary search and so find in polynomial time provided that Decision problem is polynomial. So solving Decision problems is sufficent to solve Optimization problems in general.

These problems lead us defining a new class of problems as *Class NP*. A yes or no decision problem is in NP if there is an easy verifyable proof for each yes input. Mathematicians started to realize that these class NP problems are actually equivalent,i.e, if one problem is solved in polynomial time, many others equivalent of them are also solvable in P. Lets see one such example.

Equivalence between 3-colorability and a subroutine for SAT

3-colorability: Color the nodes of the graph such that adjacent nodes are not of same color.
SAT: Seen in cs208. Here, proof of satifyability is easy. Just give the satisfying assignment and can be checked. But its difficult to prove that something is unsatisfyable. (Similarly, even for 3-color problem, its easy to verify if its colorable, but difficult to say its not colorable). If we can solve SAT, can we also find its corresponding assignment? Yes. first assign x1 = 1 and check satifyability of remining. If its unsatisfyable, we know x1 = 0 and we proceed the same for remining variables. So like this, the assignment can be found in polynomial time.

Let's create boolean variables to represent coloring. 3 variables for each vertex x_i, y_i, z_i and we encode the problem using these variables. We put constraints as we've seen in cs208 to convert the problem to a boolean formula ϕ_G. So, here, we had a graph G, converted to a boolean formula ϕ_G which can be checked if satifyable or not using SAT algo. If yes, 3-colorability is true and false otherwise. Thus we can say both the problems are equaivalent.
Thus similarly, all the problems like 3-colorability, Load Balencing, TSP, Integer factoring, etc can be converted to SAT problem.
So, which all problems can be reduced to satisfiability problem? These are called NP problems. Actually, this is an alternate defenition of NP problem. Problem A is said to be NP if $A \leq SAT$.
Definition: A problem A is said to be in NP if there is a verification algorithm V such that,

- If x is a "yes" then there exists c such that size(c) $\leq$ polynomial(size(V)).
- If x is a "no" input then, $\forall_c V(x, c) = False$.

Problem in the defenition:
3-colorability : is the graph 3-colorable?
not 3-colrable : is the graph not 3-colorable.
The first is in NP, but the second is not though both are littrally the same just the complement of each other. So NP defenition if kind of vague.

SAT to IND-SET Reduction

Defenition of Independent Set: A set of vertices I in a graph is called independent if there are no edges between any two vertices in I. Optimization: Given a graph, find the largest independent set. Now, we need to prove that, IND-SET problem is NP-complete.
Before that, we see if the Given a graph G, its easy to check if I is independent or not. Given a graph G, and a number k, is there an independent set of size k?
We know that, SAT is a NP-complete i.e, its in NP, and any other problem reduces to SAT. To prove that, IND-SET is NP-complete, we prove for self reduction. **self reduction:**Finding an ind set of size k $\leq$ is there an independent set of size kind.

Is there an ind set of size k in $G - V_1$,
if yes, delete V_1 and proceed with $G - V_1$, and
if no, V_1+ Find an ind set of size $k - 1$ in $G - N(V_1)$.

Proof

We need to prove that,

1. it is in NP i.e, for any yes input, the proof is an independent set of size k. Eg- Is a given number prime? The verification step is difficult.

2. It is NP-hard i.e, any problem in NP reduces to IND-SET. But this is difficult right. We can't show it for every problem. So we show that SAT reduces to IND-SET, and because of transitivity of reducibility, we can say any problem will be reducible to IND-SET.

So, given a boolean formula ϕ, we need to find if its satifiable or not. We construct a graph G_ϕ and a number K_ϕ and try to say that, there is an ind set of size K_ϕ in G_ϕ iff ϕ is satisfiable. Then we would be done with the proof.

1. Convert ϕ to 3-CNF form, ϕ'.