General matching

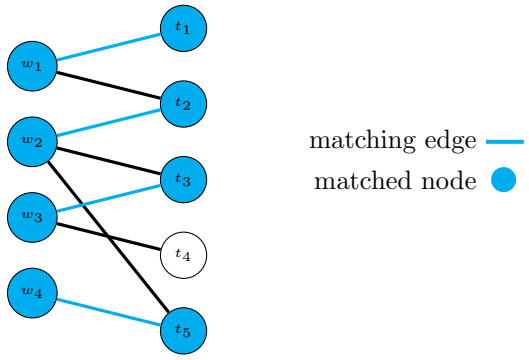
March 28, 2017

Before we introduce the general matching problem, let's first review the bipartite matching problem. Assume that you have n workers and m tasks to do. Not every worker is suited to perform all of the tasks so you are also given, for each of the n workers, the list of tasks he can do. Each task can only be assigned to a single worker. The goal is to assign a maximum number of workers to the tasks.

For instance, assume we have 4 workers and 5 tasks and that the following table describes which workers are able to perform which tasks.

	$t1$	$t2$	$t3$	$t4$	$t5$
$w1$	✓	✓			
$w2$		✓	✓		✓
$w3$			✓	✓	
$w4$					✓

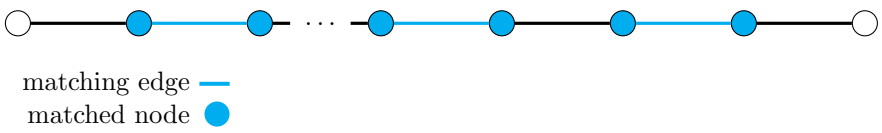
From this we can build a bipartite graph where the left-hard side nodes correspond to the workers and the right-hand side to the tasks. We put an edge between a worker and a task if that worker is able to perform that task. In this case we get the following graph.



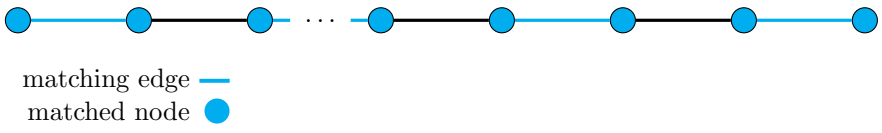
We can see that, in this case, it is possible to assign every worker to some task as show in blue in the figure (assign $w1$ to $t1$, $w2$ to $t2$, $w3$ to $t3$ and $w4$ to $t5$).

The set of blue edges is called a *matching*. More generally, a matching is a subset of edges such that no two edges share an endpoint. Because they share no endpoints, we know that each worker is assigned to at most one task.

To compute such a solution, we start from an empty matching and try to iteratively increase its size. In order to achieve this, we look for an *augmenting path* in the graph. An augmenting path is a path between two unmatched nodes that alternates between unmatched and matched edges. Before we continue, let's take a look at the structure of an augmenting path.



Any edge of the graph connecting a node from an augmenting path to a node not in it must be an unmatched edge. The reason for this is that every node except the first and the last are matched to another node belonging to the path. Since a node cannot be matched to more than one node, those nodes cannot have other matched edges out of them. By definition, the first and the last nodes are unmatched so they don't have matched edges going out of them. An augmenting path always has odd length. If it has a total of k matching edges, it will have $k + 1$ unmatched edges giving a total of $2k + 1$ edges, which is odd. Finally, if we flip each edge of an augmenting path (matched becomes unmatched and vice-versa), the result is still a valid matching.



This is simply because, as we first observed, there are no matching edges leaving an augmenting path. Hence there is no risk of having some node in the path touching two matching edges after this. Moreover, the size of the matching is increased because now we have $k + 1$ matching edges instead of k in the path.

These properties will be important later on so we summarize them:

- there are no matching edges between a node in an augmenting path and a node outside that path;
- an augmenting path always has odd length;
- inverting each edge of the path yields a matching of size +1.

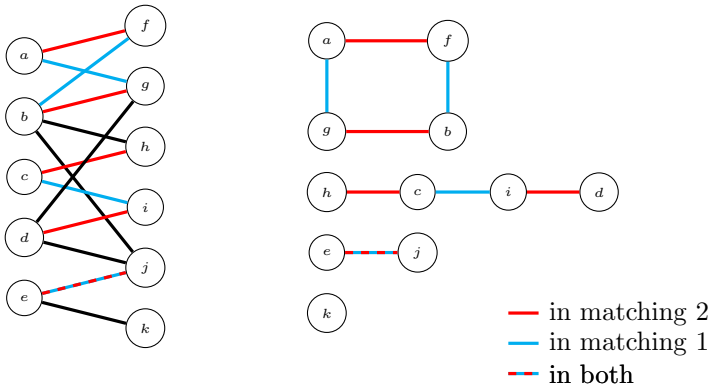
The basic algorithm to compute a maximum bipartite matching simply consists on finding an augmenting path and reversing its edges while such a path exists. Let's try to understand why is it that a this algorithm works. On one hand, it is clear that is some augmenting path exists, then by inverting the edges, we get a bigger matching and therefore the initial matching could not be maximum. Now we need to understand why having no augmenting path makes a matching optimal. One could think that maybe by finding some bad sequence of augmenting paths we would reach a local maximum instead of a global maximum. To prove that this is not the case, consider two matchings M_1 and M_2 such that M_2 is bigger than M_1. This obviously means that M_1 is not maximum. We are

going to show that using M_2 we can build an augmenting path for M_1.

Consider the subgraph formed by removing from G all edges that are in neither of the two matchings. This is a graph such that each connected component is one of the following:

1. a single node;
2. a path alternating between M_1 and M_2;
3. a cycle alternating between M_1 and M_2.

For instance, the following figure shows two matchings on a bipartite graph on the left and the subgraph obtained from removing all edges that are unmatched. As we can see, the connected components are indeed single nodes, alternating paths and alternating cycles. Note that some edges might belong to both matchings like edge between e and j.



The reason why this is true in general is that each matching forms a subgraph where each node has either degree 0 or 1. When we combine both there graphs we get a graph where each node has either degree 0, 1 or 2. Such a graph can only be a collection of isolated nodes, paths and cycles. These paths (and cycles) must alternate between M_1 and M_2 is because, being matchings, no node can be connected to two edges of the same matching.

Now, the cycles must be composed of an even number of edges. Half from the first matching and the other half from the second. If it was odd, we would have one node connected by two edges of the same matching which violates the matching definition. Therefore, since M_2 is bigger than M_1, there must be some component that is a path and that contains more edges from M_2 than M_1. That path must be an alternating path on the graph with respect to M_1 (this path is (h, c, i, d) in the example).

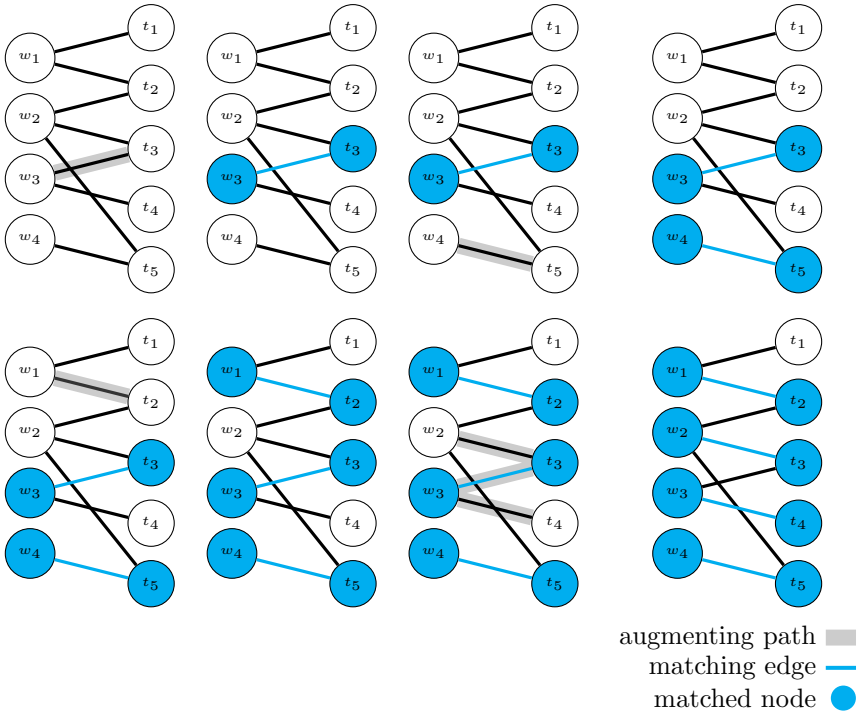
This result is known as *Berge's lemma*.

All this justifies that given a matching M on a graph G (not necessarily bipartite as we did not use the bipartiteness of G in our argument), M is maximum if and

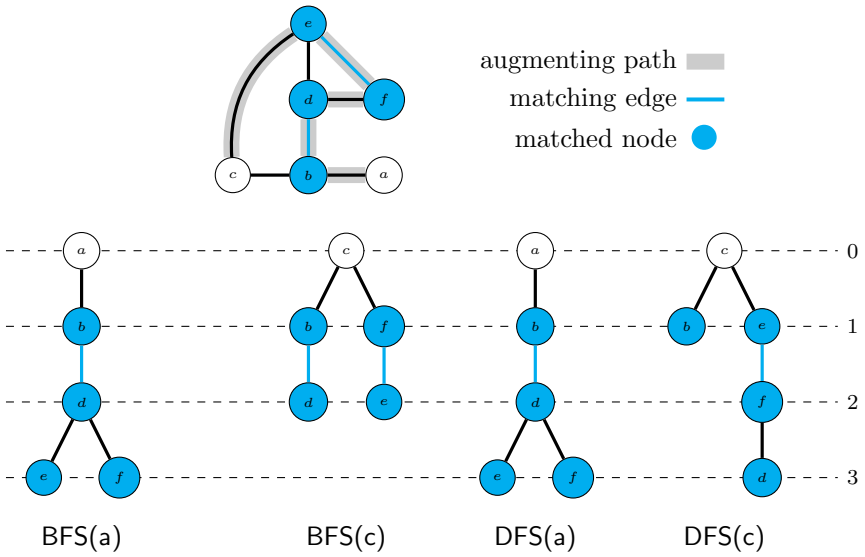
only if there is no augmenting path on G with respect to M. Thus the algorithm to compute a maximum matching in general simply consists on starting from an empty matching, successively finding augmenting paths and increasing the current matching by reversing the edges from the path.

1. $M = \emptyset$
2. while G contains an augmenting path with respect to M
   (a) find an augmenting path p with respect to M
   (b) increase M by reversing the edges of p

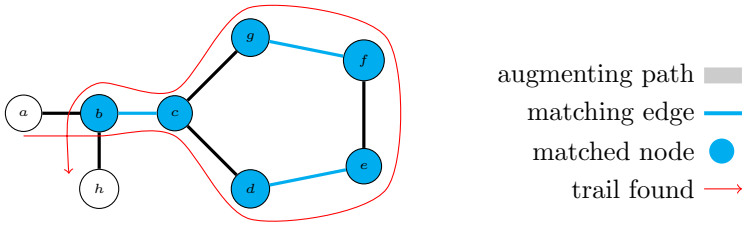
Therefore we have reduced the problem of computing a maximum matching to the problem of computing augmenting paths. The next figure shows an illustration of the execution of the algorithm on a bipartite graph.



The problem of finding an augmenting path may seem easy at first sight but it is actually quite tricky. If you try to find such a path with BFS or DFS you will fail to find to find some paths. To see this, consider the following graph. There is only one augmenting path (c, e, f, d, b, a). If you start your BFS from c, at the first iteration b will be marked as visited. Therefore when you arrive from d you will not visit it and you will fail to find the path. If you start from a, node d will mark both e and f as visited and thus f will fail to visit e. The same happens with a DFS if the nodes are expanded in alphabetical order. Note that for the DFS, there always exists an ordering of the adjacent nodes that makes it so that DFS finds the augmenting path if one exists. But there is no guarantee that the nodes will be given in that order and it is easy to make instances where the probability of finding that order is as small as we want.



The next idea that comes to mind after this is to allow to visit the nodes twice, one time coming from a matched edge and another time coming from an unmatched edge. This also does not work but for a different reason. In this case it can happen that the algorithm says that a path exists when actually no such path exists. This is because the output of the algorithm may contain cycles. The following graph shows an example.



The double label algorithm will find the trail shown in red which is not valid since it visits nodes b and c twice. The reason is that when it arrives at c from g, c was never visited while arriving via an unmatched edge. So to the algorithm proceeds. The same happens at b, which was never visited yet via an matched edge.

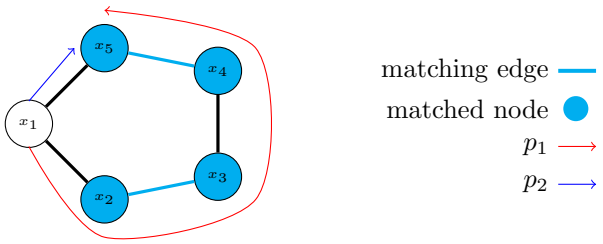
However we know that a simple BFS and DFS works for bipartite graphs. And we know that a graph is bipartite if and only if it contains no odd cycles. So what is it about odd cycles that make the algorithms fail? Actually it is not so much odd cycles but *odd alternating cycles*. An odd alternating cycle is a cycle $(x_1, x_2, x_3, \ldots, x_{2k+1})$ such that:

1. (x_1, x_2) and (x_1, x_{2k+1}) are unmatched;
2. edges in the cycle alternate between unmatched and matched.

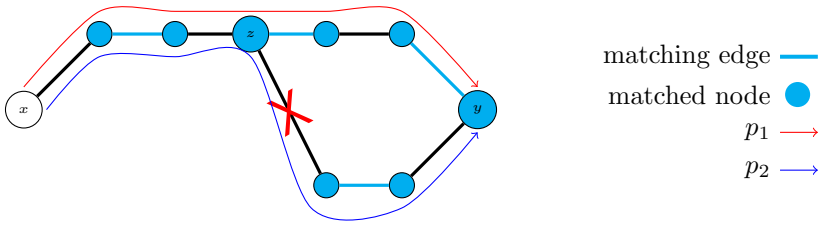
It is not hard to see that a graph contains an alternating odd cycle with respect to some matching M reachable from an unmatched node x if and only if and

only if there exists a node y such that there exist two alternating paths from x to y one of them ending with a matched edge and the other ending with an unmatched edge.

Clearly is such a cycle exists then (x_1, x_{2k+1}) and $(x_1, x_2, \ldots, x_{2k+1})$ are two such paths (with $x = x_1$ and $y = x_{k+1}$). See the next figure for $k = 2$ for clarity.

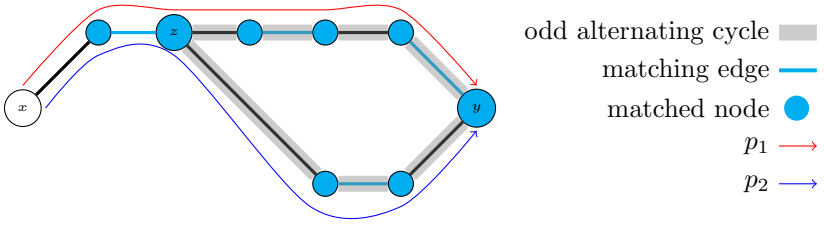


Now, suppose on the other hand that such paths exist, call them p_1, p_2. These paths must meet at some node z (maybe $z = x$). Since the paths are alternating, there cannot be a matching edge between them (we saw above that no matching edge can exit an alternating path). Hence both paths will connect via an unmatched edge. Furthermore, the edge of p_1 ending in z when walking from x must be a matching edge otherwise p_2 would not be alternating as the following picture shows (we have two unmatched edges in a row).



This means that we have situation shown in the next figure. As we can see, this gives an odd alternating cycle. The fact that the cycle has odd length comes from the fact that the sub-path of p_1 form z to y starts with an unmatched edge and ends with a matching edge. Hence it must have even length (same number of matching and unmatched edges). On the other hand the sub-path of p_2 from z to y starts with an unmatched edge and ends with an unmatched edge. Thus it must have odd length (one more unmatched edge than matching edges).

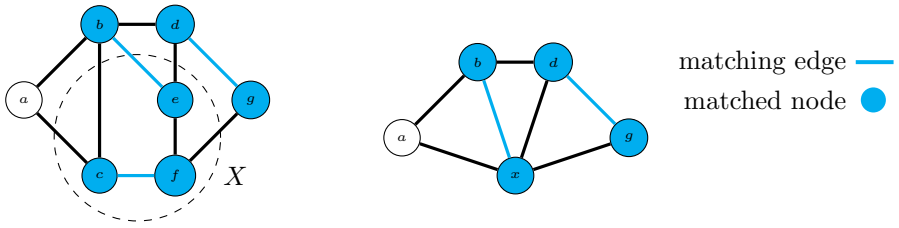
The hypothesis that x is unmatched is important because it guarantees that if $z = x$ then it still holds both paths exit z with unmatched edges. This is necessary in the definition of odd alternating cycle.



Such cycles make the previous algorithms fail because they make it so that some node of the graph (y in the above discussion) are reachable into different ways (via an unmatched and matched edge) making the order in which the nodes are explored important.

So now we know what is the problem. We just need to figure out how to overcome it. We need to find a way to get rid of such cycles. There are several possible options that we could take to achieve this. To name a few, we could, for instance, remove and edge that we are sure that no augmenting path uses it. However this seems hard to know because it kind of reduces to knowing which order the nodes should be visited. Alternatively we could contract the cycle into a single node and hope that this somehow preserves augmenting path and that given an augmenting path on the contracted graph we can build an augmenting path on the original one. This is exactly what we are going to do.

But before we proceed, let's review graph contractions. Let G be a graph and X a subset of $V(G)$, the nodes of G. The contraction of G relative to X is denoted by G/X and is a graph where all nodes of X are replaced by a single node x whose neighbors are the union of the neighbors of the nodes in X. In this document we remove self loops, thus if two nodes of X are connected, we don't add a self loop to x. We also remove multiple edges. If parallel edges would exist and at least one of them is matching, we keep it matching in the contraction. This is shown in the figure. X is connected to b via both a matching edge (e, b) and an unmatched edge (c, b). In the contraction we make the edge (x, b) matching.
The following figure shows an example of a contraction.



Theorem 1. *Let C be an alternating odd cycle in G reachable via an alternating path from some unmatched node. The graph G contains an augmenting path if and only if the graph G/C also contains one.*

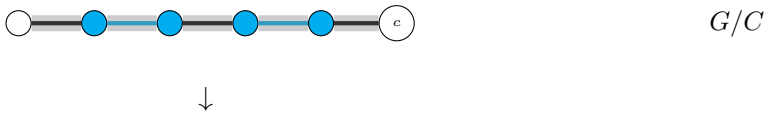
Proof. Let c be the node corresponding to C in G/C.

First note that if there exists an augmenting path in G/C that does not include node c, that path is also an augmenting path in G. Therefore we will focus on the case where the augmenting path in G/C includes node c. We will consider two cases.

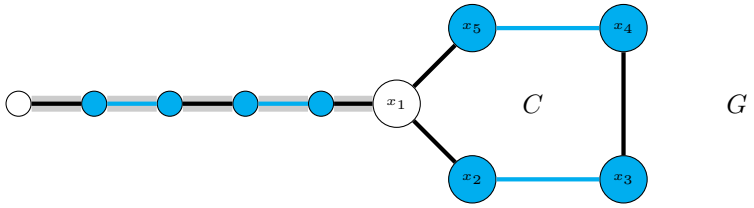
Case 1: The node x_1 from C is unmatched in G. Recall that x_1 by definition the only node from the cycle that is not matched to any other node in the cycle.

In this case, node c is unmatched in G/C. This is because x_1 is unmatched and all other nodes in C are already matched to a node within C. Hence

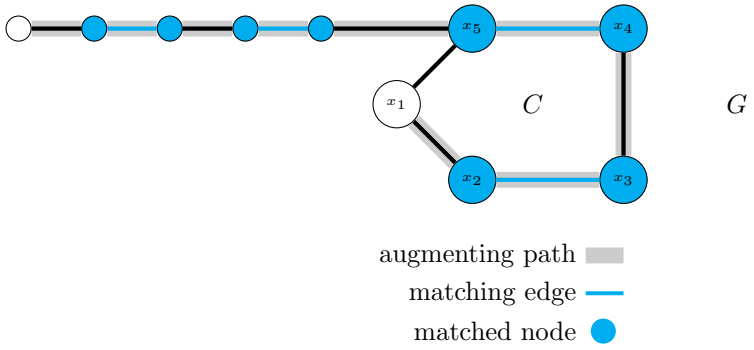
there cannot be matching edges exiting node c after the contraction. As c is unmatched, any augmenting path in G/C that includes node c, must end on it (an unmatched node can only be an endpoint of an augmenting path). When we unfold the cycle, this augmenting path will be an alternating path in G from an unmatched node in G to some node in the cycle C. If that node is x_1, since x_1 is unmatched, that path is augmenting in G. If it is any other node we can extend it by following the cycle (starting with a matched edge) until we reach x_1. In any case we get an augmenting path in G. The following figure illustrates this.



(1) the path arrives at x_1, it is already augmenting.



(2) the path arrives to some other node, we follow the cycle until x_1



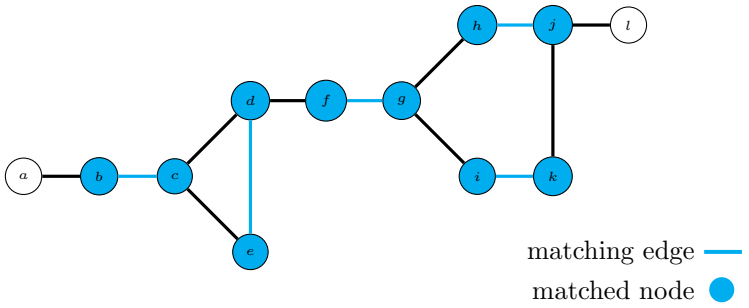
Case 2: The node x_1 from C is matched in G. By hypothesis the cycle C is reachable via an alternating path from some unmatched node, say x. Therefore, since the path is an alternating path from x to x_1, if we reverse its edges we still get a valid matching of the same size, except that now x_1 is unmatched. Then we can apply the previous case to this new matching and cycle C because x_1 has become unmatched.

This is actually enough to compute a matching of larger size. However the astute reader might notice that in this case we did not really get an augmenting path relative to the original matching (because we reversed the edge on the path from x to x_1). However it is not hard to see that by reversing those edges back

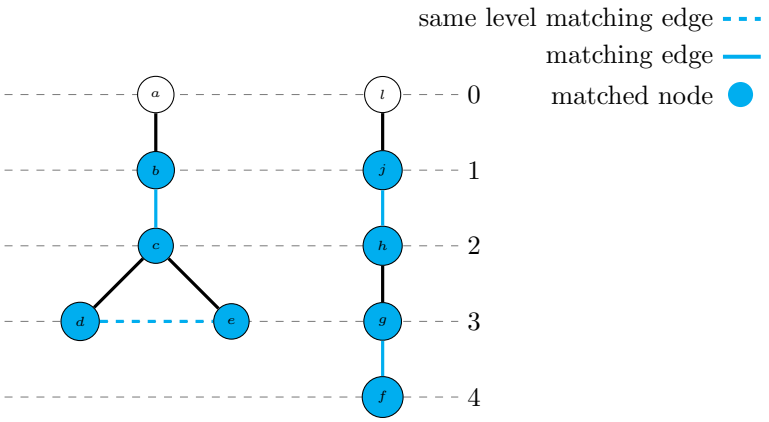
after finding an augmenting path in G/C we can actually continue to extend that path by following the cycle as before but then, instead of stopping at x_1 (which is not unmatched anymore) we continue following the path from x_1 to x and obtain a path relative to the original matching. This will be illustrated in the next example.

□

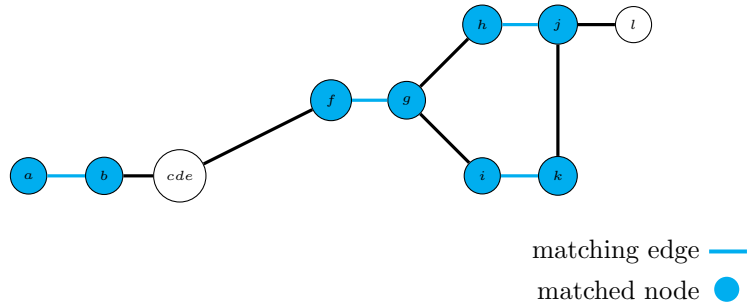
This theorem gives us an algorithm to compute augmenting paths. We start on G and compute an alternating odd cycle in G. This can be achieved using a BFS from the set of unmatched nodes and finding an edge $e = (u, v)$ between two nodes at the same level. Tracing back the parents of u and v will lead to one of two situations: either u and v are two distinct sources, in which case we have found an augmenting path; or at some point we reach a common ancestor. This ancestor together with the paths linking it to u and v forms an alternating odd cycle. Since the graph becomes smaller at each contraction, this process must stop at some point.
We will illustrate the execution of the algorithm on the following graph.



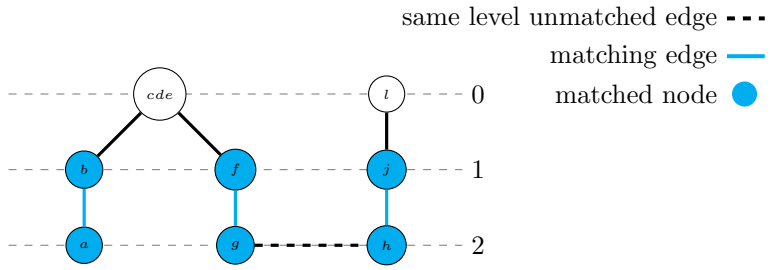
Perform a BFS from the set of unmatched nodes a and l and look for an edge between two nodes in the same level. In this case we find edge (d, e) (it does not need to be a matching edge). Such an edge yields either an alternating odd cycle or an augmenting path depending on whether they trace back to a common ancestor or two distinct sources from the BFS. This can be computed by going up the tree following the parents of d and e. We either reach a common ancestor or two sources. In this case we the common ancestor c and get the cycle $C = \{c, d, e\}$.



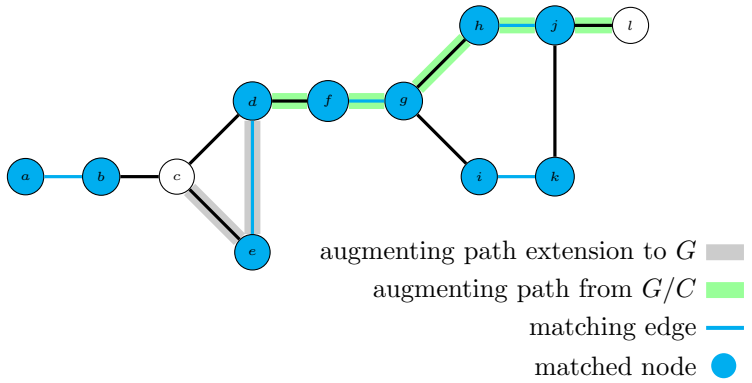
Since node c is matched, we reverse the matching on the path from a to c. Then we know that the result graph has an augmenting path if and only if G/C has an augmenting path.



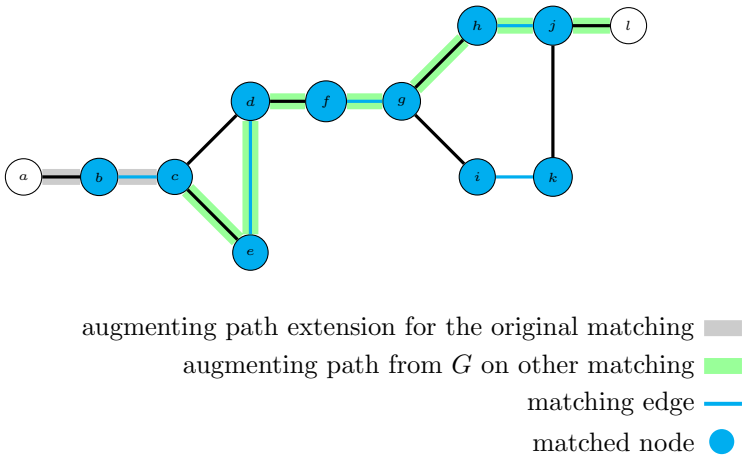
We perform a new BFS on G/C from the unmatched nodes cde and l. Here we find edge (g, h) between two nodes of the same level. By tracing back the parents we reach nodes cde and l. This means that we have found and augmenting path (cde, f, g, h, j, l) on G/C.



Unfolding the path we get a path from d to l on G as shown in the figure. Then we follow the cycle starting with a matching edge until we reach c, the unmatched node of the cycle. This yields an augmenting path in G.



As we observed at the end of the previous proof, this path is not an augmenting path for the initial matching. It is an augmenting path for the matching where b is matched to a instead of c. But if we want we can actually recover an augmenting path relative to the original matching by reversing again the edges on the path (a, b, c) and extending the previous path until a as shown next.



I will describe and implementation of the algorithm when I find the time. For now it is a good exercise for you to try to implement it yourself. Note that there can we several contractions during the execution. I will also try to add examples where this happens to make it more clear what happens in this case. Contractions are not always easy to implement because unfolding might be tricky.