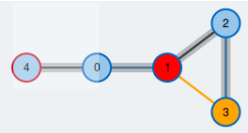


Edmonds's Blossom Algorithm

T

Introduction | Create a graph | Run the algorithm | Description of the algorithm | More

Matchings in graphs

We consider an undirected graph $G = (V, E)$ with a set of vertices V and a set of edges E. A **matching** $M \subseteq E$ is a subset of edges such that every vertex is covered by at most one edge of M. In Figure 1, we illustrate a valid matching in a graph. Its edges are colored blue. Each node is incident to at most one edge. If we added e.g. the edge between 0 and 3 to the current matching, the result would not be a matching anymore because 0 would then be contained twice in the matching.

We call a vertex that is not covered by the matching a **free vertex**. Given an edge $e = \{v, w\} \in M$, v is the **mate** of w and vice versa. In our example, 3 and 5 are free vertices because they are not incident to any of the blue edges, 0 and 1 are mates of each other and 2 and 4 as well.

A matching M is called a **maximum matching** if its number of edges is maximum, i.e. if there does not exist any other matching M' in G with |M'|>|M|. The matching in Figure 1 is not maximum. We could evidently add the edge between the free vertices 3 and 5 to our matching and, by doing so, increase its number of edges to 3.

A graph is called **bipartite** if its vertices can be partitioned into two sets V_1 and V_2 such that every edge has one endpoint in V_1 and the other one in V_2. We know that a graph is bipartite if and only if it does not contain any odd cycles. Obviously, the graph in our example is not bipartite because it contains triangles.

Edmonds's Blossom Algorithm computes a maximum matching in a general graph. In contrast to many other algorithms, the graph is not required to be bipartite. It extends the idea of the Hopcroft-Karp Algorithm, which computes a maximum matching for a bipartite graph, by treating odd cycles appropriately.

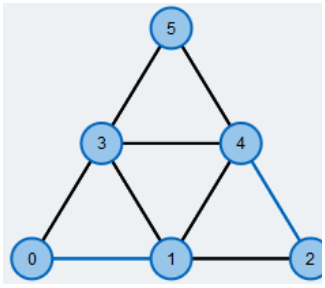


Figure 1.
Matching in a non-bipartite graph which is not maxi
and 5 are free vertices.

Algorithm for bipartite graphs

We shortly recapitulate the basic concepts of the Hopcroft-Karp Algorithm that are also relevant for Edmonds's Blossom Algorithm. In order to improve a given matching, we try to find an augmenting path. An **augmenting path** is a path starting with a free node, ending with a free node and alternating between unmatched and matched edges. You can see an example of such an augmenting path (colored in green) in Figure 2. The path starts at the free vertex 3 and ends at the free vertex 5 and alternates between unmatched and matched edges. An other (trivial) example of an augmenting path would be the direct edge between 3 and 5 that we already considered in the first paragraph.

If we have found an augmenting path, we can improve the current matching by inverting the edges along the path: matched edges are changed to unmatched ones and vice versa. By doing so, we increase the cardinality of the matching by 1. The result of this procedure, a matching with three instead of two edges, can bee seen in Figure 3.

The following theorem by C. Berge is essential for the algorithm: A matching M is not maximum iff there exists an augmenting path in G. Thus, if we cannot find an augmenting path in G anymore, the resulting matching must be maximum.

But how can we find augmenting paths in a graph? We first pick an arbitrary free node r and start a modified Breadth-First Search (BFS) from there. While traversing the graph we construct a layered tree with root r. Edges from even to odd layers are unmatched edges, edges from odd to even layers are matched ones. Such a layered BFS tree is shown in Figure 4.

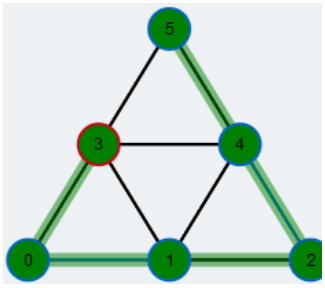


Figure 2.
Augmenting path from free vertex 3 to free verte

If we check a new node v during the execution of the BFS, one of the following cases may occur:

- **Case 1: v is a free node**
  Then we have found an augmenting path from r to v. We invert the path and are done with the BFS.
- **Case 2: v is already matched and not contained in the tree so far**
  Then we add v and its mate to the tree and push the mate of v to the BFS queue to continue the search from there later.
- **Case 3: v is already contained in the tree and we have detected a cycle of even length**
  There is nothing to do in this case, we ignore v and continue the BFS.
- **Case 4: v is already contained in the tree and we have detected a cycle of odd length**
  Since our graph is bipartite, this case cannot occur. When developing an algorithm for general graphs, however, we have to deal with this case as well.

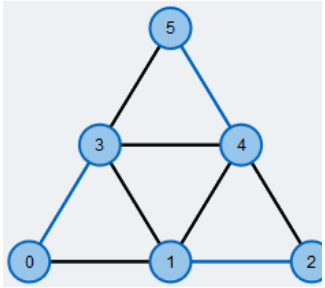
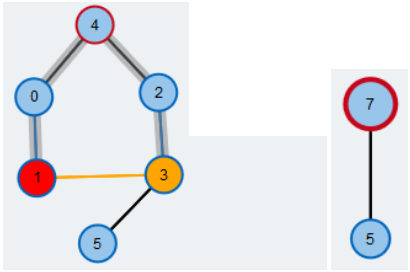


Figure 3.
Matching after inverting the augmenting path. Since
no free vertices anymore, the matching must be ma

General graphs: How to deal with odd-length cycles?

In general graphs, we also use the procedure explained above to find augmenting paths. However, we might detect odd-length cycles now (case 4) and have to treat this case appropriately. The idea is the following: We ignore the entire cycle by contracting it to a single supernode and continue the BFS in the new graph. When we have found an augmenting path at some point we have to expand all supernodes again (in reversed order) before inverting the matching. We call such odd-length cycles **blossoms** and this is why the algorithm is often referred to as the Blossom Algorithm.

We illustrate the procedure of contracting an odd-length cycle to a supernode and expanding it again with the following small example. In the picture to the left, we see a layered BFS tree with root 4. At the moment, the active node is 1 and we check the edge between 1 and 3. Since 3 is already contained in the tree, we know that there must be a cycle. The cycle has odd length, so we end up in case 4 and have to contract the whole blossom (1, 0, 4, 2, 3) to a new supernode 7 (on the right).



From 7 we continue the Breadth-First Search and find an augmenting path between 7 and 5. When we have found an augmenting path, we have to check if there are any contracted nodes that must be expanded again. Thus, we expand the supernode 7 to the original blossom (1, 0, 4, 2, 3) again (picture to the right). In addition, we have to reconstruct the augmenting path through the blossom correctly. Now that there are not any contracted nodes left we can invert the augmenting path.

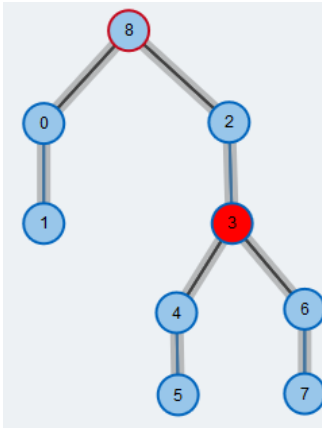
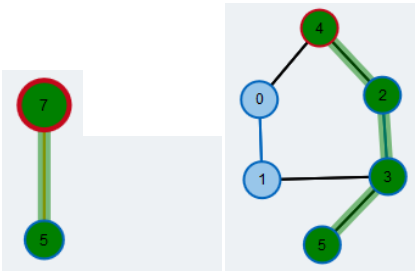
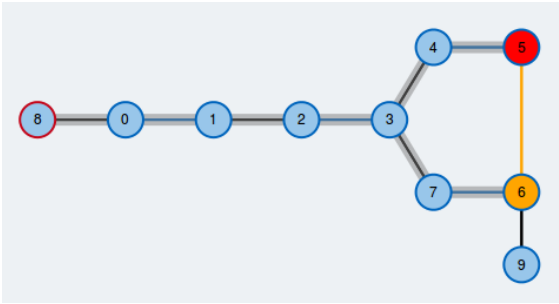


Figure 4.
Layered BFS tree (in grey) with root 8 and currentl
node 3. Layer 0: 8, Layer 1: 0 and 2, Layer 2: 1 ar

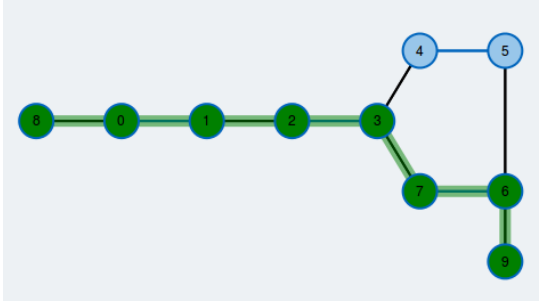


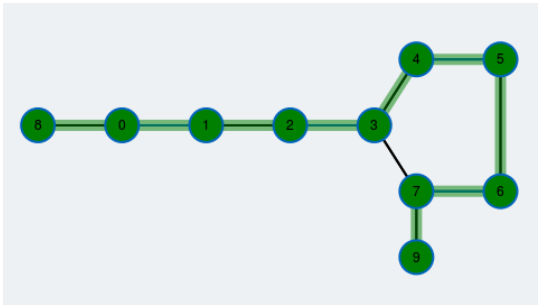
Why does the shrinking method work?

We illustrate the idea behind the contraction of blossoms by having a look at the following example graph. The picture shows a snapshot of the algorithm: We have already constructed a layered tree with root 8 and now we are detecting the blossom (3,4,5,6,7) when exploring the edge between 5 and 6. We call the path from the root to the blossom (in this case 8 to 3) the **stem** and the intersecting node 3 the **tip** of the stem.



First note that the last edge of the stem must always be a matched edge (edge from 2 to 3 in the example). Then, for each node v contained in the blossom (except for the tip itself), there are two distinct paths from the tip to v. To reach node 6 from the tip, for instance, we could either traverse the blossom clockwise and choose the path (3,4,5,6) or we traverse it counter-clockwise and choose the path (3,7,6). One of the paths always ends with a matched edge, the other one with an unmatched edge. Thus, one of the two paths will make it possible to complete an augmenting path correctly later. At this point of the algorithm, however, we cannot make this decision because we do not know yet where the augmenting path (if there is any) will lead to. Thus, we postpone the decision of correctly traversing the blossom by shrinking the entire blossom to a supernode. When we find a free node, we expand the supernodes again and can reconstruct the augmenting path correctly.
Have a look at the two examples above. In both graphs, we find the free node 9 and have to reconstruct the path through the blossom. The edge from 9 to the blossom is unmatched in both cases so we have to start the traversal of the blossom with a matched edge (between 6 and 7). Thus, in the first graph, we go clockwise until we reach the tip 3 and in the second graph we go counter-clockwise.





What now?

Create a graph and play through the algorithm

Try algorithm after creating a graph

Try algorithm on an example graph