Discrete Math Lecture 9

1 More about Trees

A very useful fact about trees was stated in the last lecture without proof. We will now prove this fact.

Theorem 1. *A tree on n vertices has exactly $n-1$ edges.*

Sketch. The proof is by induction. The tree on 1 vertex has 0 edges, so the base case is true.

For the inductive step, consider a tree T with n vertices. Remove one of the edges, say the edge between u and v. Prove that now there cannot be any path between u and v so the resulting graph has at least two connected components. Further prove that removing this one edge cannot have split the graph into more than 2 connected components. Use these facts to prove that removing the edge (u, v) must have split the original tree T into two trees T_1 and T_2. Each vertex of the original tree must be in exactly one of T_1 or T_2, because two connected components cannot share any vertex in common (prove this) and we did not remove any vertex from the original tree. Say T_1 has p vertices and T_2 has q vertices. By IH, T_1 has $p-1$ edges and T_2 has $q-1$ edges. So T in total has $(p-1)+(q-1)+1 = (p+q)-1 = n-1$ edges, which completes the proof. □

Excercise 1. *In fact a connected graph on n vertices is a tree precisely when it has exactly $n-1$ edges. Prove that if a connected graph has exactly $n-1$ edges, then it must be a tree.*

Let's prove a few more things about trees in general.

Theorem 2. *All trees are bipartite graphs.*

Proof. (Sketch) Given a tree fix any one vertex and call it R. Note that there is a unique path from R to any other vertex v of the tree (Why?). Thus we can define a notion of distance between R and any other vertex v which is just the length of the path from R to x. Argue that the vertices that are at an even distance from R and the vertices that are at an odd distance from R can be considered as the two disjoint vertex sets of a bipartite graph and there can be no edges between vertices in the same set. □

However not all bipartite graphs are trees!

Excercise 2. *Draw a graph that is bipartite but is not a tree.*

Recall we stated last time that bipartite graphs are exactly those graphs which are two colourable. Thus all trees are two colourable!

Theorem 3. *A graph is bipartite if and only if it is two colourable.*

Proof. (Sketch) Suppose the graph is two colourable. Then consider any valid 2 coloring. Put all the vertices with the one color in one set and all the vertices with the other color in the other set. Every vertex has a color so it goes into exactly one of the two sets. But there cannot be any edge between vertices in the same set or else the coloring would not have been a valid two colouring.

For the other direction, suppose a graph is bipartite. Then it's vertices can be partitioned into two disjoint subsets A and B such that every vertex goes into one of the two subsets. Furthermore there are no edges within either subset, so we can color all vertices in A red and all vertices in B blue. □

Excercise 3. *Given a tree how might you quickly find a two coloring of it? Try to come up with a simple algorithm to do this.*

So far we have been talking about trees as just any connected acyclic graphs. However, as we briefly mentioned last time, computer scientists often care about rooted trees. These can also be viewed as connected acyclic graphs, but with a special vertex, called the root. Once you have a root vertex, and the graph is acyclic, there is a very natural notion of direction that you can associate to the edges: the root is at the top and every edge takes you further and further away from the root. However in the usual computer science usage we do not actually model trees as directed graphs, the edges are still undirected and you can traverse them in either direction: for example in a tree data structure you are normally able to go from the child nodes to the parent node. Still it is sometimes useful to keep this natural sense of direction in mind.

Notice that you can pick any vertex of a tree and make it the root, this just corresponds to redrawing the same tree. Try it! Redraw the figure below with some other vertex as the root.

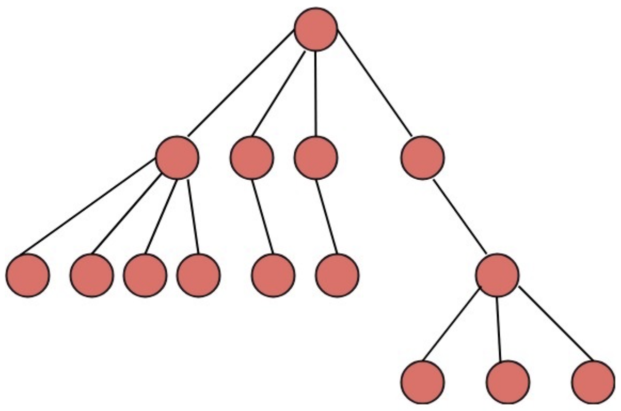


Figure 1: A typical tree in computer science.

We can define trees recursively: indeed many algorithms on trees are recursive in nature. Today we will define rooted trees, more specifically rooted binary trees, in an inductive/recursive fashion and prove some basic properties of these structures by using induction. Henceforth whenever we talk about binary trees, by default we will mean rooted binary trees.

Definition 1 (Rooted binary tree)**.** *A rooted binary tree can be defined via a recursive definition to be either:*

- *Base case: An empty node or*
- *Recursive Case: A node (vertex) with a left pointer to a binary tree and a node with a right pointer to a binary tree.*

Often while drawing the tree we don't draw the empty nodes at all. So a typical picture of a tree might look something like this.

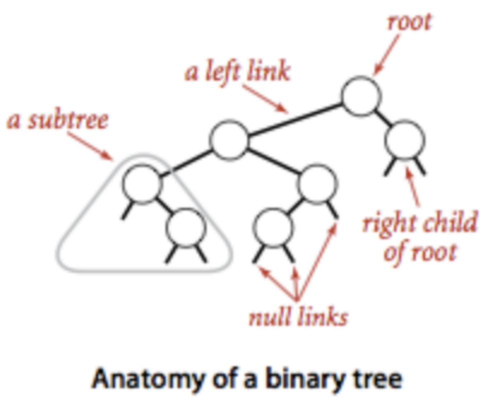


Figure 2: (Illustrating some terms about binary trees)

Typically the topmost vertex is called the root and each vertex has two two child vertices (a left child and a right child, one or both of which may just be empty), which may themselves be roots of binary trees. The subtree rooted at the left child of a vertex is called the left subtree and the subtree rooted at the right child of a vertex is called the right subtree.

Based on how far vertices are from the root, a binary tree's vertices can be divided into levels. A complete binary tree is one where all but the last level of vertices has been filled out and the last level of vertices is filled from left to right. It looks something like this:

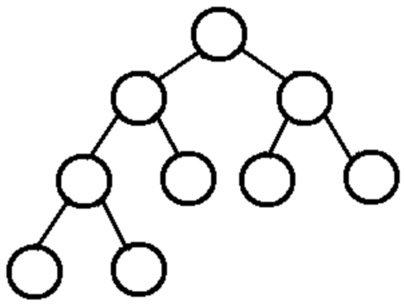


Figure 3: A complete binary tree

The nodes which have no children, are called leaves. The height of a binary tree is defined as the length of the longest path from the root of the tree to any leaf.

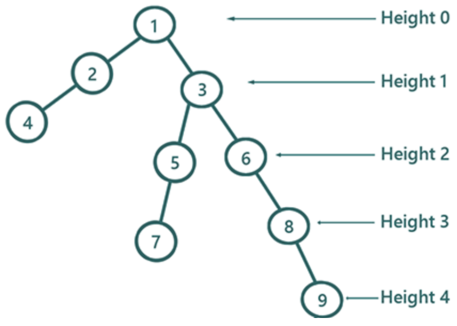


Figure 4: The height of a tree

We can reason inductively about binary trees, which makes sense since they are recursively defined. Here is an example. Let us first define full binary trees. These are binary trees where every node has 0 or 2 children. This is what a full binary tree might look like.

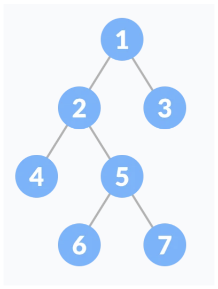


Figure 5: An example of a full tree

Theorem 4. *The number of leaf nodes of a full binary tree is always exactly one more than the number of non leaf nodes of the tree.*

Sketch. It is easy to check the base case where there is just one node, that is a leaf and there are 0 non-leaf nodes.

For the recursive case: The full tree has a root with a left subtree and a right subtree. Both left subtree and right subtree have smaller number of vertices than the whole tree and they by themselves are full trees; so, inductively, they each have one more leaf than non leaf. But when I look at the whole tree, it's leaves are exactly the leaves of both left and right sub tree, so there are now two more leaves than non-leaves! We do however have to add one non-leaf to account for the root of the full tree itself, leading to there being exactly one more leaf in the original full tree than non leaf vertices. □

Excercise 4. *We already know that any tree has exactly one fewer edges than vertices. Try to re-prove this fact specifically for binary trees via induction. The proof is quite straightforward (a sketch of this proof is available in the notes for lecture 2 notes as well).*

These definitions can be made for trees other than binary trees as well, you could have a recursive definition where either the tree is an empty node or it has a node with some larger number of children rather than just two children. However, since binary trees come up the most often in computer science, we've focused on them for the sake of this class.

2 Eulerian and Hamiltonian Cycles

Euler was famously asked the following question about the city of Konigsberg: The city has a bunch of rivers and bridges over the rivers; is it possible to do a tour of the city, starting somewhere and eventually returning back to the same point, such that you walk across each bridge exactly once?

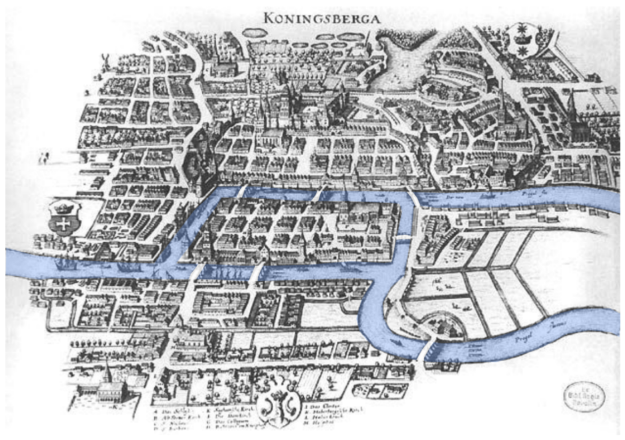


Figure 6: The map of Konigsberg

If we simplify the map a little bit, it looks something like this.

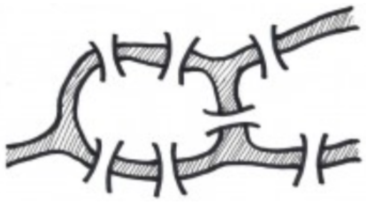


Figure 7: The map of Konigsberg (simplified sketch)

Unfortunately, if we want to model this as a graph with the bridges playing the role of edges, then we get a graph with multiple edges between vertices. However we can ask the same question about a simpler version of this graph (dropping the repeated edges). For the following graph:

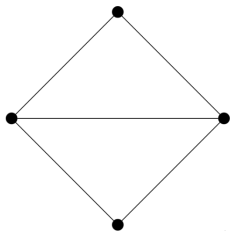
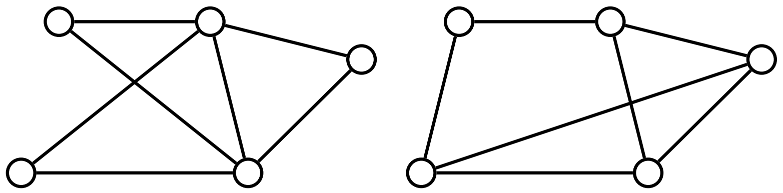


Figure 8: Simplified graph of Konigsberg

1. Is there a walk, starting and ending at the same vertex, that uses each edge exactly once? Why/why not?
2. What about if we drop the requirement that the walk should start and end at the same vertex?

You should be able to convince yourself that there is no such walk on this graph where you have to begin and end on the same vertex, but if you are allowed to begin and end on different vertices, then such a walk is possible.

What about for the following graphs?



In general, for what kinds of graphs is there a walk starting and ending at the same vertex, that uses each edge exactly once? What about if we drop the requirement that the walk should start and end at the same vertex?

Definition 2 (Euler tour)**.** *A closed walk in a graph that starts at some vertex and returns to that same vertex at the end of the walk and also goes through each edge exactly once is called a Euler tour.*

Playing around with a bunch of different examples might lead you to the intuition that for there to be a Euler tour in a graph, every vertex must have even degree. And indeed this is exactly the necessary and sufficient condition for a connected graph to have a Euler tour.

Theorem 5. *A connected graph has a Euler tour if and only if all of it's vertices have an even degree.*

The intuition really is that we have to leave a vertex and then eventually arrive at it again via a different edge and so on...and there need to be an even number of edges for each vertex so that we can arrive and leave (possibly repeatedly) from each vertex without getting "stuck". Let's now try to formalise this intuition a little more.

Proof. (Sketch) One direction is relatively easy. If the graph has a Euler tour then no vertex can have odd degree. Why?

Suppose G contains an Euler tour. Consider this tour. This is a walk that starts and ends at the same vertex. Imagine the walk going through a vertex which is not the starting vertex. The walk must traverse two edges incident to this vertex: one going in, and one going out. So for any vertex in the interior of our walk (that is except the start vertex), we need to have traversed an even number of edges, and therefore the vertex will have even degree. But since the graph is connected and the walk contains all edges, it must also contain all vertices. So except for the starting vertex, we have shown that all vertices have to have even degree. But what about the starting vertex? Our walk has to end at the starting vertex, so that gives us that even the starting vertex of the walk has to have an even degree.

Now let's prove the other direction: if the degree is even for every vertex in the connected graph, then the graph has a Euler tour.

Consider a maximal (longest) walk $W = (a, \ldots, b)$, where there are no repeated edges, in a connected graph G, where each vertex has even degree. Look at the endpoint a, which like all other vertices has even degree. Because W is maximal, a 's neighbors must be in W (otherwise we could extend the walk).

Moreover, for each neighbor x of a, each edge (x, a) must be in W in order for the walk to be maximal. Because a is the starting point, and its even number of edges are all in the path, the only way for all of them to be traversed is for W to end at a since we can only traverse each edge once. Therefore $a = b$ and W is a walk that starts and ends at the same vertex.

Now, to complete the proof via contradiction, assume W is not an Euler tour. Then there is an edge (x, y) that is not in W. Since G is connected, there is a path from y to some vertex v in W, which is $(y, \dots, v)$ Pick such a path that does not include any edges already in W, the moment you reach something in W, that is the path you consider. Create a walk $(x, y, \dots, v, \dots, a, \dots, v)$, which exists because W is a walk that has the same start and end point a.

This walk is clearly longer than W, which contradicts the assumption that it is a maximal path. So actually W must have been a Euler tour.

Note that if y was already on the walk, that is fine, you don't need any path at all just consider the walk $(x, y, \dots, a, \dots y,)$, y itself plays the role of v here. □

To address a point raised in class, when we prove that W must not have any missing edges, we already know W must start and end at the same vertex. If we assume W has some missing edge, then we can construct a longer walk than W. This a contradiction; it doesn't matter if this longer walk starts or ends at the same vertex or not.

What if we do not need to start and arrive at different vertices. We will call such a walk a Euler walk. (Note that sometimes people use the term Euler walk for what we call a Euler tour, but let's stick to this notation). Then it's ok for at most two of the vertices to have odd degree. We can use these as the start and end vertices so we don't need to pair up every departure at the start vertex with an arrival via a different edge, and similarly we end up arriving at the final vertex, so we don't need to pair every arrival at the final vertex with a corresponding departure via a different edge. This intuition suggests the following theorem, which can be proved in a similar manner. We skip the details of the proof for now.

Theorem 6. *A connected graph has a Euler walk if and only if at most 2 of its vertices have an odd degree.*

You might object that intuitively if exactly one vertex has an odd degree, that would be an issue for having a Euler walk. But actually this is a complete non-issue. (Think about why. Can a graph really have exactly one vertex with an odd degree?)

Similar to Euler tours, where we wanted to visit each edge exactly once, we might instead want to visit each vertex exactly once.

Definition 3 (Hamiltonian Path)**.** *A Hamiltonian path in a graph is a path where each vertex in the graph occurs exactly once.*

Definition 4 (Hamiltonian Cycle)**.** *A Hamiltonian cycle is a cycle where each vertex in the graph occurs exactly once.*

While we know exactly what sorts of graphs have Euler Tours, the case is less simple when it comes to Hamiltonian Cycles. Deciding if a graph has a Hamiltonian cycle is in general a computationally hard problem. We don't know an exact characterisation of graphs that have such cycles, but we do know some sufficient conditions for this to be the case.

Theorem 7 (Dirac)**.** *Let $G = (V, E)$. If $\deg(v) \geq \frac{|V|}{2}$ for all $v \in V$, then there exists a Hamiltonian cycle.*

Proof. (Sketch of a part of the proof) The proof is by an explicit construction, that is, we show that if G satisfies the condition in the theorem that we can construct a Hamiltonian cycle in G.

The idea is to pick some vertex v_1 arbitrarily and gradually extend a path P starting from v_1, say $P = v_1 v_2 \dots v_k$, where all vertices v_j are different. Eventually, if $k = n$, P will be a Hamiltonian path.

Initially, $P = (v_1)$. Suppose that we have already constructed $P = v_1 v_2 \ldots v_k$. We now show that as long as $k < n$ we can always extend P.

If v_k has a neighbor $u \in V$ that is not on P, then it is easy to extend P, for we can simply append u at the end of P. In other words, we can take $v_{k+1} = u$ and the new extended path will be $v_1 v_2 \ldots v_k v_{k+1}$.

The second case is when all neighbors of v_k are on P. This case is a bit more tricky. The idea is this: We will show that there is a neighbor v_j of v_k such that v_{j+1} has a neighbor outside P. Then we will perform a switch operation that transforms P into the following path: $v_1 v_2 \ldots v_j v_k v_{k-1} \ldots v_{j+1} u$, as in the figure below:

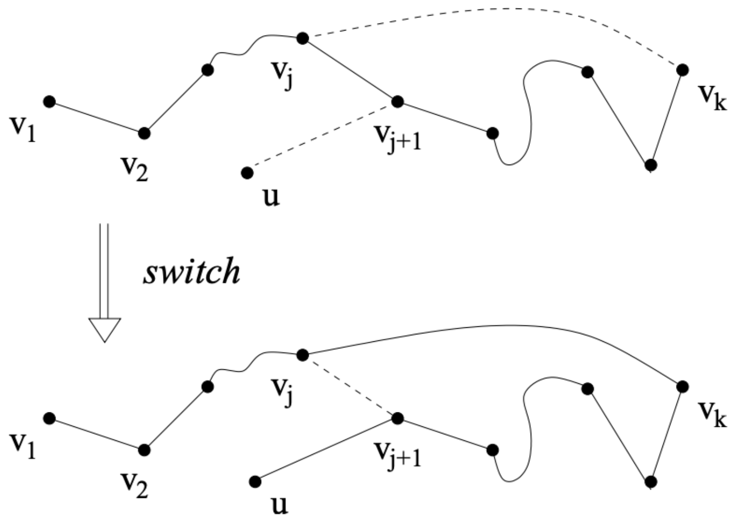


Figure 9: The switch described in the proof

Notice that the new path is indeed longer than P by one vertex. It is now sufficient to prove that such vertex v_j always exists. (Since all neighbors of v_k are on P and are different than v_k, we have $k - 1 \geq \deg(v_k) \geq n/2$, so $k \geq n/2 + 1$.)

Let's do this: for each neighbor v_j of v_k, we mark the next vertex on P, that is v_{j+1}. Since all neighbors of v_k are on P, this way we will mark $\deg(v_k)$ vertices.

Consider any vertex u not on P. If none of u 's neighbors were marked, then by adding the numbers of u 's neighbors, the marked vertices, and u itself, we would get that the total number of vertices in G is at least $\deg(u) + \deg(v_k) + 1 \geq n/2 + n/2 + 1 > n$, which is a contradiction! Therefore there must be a marked vertex that is a neighbor of u. But this means, exactly, that there will be a vertex v_j as in the figure above, and the switch operation can be applied.

Summarizing what we've done so far, the above argument shows that G has a Hamiltonian path $P = v_1 v_2 \ldots v_n$. But the theorem actually says that G has a Hamiltonian cycle, so we are not really done yet. This is left as an (optional) exercise. In other words, you need to show how (under the assumptions from the theorem) you can convert P into a Hamiltonian cycle. □

In fact a more general theorem tells us that

Theorem 8 (Ore)**.** *Let G be a graph with $n \geq 3$ vertices. We denote by deg(v) the degree of a vertex v in G. If deg(v) + deg(w) $\geq n$ for every pair of distinct non-adjacent vertices v and w of G*

then G has a Hamiltonian cycle.

Here is a graph with a Hamiltonian cycle that satisfies Ore's condition, but not Dirac's.

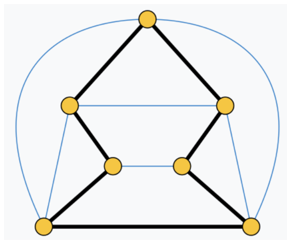


Figure 10: Graph with Hamiltonian Cycle

Note that Dirac's theorem follows immediately from Ore's theorem. We will skip the proof of Ore's theorem and leave it as a (optional) exercise.

However, neither Ore's theorem nor Dirac's theorem provide necessary conditions for the existence of a Hamiltonian cycle. For example a graph consisting of a single large cycle clearly has a Hamiltonian cycle but satisfies neither the condition in Ore's theorem nor that of Dirac's theorem.

Why do we care about Eulerian tours or Hamiltonian cycles? Apart from the more obvious applications to routing or shortest path finding in various settings (the Travelling Salesman Problem is a famous example), these concepts find applications in many varied fields such as coding theory (Gray codes), graphics (optimal polygonal mesh representations), genome sequencing, circuit design, operations research and even organic chemistry (while looking at molecular stability and transformations between molecular structures).

3 Matchings (not on the Syllabus for the Final)

One extremely common problem that needs to be solved in numerous applications is the task of pairing up agents. For example a dating app seeks to pair people up based on their preferences. Employees need to be matched with jobs, PhD students with PhD advisors and do on. There are elaborate mechanisms to pair up students with schools, or doctors with hospitals. The field of matching itself is a wide one since you can consider matching problems with all kinds of constraints.

One issue is what do you regard as a good matching? People have preferences and you want the matching to in some sense to respect these preferences. During my PhD I worked on one notion of a matching, stable matching, that tries to come up with a matching where people are in a sense incentivized to stick with their partner as decided by this matching. However there are numerous notions of matching that try to capture the notion of being aligned with the preferences of the agents you are trying to match. And here too there may be many constraints. For example, Uber faces a continuous online matching problem of allocating drivers to passengers where both the drivers and passengers have various kinds of preferences based on location, time distance, type of car etc. They probably want to minimize wait time, while still providing drivers with acceptable rides as per the driver's constraints.

Often you are trying to match people with items of some kind rather than other people. But the items are not all known at the beginning but rather appear one by one. You can imagine such an online matching problem as having a bunch of bidders who will bid for items that are arrive one by one. However you do not know what item will arrive next or exactly how much people will bid for it. A rather natural and important question is how should you allocate items to maximize some objective, say your revenue. Perhaps you shouldn't simply give items to the highest bidder since maybe you have some reason to hope that the bidder would pay even more for a later item that is yet to arrive. This sets up a complicated situation where the bidders may be incentived to lie to you in order to get their favourite items! We won't delve into the details of such situations in this class, but rather will merely define the simplest possible notion of a matching.

Definition 5 (Matchings)**.** *A matching in a graph $G = (V, E)$ is a subset of edges $M \subseteq E$ such that no two edges in E share the same incident vertex. A vertex that is incident to an edge in a matching M is matched in M; otherwise, it is unmatched. A* ***maximal matching*** *is a matching that cannot be made larger; in other words, every edge in $E \backslash M$ is incident to a vertex matched in M. A maximum matching is a matching with the maximum possible number of edges.*

Intuitively this motion of matching pairs up the vertices and no vertex can be part of multiple pairs. A perfect matching is one in which every vertex is matched. Of course not all graphs have perfect matchings and in general we are interested in finding the maximum matching in a graph.

Just because a matching is maximal does not mean that it is the largest possible matching in the graph. Here is a graph with a maximal matching, but a larger matching exists in the same graph.

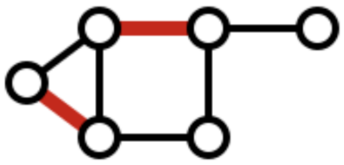


Figure 11: A maximal matching

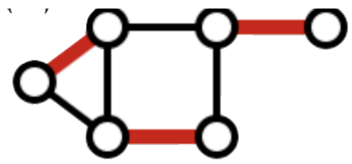
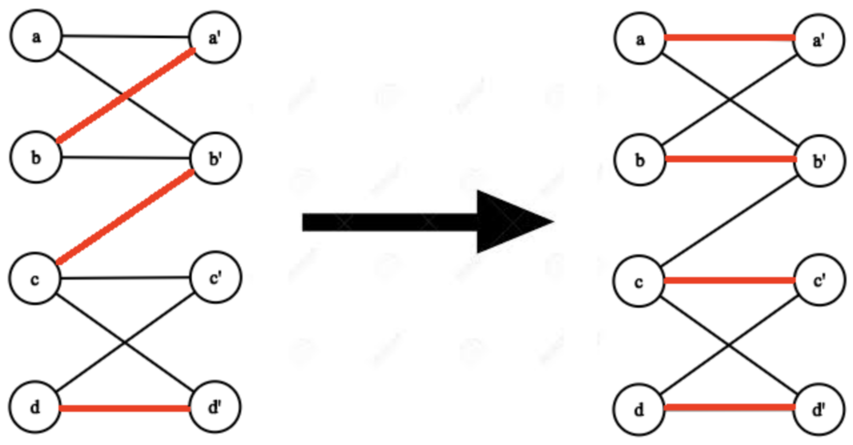


Figure 12: A maximum matching

There are efficient algorithms to calculate maximum matchings in graphs. A central idea for these algorithms is that if you can find a path in a graph that alternates between edges in the matching and edges not in the matching then you can flip these, put all edges in that path that were in the matching to be not in the new matching and all edges originally in the matching to be not in the new matching. If you have such an alternating path where the first and last edges were both not in the matching, then doing such a flip will increase the number of edges in the matching by one and thus increase the size of the matching. Such a path is called an augmenting path. We can keep doing this as long as there are augmenting paths and keep increasing the matching size.



The augmenting path we used is shown below:

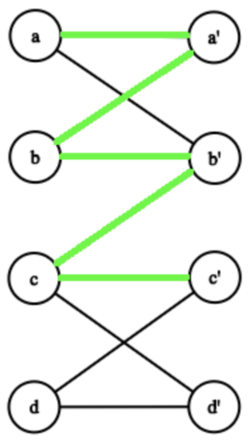


Figure 13: Flipping an augmenting path to grow a matching

Indeed if you are restricted to bipartite graphs, just finding augmenting paths and doing such flips is quite straightforward. But if you want to find the maximum matching in a general graph, we might run into some issues in trying to find the augmenting paths.

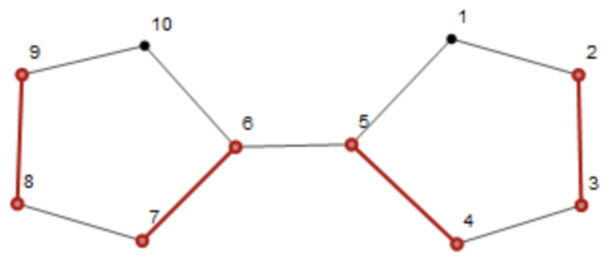


Figure 14: Graph where you need to find the correct augmenting path to pick

Here, $(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$ is an augmenting path. But we won't find it starting from vertex 1 if we happen to explore from 1 to 5 first: we'll walk $(1, 5, 4, 3, 2)$ and get stuck because we can't actually flip that path since it is an odd cycle. Similarly, we won't find it starting from vertex 10 if we happen to explore from 10 to 6 first: we'll walk $(10, 6, 7, 8, 9)$ and get stuck. Of course it's easy enough to fix in this small example, you should find the augmenting path $(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$. But it's not so obvious how to deal with this issue, in a general graph, in some kind of efficient way. This issue can in fact be dealt with efficiently even in general, which leads to the so called Blossom algorithm for maximum matching, pioneered by Jack Edmonds in 1965, but we won't get into the details of that algorithm in this class.