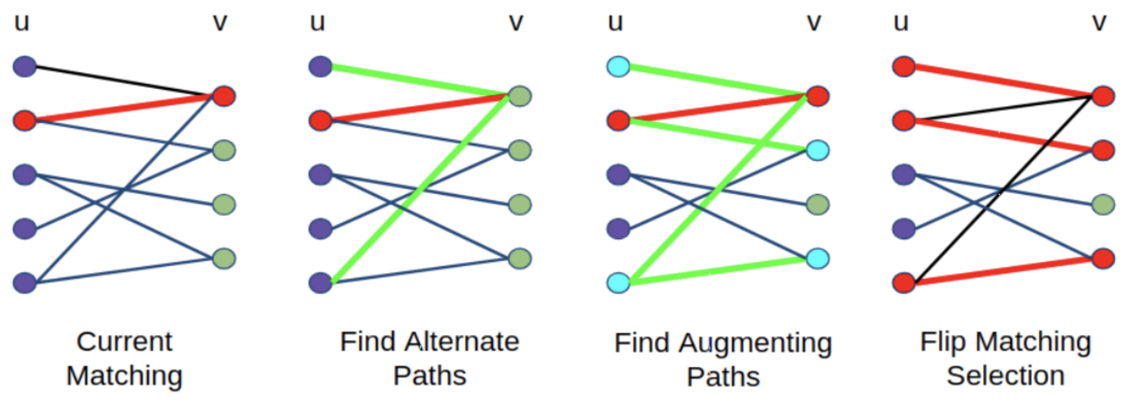
Hopcroft-Karp Algorithm

We will first define a few terms before moving to the actual algorithm that will be needed later on.

- A **matching** M is a set of pairwise non-adjacent edges of a graph (in other words, no more than one edge from the set should be incident to any vertex of the graph M). The **cardinality** of a matching is the number of edges in it. The maximum (or largest) matching is a matching whose cardinality is maximum among all possible matchings in a given graph. All those vertices that have an adjacent edge from the matching (i.e., which have a degree exactly one in the subgraph formed by M) are called **saturated** by this matching.
- A **path** of length k here means a *simple* path (i.e. not containing repeated vertices or edges) containing k edges, unless specified otherwise.
- An **alternating path** (in a bipartite graph, with respect to some matching) is a path in which the edges alternately belong/do not belong to the matching.
- An **augmenting path** (in a bipartite graph, with respect to some matching) is an alternating path whose initial and final vertices are unsaturated, i.e., they do not belong in the matching.

The Hopcroft Karp algorithm is based on the below concept.
A matching M is not maximum if there exists an augmenting path. It is also true another way, i.e, a matching is maximum if no augmenting path exists.
So the idea is to one by one look for augmenting paths. And add the found paths to current matching.



- In the above diagram the current matching is shown in red colour.
- The green colored path in the second figure is an alternate path discovered by the bfs function involving the current set of matched vertices.

- The third figure shows us the various augmenting paths found by the dfs function where the initial and the final vertices are unsaturated.
- Finally, we flip the matching found by the augmenting paths in the network.

Steps

1) Initialize Maximal Matching M as empty.
2) While there exists an Augmenting Path p
   Remove matching edges of p from M and add not-matching edges of p to M
   (This increases the size of M by 1 as p starts and ends with a free vertex)
3) Return M.

Here is the implementation of the above idea.

```
/*
        function to construct the level graph
        using breadth-first search.
*/

function bfs()

        //   dist array to store the distances in level graph
        dist = array(n1, -1)
        q = queue()
        for u from 0 to n1
                if (!used[u])
                        //   if u is unmatched then push u to queue
                        q.push(u)
                        dist[u] = 0


        //   construct level graph using bfs algorithm
        while q is not empty
                u = q.front()
                for v in adj[v]
                        u2 = matching[u]
                        if (u2 >= 0 && dist[u2] < 0)
                                dist[u2] = dist[u] + 1
                                q.push(u2)
/*
        function to find the augmenting path
        starting at vertex u. Return true if found
        else return false
```

```
*/

function dfs(u)

        //   mark u as visited
        vis[u] = true
        for v in adj[u]
                u2 = matching[v]
                //   if v is currently unmatched or there exists augmenting path
                //   from u2 then match v to u
                if (u2 < 0 || (!vis[u2] && dist[u2] == dist[u] + 1 && dfs(u2)))
                        matching[v] = u
                        used[u] = true
                        return true

        //   return false if no augmenting path from u is found
        return false

/*
        function to return the size of maximum matching in the given graph G
        with number of nodes n1 and n2 in 2 partitions
*/

function HopcroftKarp(G, n1, n2)

        /*
                matching array stores the current assignment
                of vertices in the second partition
        */
        matching = array(n2, -1)

        //   variable to store the current size of matching
        res = 0
        while true
                //   call dfs to construct new level graph
                bfs()
                //   variable to store matching in current phase
                f = 0

                //   try to find augmenting path using dfs
                for (u = 0 u < n1 ++u)
                        if (!used[u] && dfs(u))
                ++f

        //   if no augmenting path is found return res
        if (!f)
            return res

        //   add to res the size of matching in current phase
        res += f
```

Time Complexity: The time complexity of the above algorithm is $O(E\sqrt{V})$ where E is the total number of edges in the graph and V is the number of vertices. Each iteration of the Hopcroft-Karp algorithm executes the breadth-first search and depth-first search once. This takes $O(E)$ time. Since each iteration of the algorithm finds the maximal set of shortest augmenting paths by eliminating a path from a root to an unmatched edge, there are at most $O(\sqrt{V})$ iterations needed in the worst case.

Space Complexity: Hopcroft-Karp algorithm takes O(V+E) space in the worst case which is the same as that of a bfs or dfs algorithm.