MA859 - Selected Topics in Graph Theory

Lecture - 5

(TREES AND MATCHINGS)

Let us begin with some basic facts about the trees.

Theorem 1: Every tree with at least two vertices has a leaf.

Proof:- Take a longest path $x_0, x_1, \ldots, x_k$ in the tree.
($k \geq 1$ since the tree has at least two vertices).
A neighbour of x_0 can not be outside the path; otherwise,
the path could be extended.
But if x_0 were adjacent to x_i for some $i > 1$, then
$x_0, x_1, \ldots x_i x_0$ would be a cycle.

So, the only neighbour of x_0 is x_1 and x_0 is a leaf.
[The same argument holds for x_k and so, there are at least two leaves].

Distances and Breadth-First Search (BFS)

A spanning tree of a graph G is a subgraph T of G, which is a tree with $V(T) = V(G)$.
We now discuss a specific algorithm that gives a spanning tree with special properties, called a Breadth-First Search Tree or BFS Tree.

This algorithm answers various natural questions such as determining the distance between two vertices.

Defn: Let G be a graph. For two vertices $x, y \in V(G)$, the distance $d(x,y) = d_G(x,y)$ is the length of a shortest path between x and y.

The diameter of G is the length of the longest shortest path:

$$\text{diam}(G) = \max_{x,y \in V(G)} d(x,y)$$

Given a graph G and a subgraph H of G, we define

$\partial(H) = \{ xy \in E(G) \mid x \in V(H),\ y \notin V(H) \}$, for the set of edges going from the vertices of H to vertices not in H.

To single out a vertex of a tree, we call it a <u>root</u>.

BFS Algorithm:

Given a graph G and a root $r \in V(G)$,

(1) Let T be the graph consisting only of r;

(2) Iterate:
   (a) Set $S = \partial(T)$;
   (b) For all $xy \in S$, if $T + xy$ does not contain a cycle, replace T by $T + xy$;
   (c) If $\partial(T) = \emptyset$, then go to (3).

(3) If $|V(T)| = |V(G)|$, return T;
   else, return "disconnected".

Theorem 2: Let G be a connected graph. The BFS algorithm lets us find the shortest path between any two vertices in G.

Proof: Let $r, s \in V(G)$ be the vertices between which we want to find the shortest path.

Run the BFS algorithm with root r. Give a vertex x the label $f(x) = k$ if x was added to T in the k^{th} iteration of step 3 (we begin with $f(r) = 0$). We prove, by induction on $f(x)$, that $f(x) = d_T(r, x) = d_G(r, x)$ for all x.

It trivially holds for $x = r$.

If $f(x) = k$, then by construction, x is adjacent (in T and hence in G) to a vertex y with $f(y) = k-1$, and by induction, we have $d_T(r, y) = d_G(r, y) = k-1$.
So, there is a path (in T & in G) of length $k-1$ from r to y, which gives a path (in T & in G) of length k from r to x.
Moreover, there can not be a shorter path in G from r to x because otherwise, $d_G(r, x) < k$; so by induction, we would have $f(x) = d_G(r, x) < k$, a contradiction.

Knowing that $d_{T_0}(r, x) = d_G(r, x)$, we can find the shortest path from r to x in G by finding the shortest path in T_0. To do this, we need to keep track of the "predecessor" of each x that we add in Step 3; that is, the vertex p with $f(p) = f(x) - 1$ that x is connected to.

Then, to find the shortest path from r to s, we start from s, and backtrack repeatedly through successive predecessors, until we reach r. //

Apart from finding the shortest paths, determining if G is connected or not and finding the components of G, the BFS algorithm lets us compute the distances.
It also gives algorithms for

* Compute diam(G): For each pair of vertices, we can find a shortest path; thus the distance. Hence the largest distance can be determined.
* Find a shortest cycle in G: For every edge xy, find a shortest path between x and y in G−xy (if it exists); combine this path with xy to get a cycle. Compare all these cycles to find the shortest.

Some of these algorithms are very inefficient; but they are better than the brute force approaches that go over all possible answers. (There are all kinds of algorithms that do these tasks faster; but at this point, our focus is not on efficiency; but rather on the graph theoretical aspects).

The problem of finding a shortest path is different, if we have a weight function $w : E(G) \to R_+$ and we want a path P that has $\sum_{e \in P} w(e)$ minimal.
The BFS algorithm does not do this because it finds a path with the fewest edges, whereas a minimum-weight path uses only the light weight edges.
Nevertheless, the best known algorithm, known as "Dijkstra's Algorithm", is based on this BFS approach.

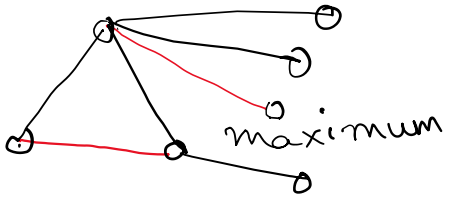
Matchings and Augmenting Paths

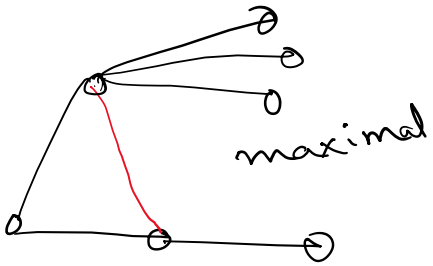
Def'n: Let G be a graph. A set of edges M is a *matching* if no two edges in M have a common vertex.

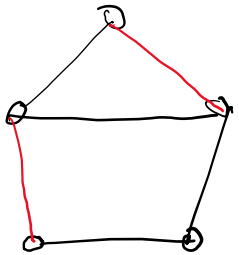
A matching M is *perfect* if every vertex of G is incident with an edge in M.

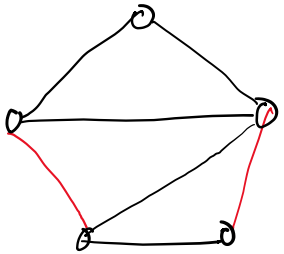
A matching M is *maximal* if there is no matching M' in G with $M \subseteq M'$. It is *maximum* if there is no matching M' in G such that $|M| < |M'|$.

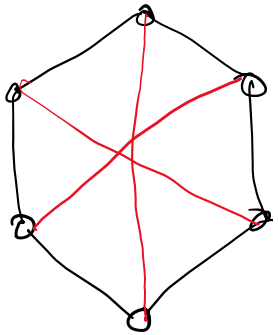
Examples:

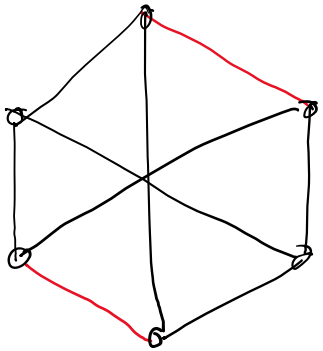




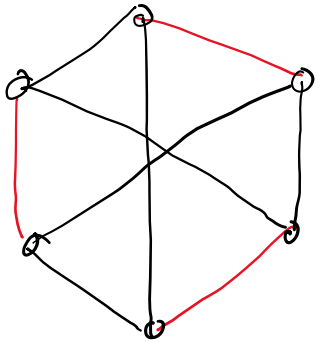








not maximum
/maximal

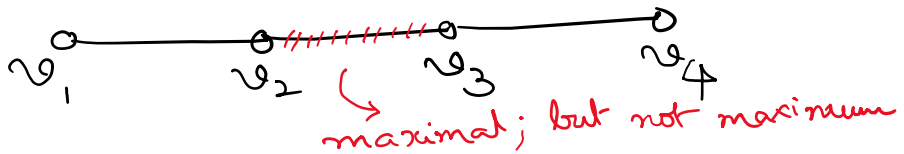


(*) How can we find a maximum matching?

(*) How can we tell if a graph has a perfect matching?

(*) How can we check if a given matching is maximum?

A maximal matching need not be maximum.
Consider the simple example:



Similarly, a greedy approach that keeps adding edges, without removing any, does not necessarily lead to a maximum matching.
Instead, such an algorithm to find maximum matching will not need an opportunity to backtrack, where we can throw away some edges that we have previously selected.
The notion (in the next slide) helps us do it in a smart way:

Def: Given a matching M in a bipartite graph G, a path is alternating if for every two consecutive edges on the path, one of the two is in M.

Note that any path of length zero or one is alternating.

An alternating path with at least one edge is augmenting if its first and last vertex are not matched by M.

Lemma 1 A matching M is maximum if and only if there is no augmenting path for M.

Proof: We prove that M is not maximum iff there is an augmenting path for M.

Firstly, suppose there is an augmenting path P for the matching M, say $P: v_1 v_2 \cdots v_k$ and the matching edges are $v_2v_3, \cdots, v_{2i}v_{2i+1}, \cdots, v_{k-2}v_{k-1}$ (k must be even).

Then we can augment M, by removing these edges from M and replacing them by $v_1v_2 \cdots v_{2i-1}v_{2i} \cdots v_{k-1}v_k$.

More formally, we replace M by

$$M' = M \Delta E(P) \left[= (M - E(P)) \cup (E(P) - M) \right].$$

Then M' is a matching since v_1 and v_k were unmatched by M, and we have $|M'| > |M|$; so, M is not maximum.

Conversely, suppose M is not maximum. Then there is a matching M' with $|M'| > |M|$.
Let D be the path $V(D) = V(G)$ and $E(D) = M \Delta M'$.
Every vertex has degree 0, 1, or 2 in D.
$\Rightarrow$ the (connected) components of D are either cycles, paths or isolated vertices.
A cycle in D has the same no. of edges from M as from M'; so $|M'| > |M|$ implies that

there is a path P in D with more edges from M' than from M. Then P must be an augmenting path for M. //

Note: The above lemma actually works in any graph (if we define matchings in arbitrary graphs in the obvious way).

Using this lemma, we have the following algorithm. It looks for an augmenting path and augments on it until it is no longer possible:

Augmenting Path Algorithm

To find a maximum matching in a bipartite graph G with bipartition $V(G) = A \cup B$.

(1) Set $M = \phi$

(2) Iterate:
- (a) Set $S = A - (\cup M)$, $T = B - (\cup M)$;
- (b) Find any alternating path P from S to T; if none exists, go to (3)
- (c) Augment along P using $M := M \Delta E(P)$; go back to (2);

(3) Return M.

It is not specified how to find alternating paths; but for bipartite graphs, this is fairly simple, by considering only non-M-edges from A to B and M-edges from B to A. Any alternating path from S to T must be of this form, since it starts from an unmatched vertex in S.

Note that for non-bipartite graphs, it is not at all clear how to find all alternating paths and in view of this, it is much harder to give an algorithm that finds maximum matchings in general graphs. (even though the lemma actually holds good for general graphs).

Stable Matchings

Hall's Theorem

This theorem gives a necessary and sufficient criterion for a bipartite graph to have a perfect matching.

We say that a matching *matches* $A \subseteq V(G)$ if every vertex in is contained in an edge of the matching.

Given a set $S \subseteq V(G)$, we define the neighbourhood of S as

$N(S) = \{v \in V(G) - S \mid \exists\ u \in S$ such that $uv \in E(G)\}$

Statement of Hall's Theorem: Let G be a bipartite graph with bipartition $V(G) = A \cup B$. Then G has a matching that matches A if and only if for every $S \subseteq A$, we have

$|N(S)| \geq |S|$.

Proof of this result is too long and hence is omitted from discussion here.

Note: Hall's condition is not a convenient way of testing if a given graph has a perfect matching, because it requires checking something for all subsets of the vertex set.

Practically, it is better to run the augmenting path algorithm.

Konig's Theorem

We already know an algorithm to find a maximum matching; we will now look for a more direct approach of verifying that a given matching is maximum.

Defn: Given a graph G, a vertex cover for G is a set $C \subseteq V(G)$ such that every edge of G is incident with a vertex in C.

Statement of Konig's Theorem

Let G be a bipartite graph. The maximum size of a matching equals the minimum size of a vertex cover.

[size of a matching is the no. of edges in it;
size of a vertex cover is the no. of vertices in it.]

Proof (omitted)

Konig's Theorem is a special case of the duality theorem of linear programming.

Suppose that we have n students to be allocated internships in n companies. Every student has sent applications to all the companies. Naturally, each student and so also, each company has a list of preferences.

We want to pair up so that this assignment is stable in the following sense:

"There is no student and company that would both prefer to work with each other to their assigned pairs."

More formally, consider a bipartite graph G with parts A and B, where $|A| = |B| = n$, and in which each vertex has a (strict) order of preferences for the vertices of the other part.

We say that a perfect matching is stable if there is no pair $a \in A$ and $b \in B$, such that both of them would prefer the other to the vertex they are currently matched to.

The algorithm due to Gale and Shapley allows to construct such a stable matching.

Gale - Shapley Algorithm

To find a stable matching M in a complete bipartite graph with bipartition $V(G) = A \cup B$ where $|A| = |B|$.

(1) Set $M = \phi$

(2) Iterate:

(a) Take an unmatched vertex $a \in A$ and let $b \in B$ be the vertex that a prefers among the ones a has not tried yet.

(b) a "proposes" to b: If b is unmatched or b is matched to a'; but prefers a over a', then "accept" a and reject a': put $M := M - a'b + ab$.
otherwise, "reject": Leave M unchanged.

(c) If there are no more unmatched vertices in A that have someone left on the list; then go to (3);

(3) Return M.

(The complexity of this algorithm is $O(n^2)$).

Theorem The matching M that G-S algorithm outputs is stable.

Proof: First we shall show that M is perfect. Indeed, if there is a pair of vertices $a \in A$, $b \in B$ such that both are not matching, then a must have proposed to b at some point. However, if a vertex $b \in B$ is in M at some step of the algorithm, then it stays in M.

Next, we will show that the matching is stable. Assume that $ab \notin M$. Upon completion of the algorithm, it is not possible for both a and b to prefer each other over their current match.

If a prefers b to its match, then a must have proposed to b before its current match. If b accepted its proposal; but is matched to another vertex at the end, then b prefers

the current match over a.

If b rejected the proposal of a, then b is already matched to a vertex that is better for b. //

We now show that this algorithm always yields the same matching. We will prove that each $a \in A$ ends up with the best possible match in concrete sense.

We will say that $b \in B$ is a <u>valid match</u> for a if there is a stable matching that contains the edge ab.

We say that b is the best/worst valid match of a if b is a valid match, and no $b' \in B$ that ranks higher/lower than b on the list of a is a valid match for a.

(We use the same notation for $b \in B$).

Let S^* denote the set of edges $\{\{a, best(a)\} : a \in A\}$.

Theorem The Gale-Shapley algorithm always outputs S^*.

Proof: Assume that some execution of G-S algorithm results in a matching S in which some $a \in A$ is paired with $b \in B$; but b is not its best valid match.

Since a's propose in the decreasing order of preference, this means that some a is rejected by a valid match during the execution of the algorithm.

So, consider the first moment during the execution in which a vertex of A, say a, is rejected by a valid match b. Since this is the first reject of a valid match, we must have b = best(a).

Irrespective of how the rejection happened, after this step, b is paired with a', which is preferable for b.

Since b is a valid partner of a, there exists a stable matching S', containing the edge ab.

In S', the vertex a' is matched to b'. Since the rejection of b by a was the first rejection, and a' proposes in the decreasing order of preference, we see that a' prefers b over b'. Therefore, in S' both a' and b would prefer to change their match, which means S' is not stable, a contradiction. //