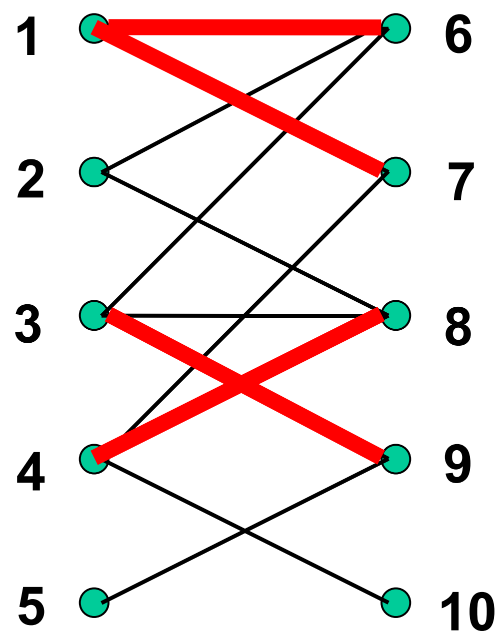
Part 3
Graph and matching algorithms

- **Augmenting path algorithm** for finding a **maximum cardinality matching** in a bipartite graph
- **Ford-Fulkerson algorithm** for finding a **maximum flow** in a network
- **Gale / Shapley algorithm** for finding a stable matching in an instance of the **stable marriage problem**
- **Applications of matching problems**
- **Floyd-Warshall** algorithm for computing **all-pairs shortest paths** in a graph

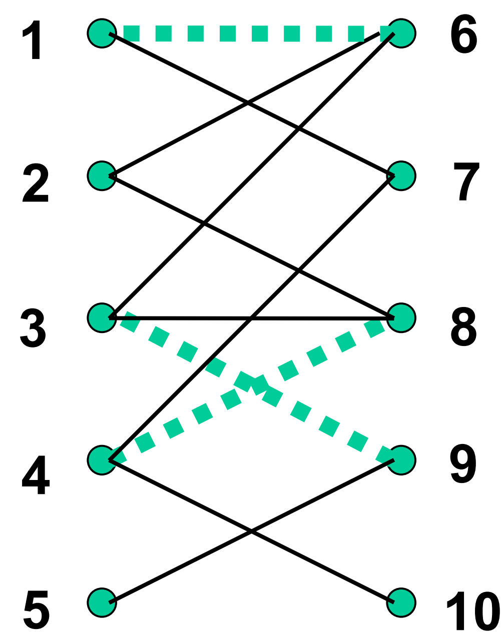
Matching in bipartite graphs

- A *bipartite* graph is a graph G=(V,E), where V can be partitioned into a "left hand side" U and a "right hand side" W so that every edge in E joins a vertex in U to a vertex in W

- A *matching* in G is a subset M of E such that no two edges in M have a vertex in common



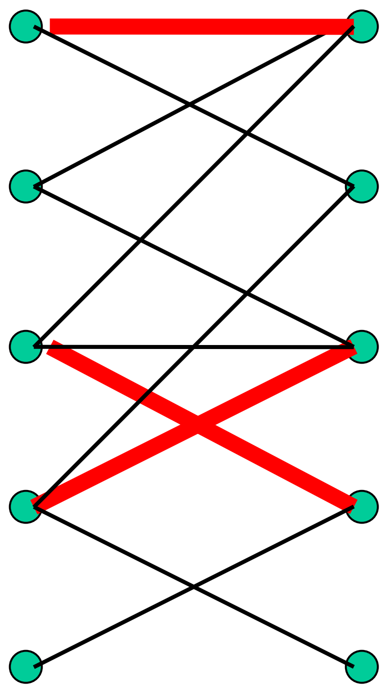
Not a matching

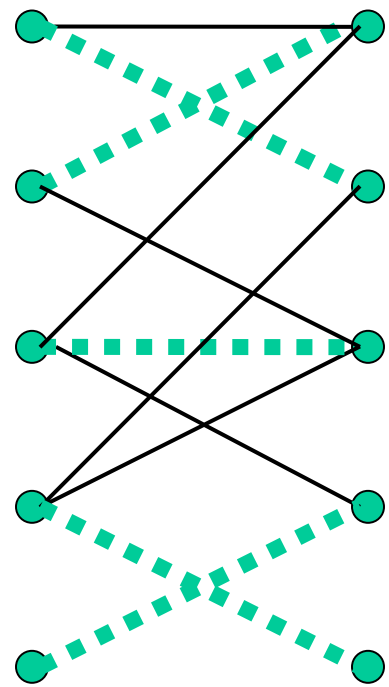


A matching

- A *maximum (cardinality) matching* in G is a matching that contains the largest number of edges
  - it is *perfect* if $|M| = |V|/2$, i.e., if every vertex is incident to an edge in M

$|M|=3$





$|M|=5=|V|/2$

Maximum matching problem

Input: A bipartite graph G

Output: A maximum matching M in G

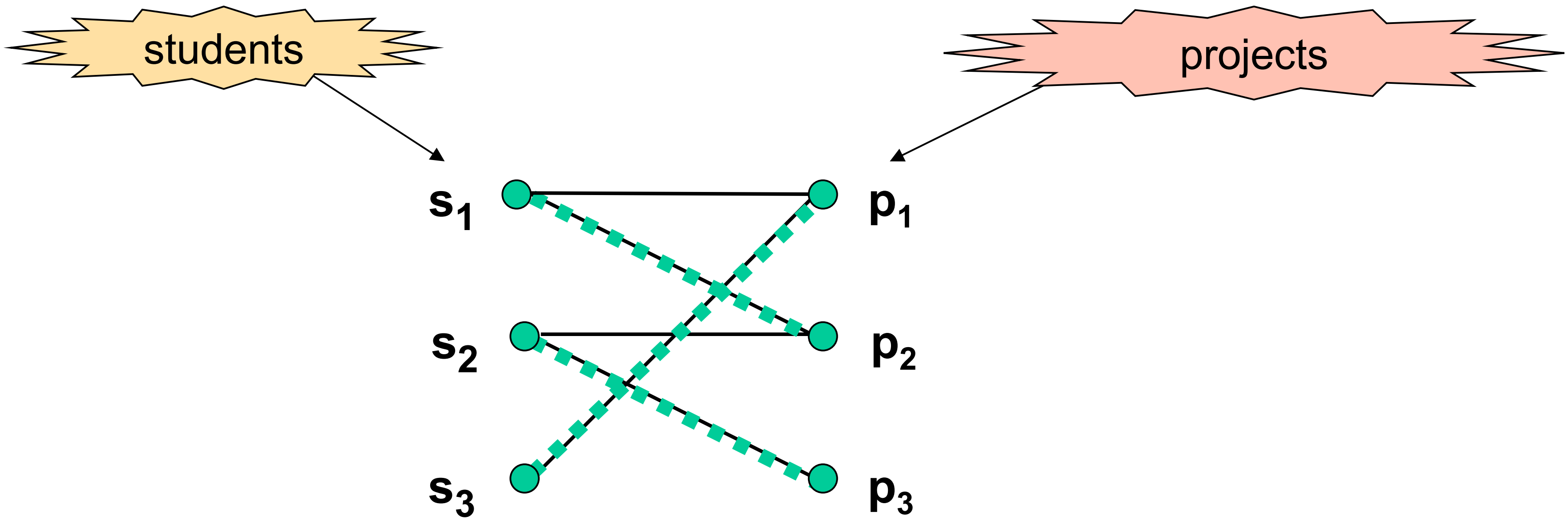
Application – student-project allocation

- Suppose there are 3 students s_1, s_2, s_3 and 3 projects p_1, p_2, p_3
- Students' preferences:

  s_1: p_1 p_2
  s_2: p_2 p_3
  s_3: p_1

- First-come first-served algorithm might consider students in the order s_1, s_2, s_3
- Matching obtained is of size 2
- Consider instead the order s_3, s_1, s_2
- Matching obtained is of size 3

Graph-theoretic formulation

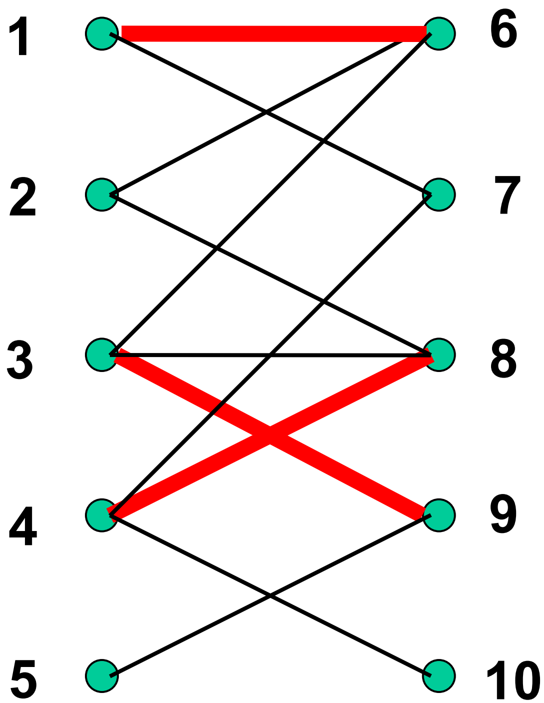


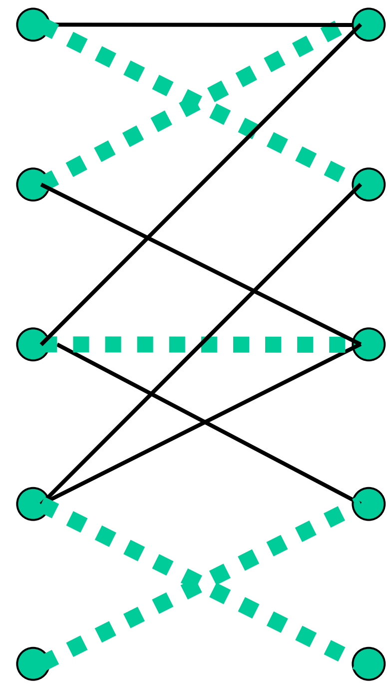
- **Seek maximum cardinality matching of students to projects in the constructed bipartite graph**

Naïve algorithm for maximum matching

- Suppose there are n students and n projects
- Try out all possible assignments of students to projects
  - allowing for a student to have no project
- Check whether the assignment is a matching
- Output largest size of matching found
- More than $n!$ assignments to try
- But, for example, $70! > 10^{100}$
- And $n>>70$ in many applications!
- Faster algorithm: $O(n^3)$

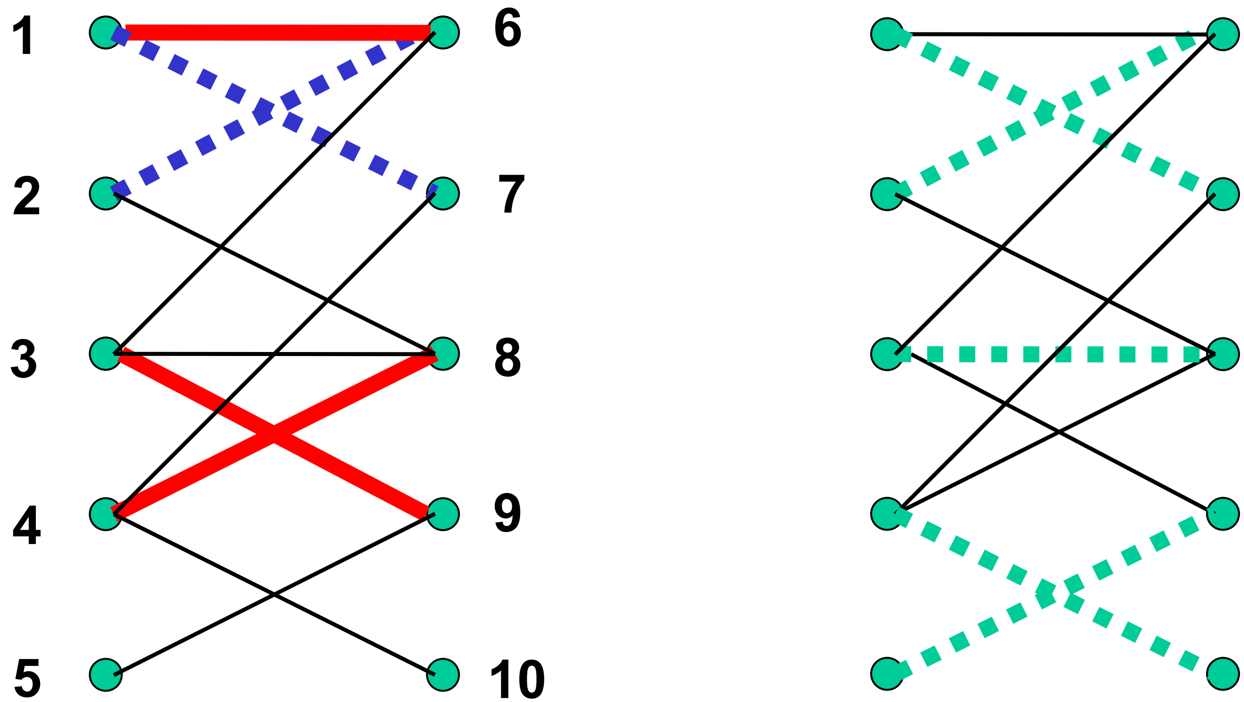
Towards a faster algorithm





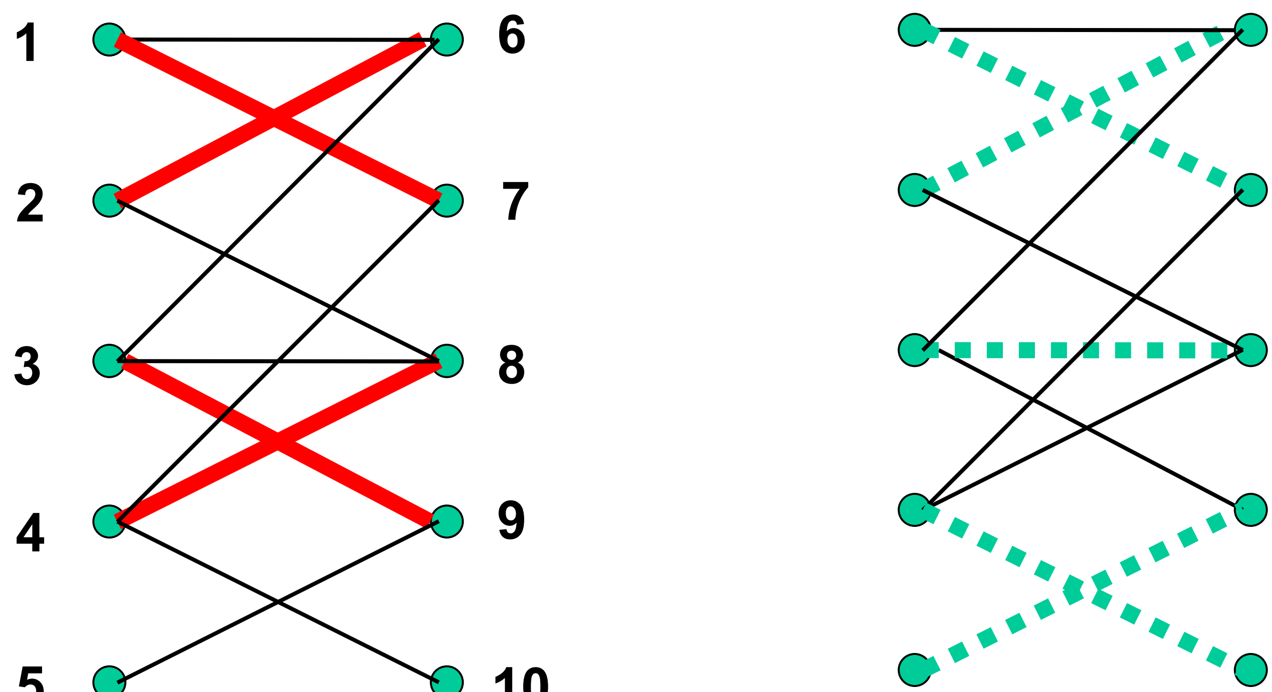
LH matching is of size 3 **RH matching is of size 5**

Towards a faster algorithm



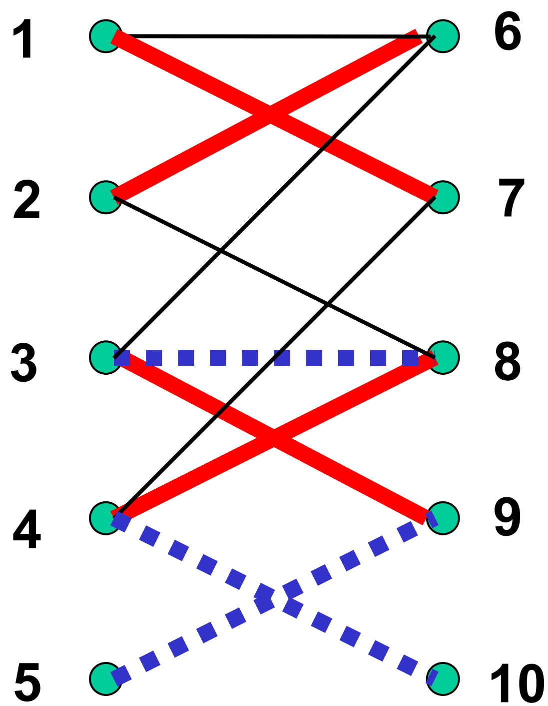
Remove edge {1,6} from LH matching and replace it with edges {1,7} and {2,6}

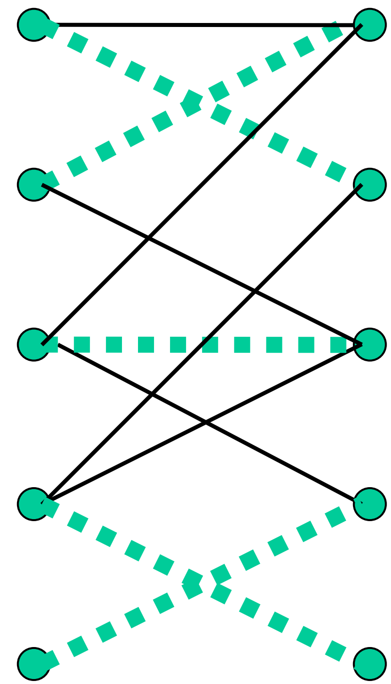
Towards a faster algorithm



Replacement has been made

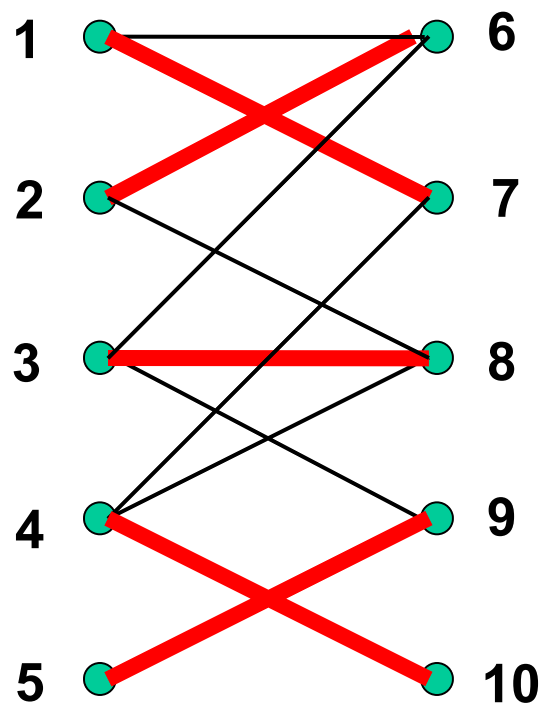
Towards a faster algorithm

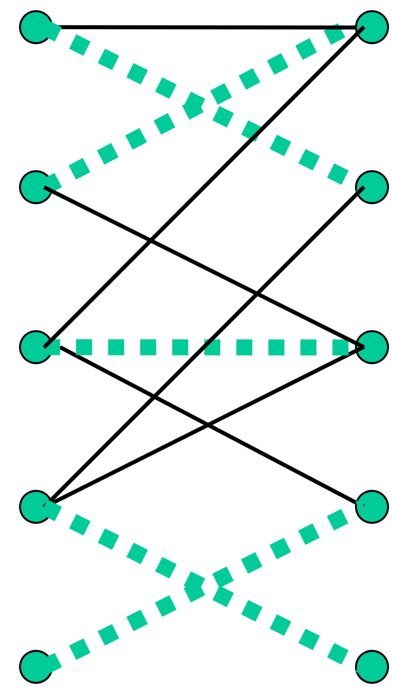




Remove edges {3,9} and {4,8} from LH matching and replace them with edges {3,8}, {4,10} and {5,9}

Towards a faster algorithm





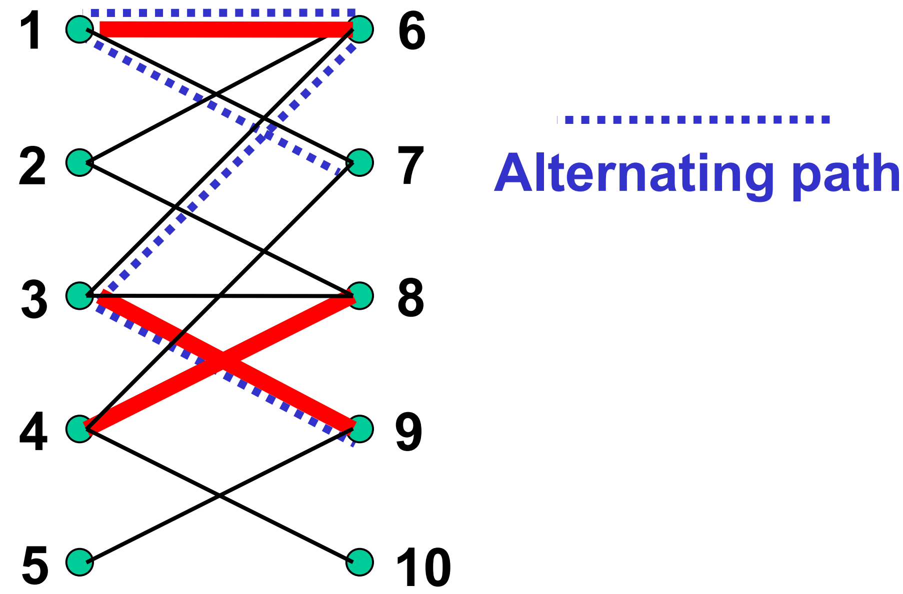
Replacement has been made

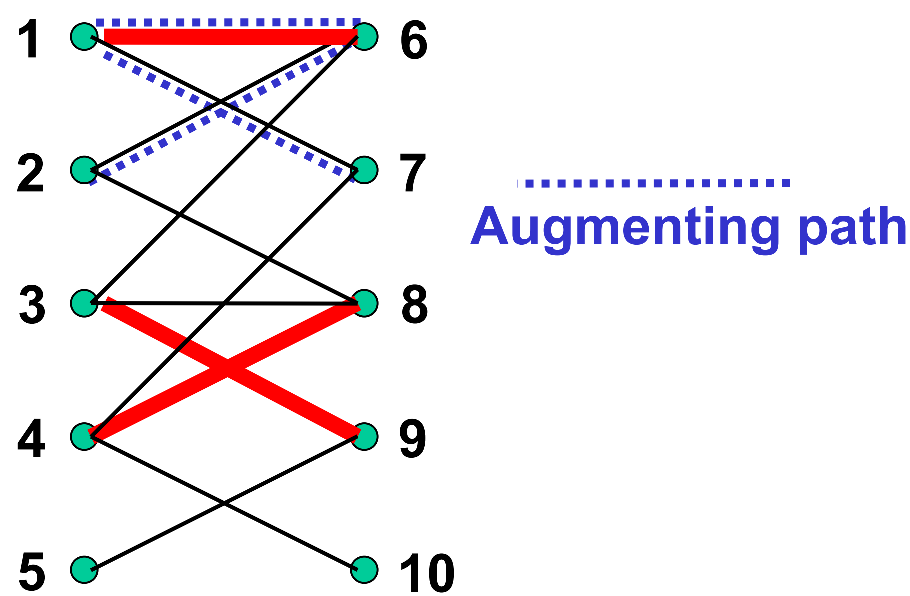
LH matching now equals RH matching

Augmenting paths in graphs

Given a matching **M** in a bipartite graph **G**:

- a vertex **u** is ***matched*** if **{u,v}∈M** for some vertex **v** - in this case **u** and **v** are ***mates***
- a vertex **u** is ***exposed*** if it is not matched
- an ***alternating path*** comprises edges in **M** and edges not in **M** alternately
- an ***augmenting path*** for **M** is an alternating path which starts and ends at exposed vertices

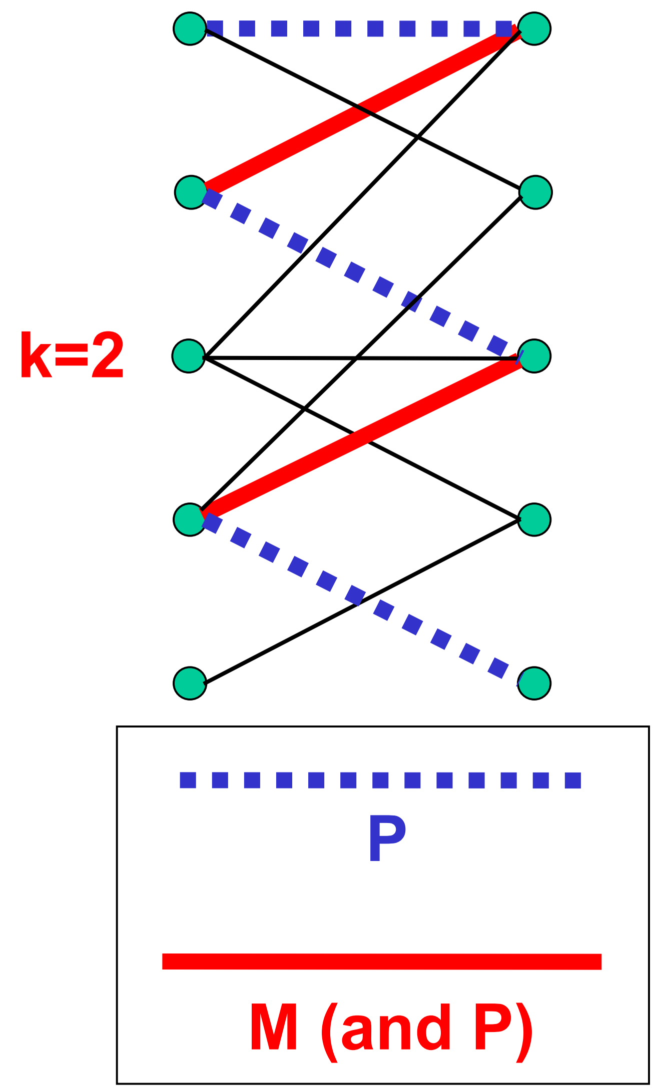




Why are augmenting paths important?

Suppose we have a matching M in a graph G, where M admits an augmenting path P

- **The augmenting path must have 2k+1 edges for some k**
- **We can form a matching L of size |M|+1 by "augmenting along P" as follows:**
  - **Initially let L=M**
  - **Remove from L the k edges on the augmenting path P belonging to M**
  - **Add to L the k+1 edges on the augmenting path P not belonging to M**



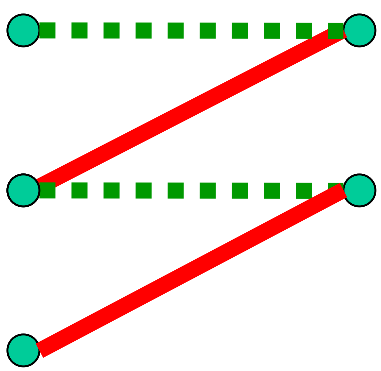
Augmenting Path Theorem

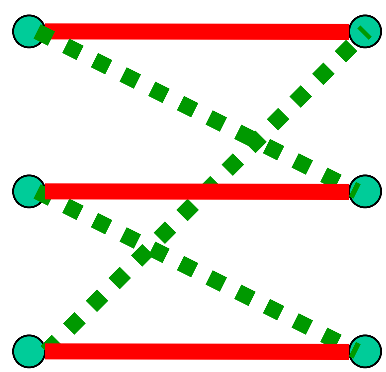
Theorem: **M is of maximum cardinality if and only if M admits no augmenting path**

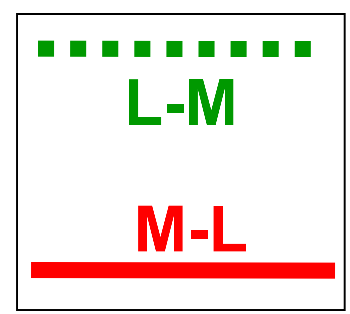
Proof: **If M admits an augmenting path, then M cannot be of maximum cardinality (see previous slide).**

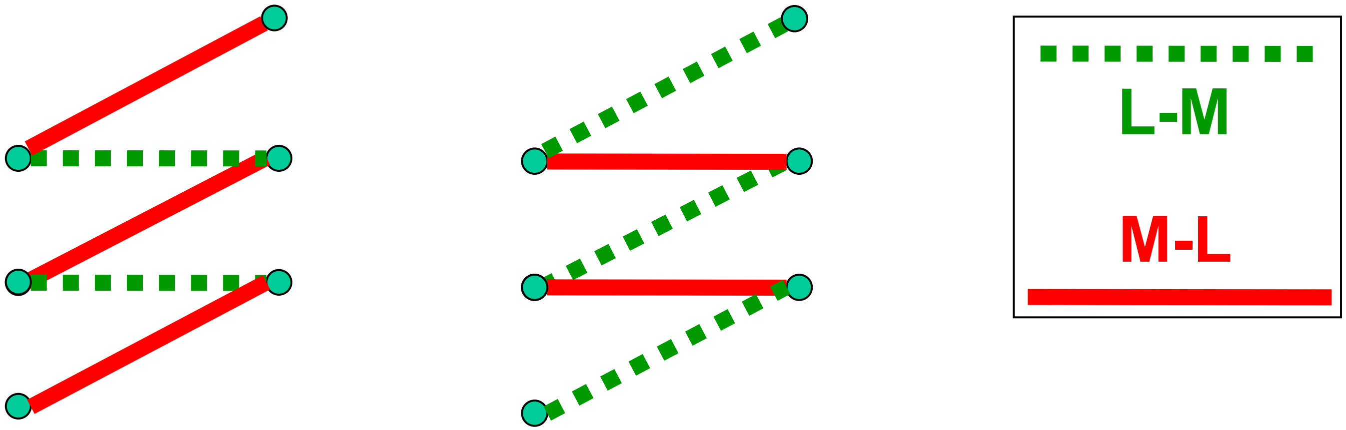
Conversely suppose that M admits no augmenting path. Let L be a maximum cardinality matching. We prove $|L|=|M|$.

Let $X=L\oplus M=(L-M)\cup(M-L)$. Colour edges of X: e is green if $e\in L-M$, e is red if $e\in M-L$. Connected components of X are paths and cycles of alternating colours.



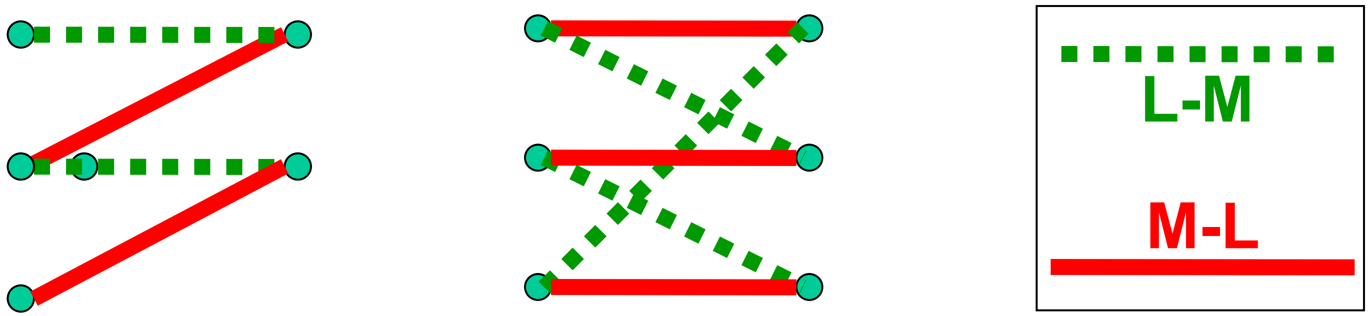






Suppose there is an alternating path of odd length.

- If both end edges red, L admits an augmenting path
- If both end edges green, M admits an augmenting path
- So the connected components of X can only be paths and cycles of even length.
- For each component, number of green edges = number of red edges, so |L|=|M|.



The augmenting path algorithm

```
/** Input: bipartite graph g
  * Output: maximum matching m in g */
m = ∅;
while (true)
{   search for augmenting path p;
    if (found)
       augment m along p;
    else
      break; // m is a maximum matching
}
```

Searching for an augmenting path

- search for an augmenting path uses a special type of breadth-first search
- search ‘fans out’ only from vertices on one ‘side’ of the graph
- idea: traverse
  - left-right using non-matching edges
  - right-left using matching edges

Searching for an augmenting path

```
public class Vertex {
  public boolean visited, startVertex; // false initially
  public Vertex predecessor, mate;
  // graph represented by adjacency lists
  public List<Vertex> adjacentV;
}
```

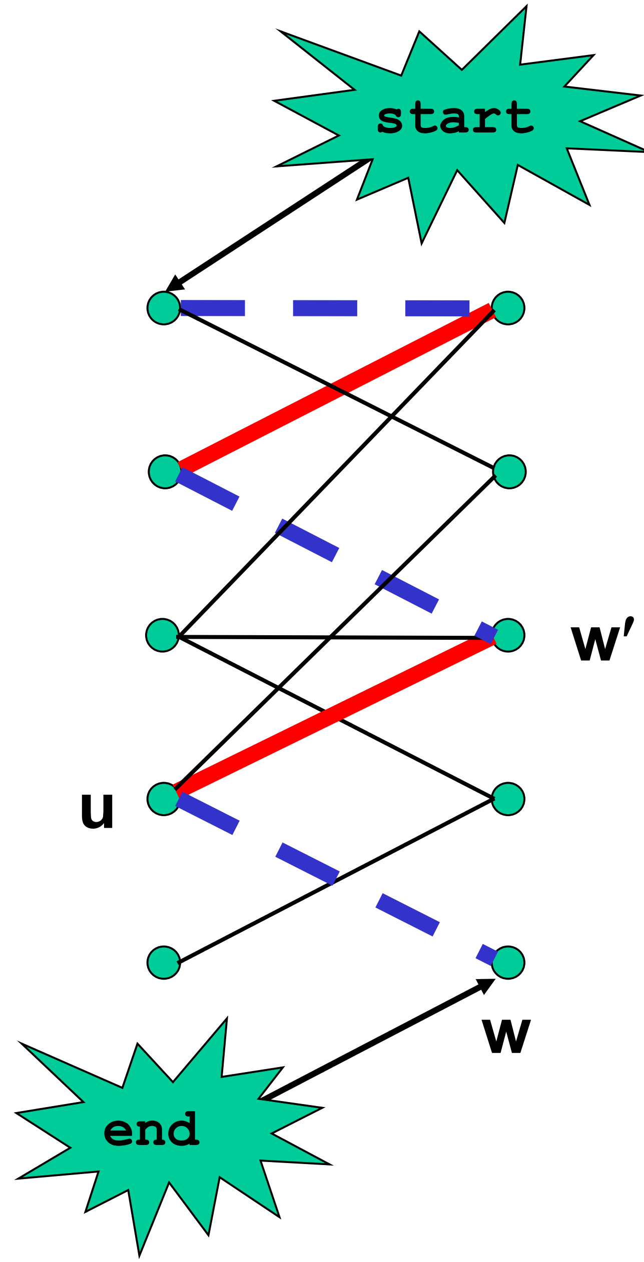
```
 /* Input: set of vertices vL to be searched for the start
  * of an augmenting path (list of vertices on LHS)
  * Output: the end vertex if an augmenting path is found
  * or null otherwise */
 public Vertex searchAP(List<Vertex> vL) {
    for (Vertex u : vL) {
       u.startVertex = false;
       for (Vertex w : u.adjacentV)
          w.visited = false;
    }
    List<Vertex> queue = new List<Vertex>();
    Vertex u;
    // continued on next slide
```

Searching for an augmenting path (cont)

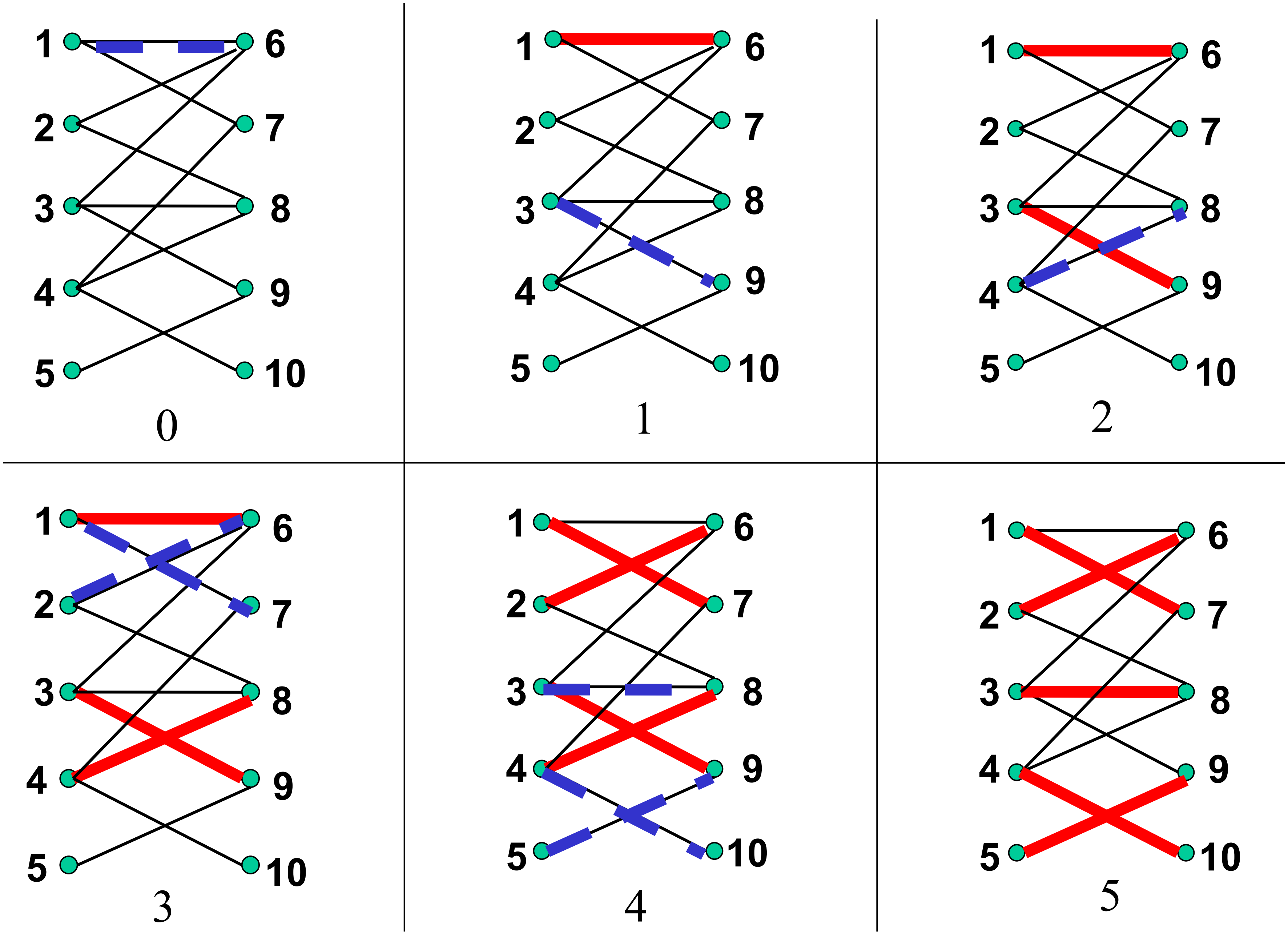
```
while ((u=getExposedUnvisited(vL))!=null)
{  // find an exposed and unvisited vertex u
   queue.add(u);
   u.startVertex = true; // first vertex in alternating path
   while (queue.size()>0)
   {  Vertex v = q.remove(0);    // from front of queue
      v.visited = true;
      for (Vertex w : v.adjacentV)
         if (!w.visited)
         {  w.visited = true;
            w.predecessor = v;
            if (w.mate == null)   // w is exposed
               return w;          // end of path
            else
               queue.add(w.mate);
         }
   }
}
return null; // no path found
```

Augmenting the matching along an augmenting path

```
public void augment(Vertex endVertex)
{   Vertex u, w, temp;
    w = endVertex;
    u = w.predecessor;
    while (!u.startVertex)
    {     temp = u.mate;
          u.mate = w;
          w.mate = u;
          w = temp;
          u = w.predecessor;
    }
    u.mate = w;
    w.mate = u;
}
```



A complete example



Algorithm analysis

- Let p and q be the numbers of vertices on the two sides U and W of the graph ($p \leq q$), $n=|V|$ and $m=|E|$

```
public List<Vertex> findMaxMatch(List<Vertex> vL)
{   Vertex end;
    while ( (end = searchAP(vL)) != null)
        augment(end);
        // the mate components of the vL Vertex
        // objects spell out the matching
    return vL;
}
```

- Searching for an augmenting path takes $O(p + m)$ time
- Augmenting along that path takes $O(m)$ time
- There are at most p iterations of the main loop
- So overall, the algorithm takes $O(p(p+m)) = O(n(n+m))$ time
- In general $m=O(n^2)$ so the algorithm is of $O(n^3)$ complexity

Summary

- **A maximum cardinality matching in a bipartite graph $G=(V,E)$ may be found in $O(n(n+m))$ time, where $n=|V|$ and $m=|E|$ using the augmenting path algorithm**
- **Faster method - $O(\sqrt{n}(n+m))$ algorithm**
  - **Hopcroft and Karp (1973)**
- **Also there is an efficient algorithm for finding a maximum cardinality matching in a general (not necessarily bipartite) graph**
  - **Edmonds (1965)**
- **Fastest known implementation of Edmonds' algorithm is also $O(\sqrt{n}(n+m))$**
  - **Micali and Vazirani (1980)**