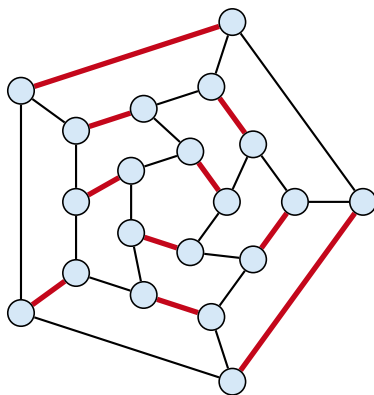


# Part V

## Matchings

# Matching

- ▶ Input: undirected graph  $G = (V, E)$ .
- ▶  $M \subseteq E$  is a **matching** if each node appears in at most one edge in  $M$ .
- ▶ Maximum Matching: find a matching of maximum cardinality



# 16 Bipartite Matching via Flows

## Which flow algorithm to use?

- ▶ Generic augmenting path:  $\mathcal{O}(m \text{val}(f^*)) = \mathcal{O}(mn)$ .
- ▶ Capacity scaling:  $\mathcal{O}(m^2 \log C) = \mathcal{O}(m^2)$ .
- ▶ Shortest augmenting path:  $\mathcal{O}(mn^2)$ .

For **unit capacity simple graphs** shortest augmenting path can be implemented in time  $\mathcal{O}(m\sqrt{n})$ .

# 17 Augmenting Paths for Matchings

## Definitions.

- ▶ Given a matching  $M$  in a graph  $G$ , a vertex that is not incident to any edge of  $M$  is called a **free vertex** w. r. .t.  $M$ .

# 17 Augmenting Paths for Matchings

## Definitions.

- ▶ Given a matching  $M$  in a graph  $G$ , a vertex that is not incident to any edge of  $M$  is called a **free vertex** w. r. .t.  $M$ .
- ▶ For a matching  $M$  a path  $P$  in  $G$  is called an **alternating path** if edges in  $M$  alternate with edges not in  $M$ .

# 17 Augmenting Paths for Matchings

## Definitions.

- ▶ Given a matching  $M$  in a graph  $G$ , a vertex that is not incident to any edge of  $M$  is called a **free vertex** w. r. .t.  $M$ .
- ▶ For a matching  $M$  a path  $P$  in  $G$  is called an **alternating path** if edges in  $M$  alternate with edges not in  $M$ .
- ▶ An alternating path is called an **augmenting path** for matching  $M$  if it ends at distinct free vertices.

# 17 Augmenting Paths for Matchings

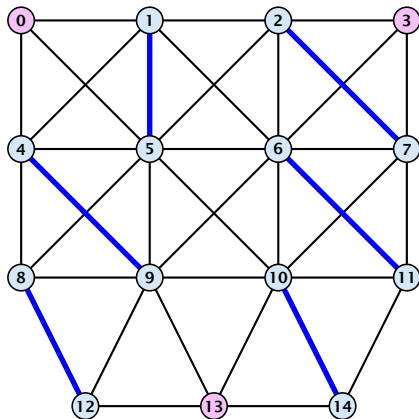
## Definitions.

- ▶ Given a matching  $M$  in a graph  $G$ , a vertex that is not incident to any edge of  $M$  is called a **free vertex** w. r. .t.  $M$ .
- ▶ For a matching  $M$  a path  $P$  in  $G$  is called an **alternating path** if edges in  $M$  alternate with edges not in  $M$ .
- ▶ An alternating path is called an **augmenting path** for matching  $M$  if it ends at distinct free vertices.

## Theorem 89

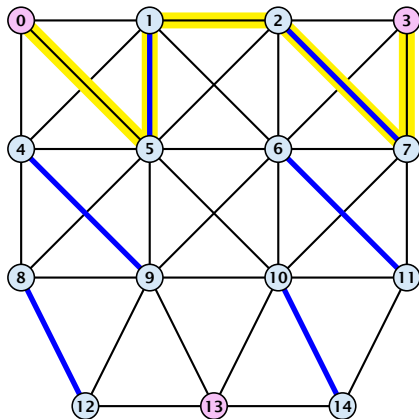
*A matching  $M$  is a maximum matching if and only if there is no augmenting path w. r. t.  $M$ .*

# Augmenting Paths in Action

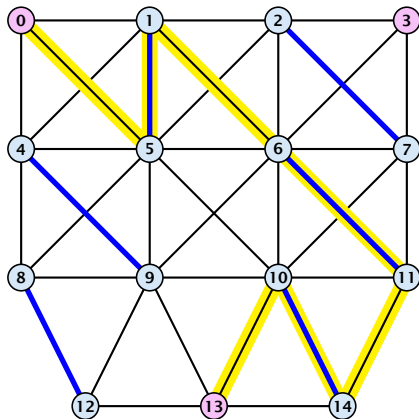




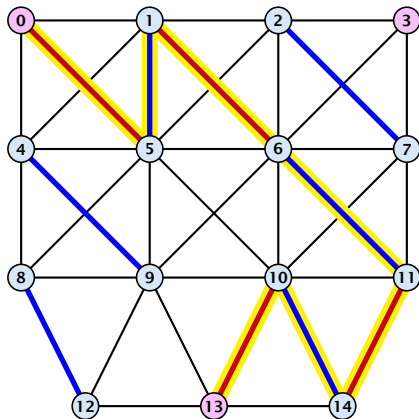
# Augmenting Paths in Action



# Augmenting Paths in Action

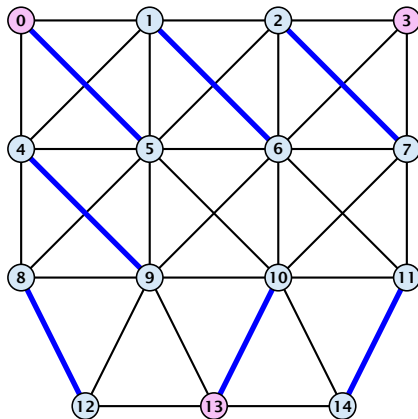


# Augmenting Paths in Action





# Augmenting Paths in Action



# 17 Augmenting Paths for Matchings

## Proof.

- ⇒ If  $M$  is maximum there is no augmenting path  $P$ , because we could switch matching and non-matching edges along  $P$ . This gives matching  $M' = M \oplus P$  with larger cardinality.

# 17 Augmenting Paths for Matchings

## Proof.

- ⇒ If  $M$  is maximum there is no augmenting path  $P$ , because we could switch matching and non-matching edges along  $P$ . This gives matching  $M' = M \oplus P$  with larger cardinality.
- ⇐ Suppose there is a matching  $M'$  with larger cardinality. Consider the graph  $H$  with edge-set  $M' \oplus M$  (i.e., only edges that are in either  $M$  or  $M'$  but not in both).

# 17 Augmenting Paths for Matchings

## Proof.

- ⇒ If  $M$  is maximum there is no augmenting path  $P$ , because we could switch matching and non-matching edges along  $P$ . This gives matching  $M' = M \oplus P$  with larger cardinality.
- ⇐ Suppose there is a matching  $M'$  with larger cardinality. Consider the graph  $H$  with edge-set  $M' \oplus M$  (i.e., only edges that are in either  $M$  or  $M'$  but not in both).

Each vertex can be incident to at most two edges (one from  $M$  and one from  $M'$ ). Hence, the connected components are alternating cycles or alternating path.



# 17 Augmenting Paths for Matchings

## Proof.

- ⇒ If  $M$  is maximum there is no augmenting path  $P$ , because we could switch matching and non-matching edges along  $P$ . This gives matching  $M' = M \oplus P$  with larger cardinality.
- ⇐ Suppose there is a matching  $M'$  with larger cardinality. Consider the graph  $H$  with edge-set  $M' \oplus M$  (i.e., only edges that are in either  $M$  or  $M'$  but not in both).

Each vertex can be incident to at most two edges (one from  $M$  and one from  $M'$ ). Hence, the connected components are alternating cycles or alternating path.

As  $|M'| > |M|$  there is one connected component that is a path  $P$  for which both endpoints are incident to edges from  $M'$ .  $P$  is an augmenting path.

# 17 Augmenting Paths for Matchings

## **Algorithmic idea:**

As long as you find an augmenting path augment your matching using this path. When you arrive at a matching for which no augmenting path exists you have a maximum matching.

# 17 Augmenting Paths for Matchings

## Algorithmic idea:

As long as you find an augmenting path augment your matching using this path. When you arrive at a matching for which no augmenting path exists you have a maximum matching.

## Theorem 90

Let  $G$  be a graph,  $M$  a matching in  $G$ , and let  $u$  be a free vertex w.r.t.  $M$ . Further let  $P$  denote an augmenting path w.r.t.  $M$  and let  $M' = M \oplus P$  denote the matching resulting from augmenting  $M$  with  $P$ . If there was no augmenting path starting at  $u$  in  $M$  then there is no augmenting path starting at  $u$  in  $M'$ .

The above theorem allows for an easier implementation of an augmenting path algorithm. Once we checked for augmenting paths starting from  $u$  we don't have to check for such paths in future rounds.

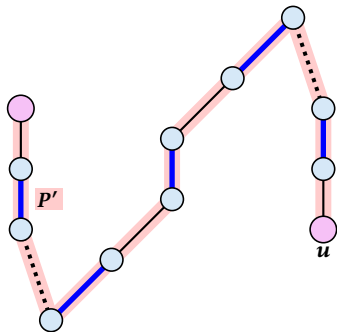
# 17 Augmenting Paths for Matchings

## Proof

# 17 Augmenting Paths for Matchings

## Proof

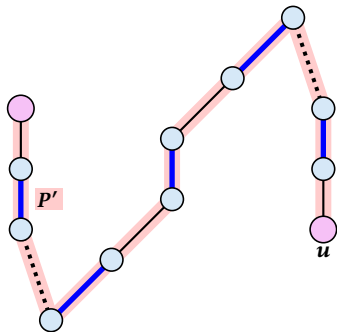
- ▶ Assume there is an augmenting path  $P'$  w.r.t.  $M'$  starting at  $u$ .



# 17 Augmenting Paths for Matchings

## Proof

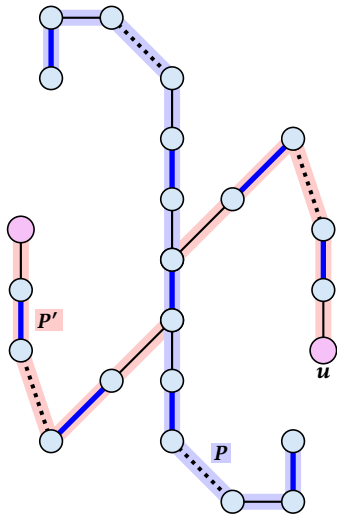
- ▶ Assume there is an augmenting path  $P'$  w.r.t.  $M'$  starting at  $u$ .
- ▶ If  $P'$  and  $P$  are node-disjoint,  $P'$  is also augmenting path w.r.t.  $M$  ( $\neq$ ).



# 17 Augmenting Paths for Matchings

## Proof

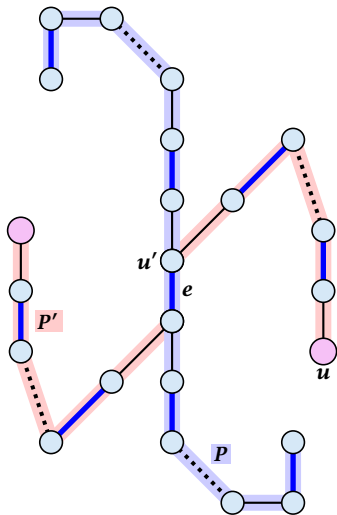
- ▶ Assume there is an augmenting path  $P'$  w.r.t.  $M'$  starting at  $u$ .
- ▶ If  $P'$  and  $P$  are node-disjoint,  $P'$  is also augmenting path w.r.t.  $M$  ( $\neq$ ).



# 17 Augmenting Paths for Matchings

## Proof

- ▶ Assume there is an augmenting path  $P'$  w.r.t.  $M'$  starting at  $u$ .
- ▶ If  $P'$  and  $P$  are node-disjoint,  $P'$  is also augmenting path w.r.t.  $M$  ( $\neq$ ).
- ▶ Let  $u'$  be the **first** node on  $P'$  that is in  $P$ , and let  $e$  be the matching edge from  $M'$  incident to  $u'$ .

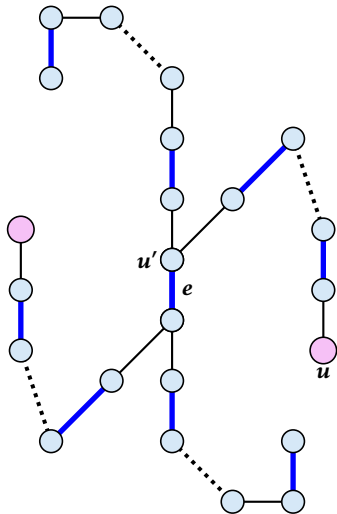




# 17 Augmenting Paths for Matchings

## Proof

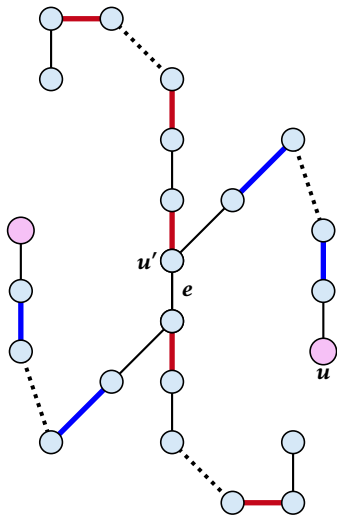
- ▶ Assume there is an augmenting path  $P'$  w.r.t.  $M'$  starting at  $u$ .
- ▶ If  $P'$  and  $P$  are node-disjoint,  $P'$  is also augmenting path w.r.t.  $M$  ( $\neq$ ).
- ▶ Let  $u'$  be the **first** node on  $P'$  that is in  $P$ , and let  $e$  be the matching edge from  $M'$  incident to  $u'$ .



# 17 Augmenting Paths for Matchings

## Proof

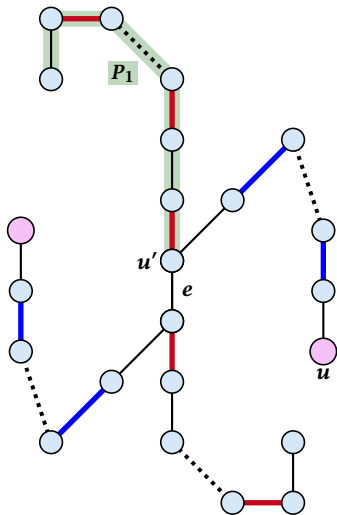
- ▶ Assume there is an augmenting path  $P'$  w.r.t.  $M'$  starting at  $u$ .
- ▶ If  $P'$  and  $P$  are node-disjoint,  $P'$  is also augmenting path w.r.t.  $M$  ( $\neq$ ).
- ▶ Let  $u'$  be the **first** node on  $P'$  that is in  $P$ , and let  $e$  be the matching edge from  $M'$  incident to  $u'$ .



# 17 Augmenting Paths for Matchings

## Proof

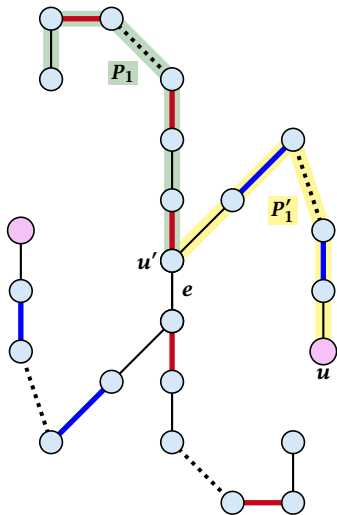
- ▶ Assume there is an augmenting path  $P'$  w.r.t.  $M'$  starting at  $u$ .
- ▶ If  $P'$  and  $P$  are node-disjoint,  $P'$  is also augmenting path w.r.t.  $M$  ( $\neq$ ).
- ▶ Let  $u'$  be the **first** node on  $P'$  that is in  $P$ , and let  $e$  be the matching edge from  $M'$  incident to  $u'$ .
- ▶  $u'$  splits  $P$  into two parts one of which does not contain  $e$ . Call this part  $P_1$ . Denote the sub-path of  $P'$  from  $u$  to  $u'$  with  $P'_1$ .



# 17 Augmenting Paths for Matchings

## Proof

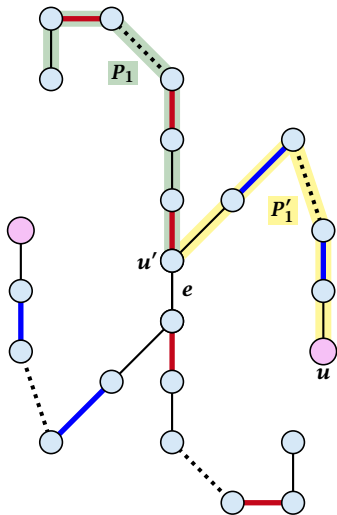
- ▶ Assume there is an augmenting path  $P'$  w.r.t.  $M'$  starting at  $u$ .
- ▶ If  $P'$  and  $P$  are node-disjoint,  $P'$  is also augmenting path w.r.t.  $M$  ( $\neq$ ).
- ▶ Let  $u'$  be the **first** node on  $P'$  that is in  $P$ , and let  $e$  be the matching edge from  $M'$  incident to  $u'$ .
- ▶  $u'$  splits  $P$  into two parts one of which does not contain  $e$ . Call this part  $P_1$ . Denote the sub-path of  $P'$  from  $u$  to  $u'$  with  $P'_1$ .



# 17 Augmenting Paths for Matchings

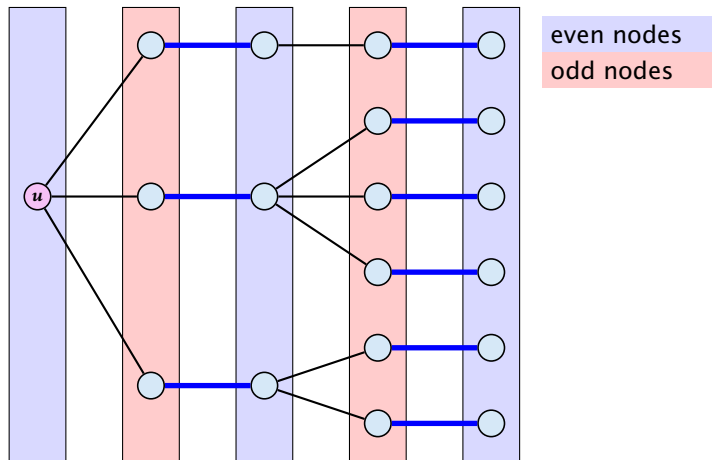
## Proof

- ▶ Assume there is an augmenting path  $P'$  w.r.t.  $M'$  starting at  $u$ .
- ▶ If  $P'$  and  $P$  are node-disjoint,  $P'$  is also augmenting path w.r.t.  $M$  ( $\cancel{\neq}$ ).
- ▶ Let  $u'$  be the **first** node on  $P'$  that is in  $P$ , and let  $e$  be the matching edge from  $M'$  incident to  $u'$ .
- ▶  $u'$  splits  $P$  into two parts one of which does not contain  $e$ . Call this part  $P_1$ . Denote the sub-path of  $P'$  from  $u$  to  $u'$  with  $P'_1$ .
- ▶  $P_1 \circ P'_1$  is augmenting path in  $M$  ( $\cancel{\neq}$ ).



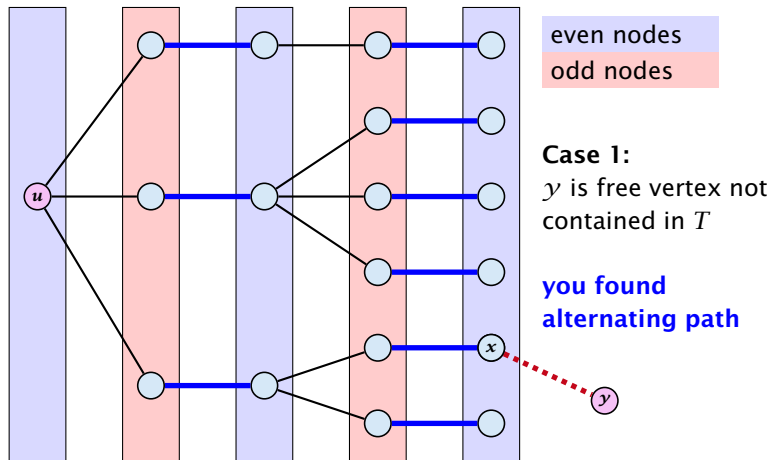
# How to find an augmenting path?

Construct an alternating tree.



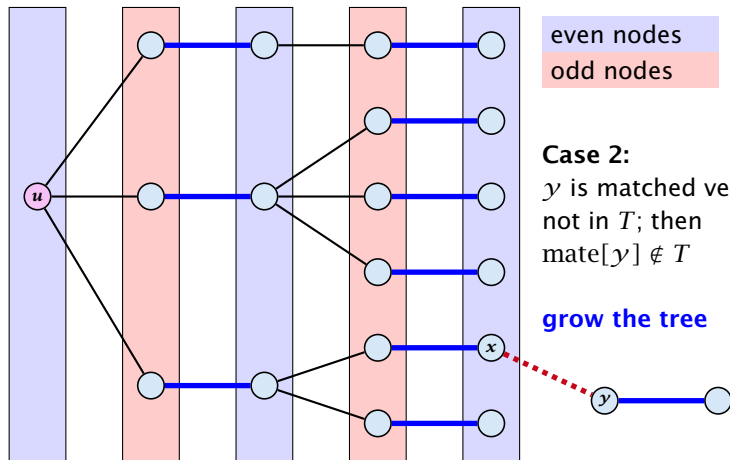
# How to find an augmenting path?

Construct an alternating tree.



# How to find an augmenting path?

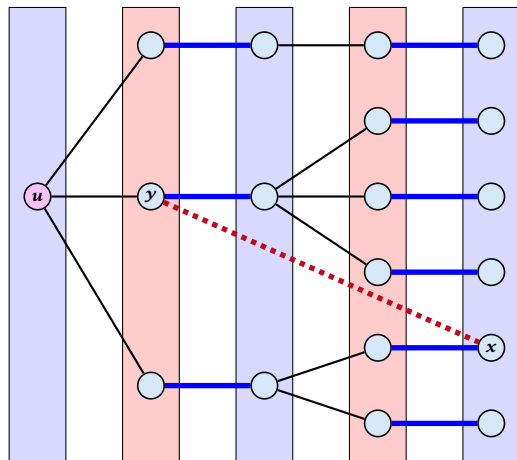
Construct an alternating tree.





# How to find an augmenting path?

Construct an alternating tree.



even nodes

odd nodes

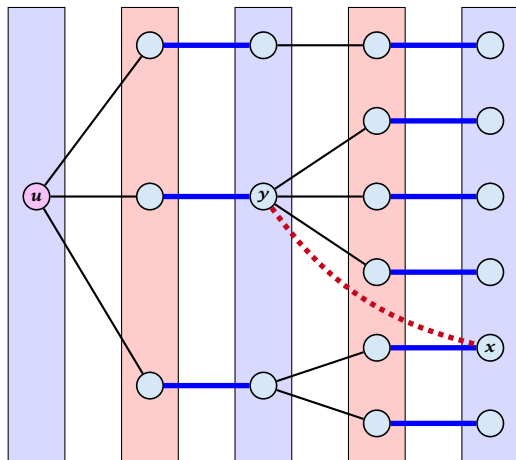
**Case 3:**

$y$  is already contained  
in  $T$  as an odd vertex

**ignore successor  $y$**

# How to find an augmenting path?

Construct an alternating tree.



even nodes

odd nodes

**Case 4:**

$y$  is already contained  
in  $T$  as an even vertex

can't ignore  $y$

does not happen in  
bipartite graphs

**Algorithm 49** BiMatch( $G, match$ )

```
1: for  $x \in V$  do  $mate[x] \leftarrow 0$ ;  
2:  $r \leftarrow 0$ ;  $free \leftarrow n$ ;  
3: while  $free \geq 1$  and  $r < n$  do  
4:    $r \leftarrow r + 1$   
5:   if  $mate[r] = 0$  then  
6:     for  $i = 1$  to  $n$  do  $parent[i'] \leftarrow 0$   
7:      $Q \leftarrow \emptyset$ ;  $Q.append(r)$ ;  $aug \leftarrow false$ ;  
8:     while  $aug = false$  and  $Q \neq \emptyset$  do  
9:        $x \leftarrow Q.dequeue()$ ;  
10:      for  $y \in A_x$  do  
11:        if  $mate[y] = 0$  then  
12:           $augm(mate, parent, y)$ ;  
13:           $aug \leftarrow true$ ;  
14:           $free \leftarrow free - 1$ ;  
15:        else  
16:          if  $parent[y] = 0$  then  
17:             $parent[y] \leftarrow x$ ;  
18:             $Q.enqueue(mate[y])$ ;
```

graph  $G = (S \cup S', E)$

$S = \{1, \dots, n\}$

$S' = \{1', \dots, n'\}$

### Algorithm 49 BiMatch( $G, match$ )

```
1: for  $x \in V$  do  $mate[x] \leftarrow 0$ ;  
2:  $r \leftarrow 0$ ;  $free \leftarrow n$ ;  
3: while  $free \geq 1$  and  $r < n$  do  
4:    $r \leftarrow r + 1$   
5:   if  $mate[r] = 0$  then  
6:     for  $i = 1$  to  $n$  do  $parent[i'] \leftarrow 0$   
7:      $Q \leftarrow \emptyset$ ;  $Q.append(r)$ ;  $aug \leftarrow false$ ;  
8:     while  $aug = false$  and  $Q \neq \emptyset$  do  
9:        $x \leftarrow Q.dequeue()$ ;  
10:      for  $y \in A_x$  do  
11:        if  $mate[y] = 0$  then  
12:           $augm(mate, parent, y)$ ;  
13:           $aug \leftarrow true$ ;  
14:           $free \leftarrow free - 1$ ;  
15:        else  
16:          if  $parent[y] = 0$  then  
17:             $parent[y] \leftarrow x$ ;  
18:             $Q.enqueue(mate[y])$ ;
```

start with an  
empty matching

**Algorithm 49** BiMatch( $G, match$ )

```
1: for  $x \in V$  do  $mate[x] \leftarrow 0$ ;  
2:  $r \leftarrow 0$ ;  $free \leftarrow n$ ;  
3: while  $free \geq 1$  and  $r < n$  do  
4:    $r \leftarrow r + 1$   
5:   if  $mate[r] = 0$  then  
6:     for  $i = 1$  to  $n$  do  $parent[i'] \leftarrow 0$   
7:      $Q \leftarrow \emptyset$ ;  $Q.append(r)$ ;  $aug \leftarrow false$ ;  
8:     while  $aug = false$  and  $Q \neq \emptyset$  do  
9:        $x \leftarrow Q.dequeue()$ ;  
10:      for  $y \in A_x$  do  
11:        if  $mate[y] = 0$  then  
12:           $augm(mate, parent, y)$ ;  
13:           $aug \leftarrow true$ ;  
14:           $free \leftarrow free - 1$ ;  
15:        else  
16:          if  $parent[y] = 0$  then  
17:             $parent[y] \leftarrow x$ ;  
18:             $Q.enqueue(mate[y])$ ;
```

*free*: number of  
unmatched nodes in  $S$

*r*: root of current tree

**Algorithm 49** BiMatch( $G, match$ )

```
1: for  $x \in V$  do  $mate[x] \leftarrow 0$ ;  
2:  $r \leftarrow 0$ ;  $free \leftarrow n$ ;  
3: while  $free \geq 1$  and  $r < n$  do  
4:    $r \leftarrow r + 1$   
5:   if  $mate[r] = 0$  then  
6:     for  $i = 1$  to  $n$  do  $parent[i'] \leftarrow 0$   
7:      $Q \leftarrow \emptyset$ ;  $Q.append(r)$ ;  $aug \leftarrow false$ ;  
8:     while  $aug = false$  and  $Q \neq \emptyset$  do  
9:        $x \leftarrow Q.dequeue()$ ;  
10:      for  $y \in A_x$  do  
11:        if  $mate[y] = 0$  then  
12:           $augm(mate, parent, y)$ ;  
13:           $aug \leftarrow true$ ;  
14:           $free \leftarrow free - 1$ ;  
15:        else  
16:          if  $parent[y] = 0$  then  
17:             $parent[y] \leftarrow x$ ;  
18:             $Q.enqueue(mate[y])$ ;
```

as long as there are  
unmatched nodes and  
we did not yet try to  
grow from all nodes we  
continue

**Algorithm 49** BiMatch( $G, match$ )

```
1: for  $x \in V$  do  $mate[x] \leftarrow 0$ ;  
2:  $r \leftarrow 0$ ;  $free \leftarrow n$ ;  
3: while  $free \geq 1$  and  $r < n$  do  
4:    $r \leftarrow r + 1$   
5:   if  $mate[r] = 0$  then  
6:     for  $i = 1$  to  $n$  do  $parent[i'] \leftarrow 0$   
7:      $Q \leftarrow \emptyset$ ;  $Q.append(r)$ ;  $aug \leftarrow false$ ;  
8:     while  $aug = false$  and  $Q \neq \emptyset$  do  
9:        $x \leftarrow Q.dequeue()$ ;  
10:      for  $y \in A_x$  do  
11:        if  $mate[y] = 0$  then  
12:           $augm(mate, parent, y)$ ;  
13:           $aug \leftarrow true$ ;  
14:           $free \leftarrow free - 1$ ;  
15:        else  
16:          if  $parent[y] = 0$  then  
17:             $parent[y] \leftarrow x$ ;  
18:             $Q.enqueue(mate[y])$ ;
```

$r$  is the new node that we grow from.

**Algorithm 49** BiMatch( $G, match$ )

```
1: for  $x \in V$  do  $mate[x] \leftarrow 0$ ;  
2:  $r \leftarrow 0$ ;  $free \leftarrow n$ ;  
3: while  $free \geq 1$  and  $r < n$  do  
4:    $r \leftarrow r + 1$   
5:   if  $mate[r] = 0$  then  
6:     for  $i = 1$  to  $n$  do  $parent[i'] \leftarrow 0$   
7:      $Q \leftarrow \emptyset$ ;  $Q.append(r)$ ;  $aug \leftarrow false$ ;  
8:     while  $aug = false$  and  $Q \neq \emptyset$  do  
9:        $x \leftarrow Q.dequeue()$ ;  
10:      for  $y \in A_x$  do  
11:        if  $mate[y] = 0$  then  
12:           $augm(mate, parent, y)$ ;  
13:           $aug \leftarrow true$ ;  
14:           $free \leftarrow free - 1$ ;  
15:        else  
16:          if  $parent[y] = 0$  then  
17:             $parent[y] \leftarrow x$ ;  
18:             $Q.enqueue(mate[y])$ ;
```

If  $r$  is free start tree construction



**Algorithm 49** BiMatch( $G, match$ )

```
1: for  $x \in V$  do  $mate[x] \leftarrow 0$ ;  
2:  $r \leftarrow 0$ ;  $free \leftarrow n$ ;  
3: while  $free \geq 1$  and  $r < n$  do  
4:    $r \leftarrow r + 1$   
5:   if  $mate[r] = 0$  then  
6:     for  $i = 1$  to  $n$  do  $parent[i'] \leftarrow 0$   
7:      $Q \leftarrow \emptyset$ ;  $Q.append(r)$ ;  $aug \leftarrow false$ ;  
8:     while  $aug = false$  and  $Q \neq \emptyset$  do  
9:        $x \leftarrow Q.dequeue()$ ;  
10:      for  $y \in A_x$  do  
11:        if  $mate[y] = 0$  then  
12:           $augm(mate, parent, y)$ ;  
13:           $aug \leftarrow true$ ;  
14:           $free \leftarrow free - 1$ ;  
15:        else  
16:          if  $parent[y] = 0$  then  
17:             $parent[y] \leftarrow x$ ;  
18:             $Q.enqueue(mate[y])$ ;
```

Initialize an empty tree.  
Note that only nodes  $i'$   
have parent pointers.

### Algorithm 49 BiMatch( $G, match$ )

```
1: for  $x \in V$  do  $mate[x] \leftarrow 0$ ;  
2:  $r \leftarrow 0$ ;  $free \leftarrow n$ ;  
3: while  $free \geq 1$  and  $r < n$  do  
4:    $r \leftarrow r + 1$   
5:   if  $mate[r] = 0$  then  
6:     for  $i = 1$  to  $n$  do  $parent[i'] \leftarrow 0$   
7:      $Q \leftarrow \emptyset$ ;  $Q.append(r)$ ;  $aug \leftarrow false$ ;  
8:     while  $aug = false$  and  $Q \neq \emptyset$  do  
9:        $x \leftarrow Q.dequeue()$ ;  
10:      for  $y \in A_x$  do  
11:        if  $mate[y] = 0$  then  
12:           $augm(mate, parent, y)$ ;  
13:           $aug \leftarrow true$ ;  
14:           $free \leftarrow free - 1$ ;  
15:        else  
16:          if  $parent[y] = 0$  then  
17:             $parent[y] \leftarrow x$ ;  
18:             $Q.enqueue(mate[y])$ ;
```

$Q$  is a queue (BFS!!!).

$aug$  is a Boolean that stores whether we already found an augmenting path.

**Algorithm 49** BiMatch( $G, match$ )

```
1: for  $x \in V$  do  $mate[x] \leftarrow 0$ ;  
2:  $r \leftarrow 0$ ;  $free \leftarrow n$ ;  
3: while  $free \geq 1$  and  $r < n$  do  
4:    $r \leftarrow r + 1$   
5:   if  $mate[r] = 0$  then  
6:     for  $i = 1$  to  $n$  do  $parent[i'] \leftarrow 0$   
7:      $Q \leftarrow \emptyset$ ;  $Q.append(r)$ ;  $aug \leftarrow false$ ;  
8:     while  $aug = false$  and  $Q \neq \emptyset$  do  
9:        $x \leftarrow Q.dequeue()$ ;  
10:      for  $y \in A_x$  do  
11:        if  $mate[y] = 0$  then  
12:           $augm(mate, parent, y)$ ;  
13:           $aug \leftarrow true$ ;  
14:           $free \leftarrow free - 1$ ;  
15:        else  
16:          if  $parent[y] = 0$  then  
17:             $parent[y] \leftarrow x$ ;  
18:             $Q.enqueue(mate[y])$ ;
```

as long as we did not augment and there are still unexamined leaves continue...

### Algorithm 49 BiMatch( $G, match$ )

```
1: for  $x \in V$  do  $mate[x] \leftarrow 0$ ;  
2:  $r \leftarrow 0$ ;  $free \leftarrow n$ ;  
3: while  $free \geq 1$  and  $r < n$  do  
4:    $r \leftarrow r + 1$   
5:   if  $mate[r] = 0$  then  
6:     for  $i = 1$  to  $n$  do  $parent[i'] \leftarrow 0$   
7:      $Q \leftarrow \emptyset$ ;  $Q.append(r)$ ;  $aug \leftarrow false$ ;  
8:     while  $aug = false$  and  $Q \neq \emptyset$  do  
9:        $x \leftarrow Q.dequeue()$ ;  
10:      for  $y \in A_x$  do  
11:        if  $mate[y] = 0$  then  
12:           $augm(mate, parent, y)$ ;  
13:           $aug \leftarrow true$ ;  
14:           $free \leftarrow free - 1$ ;  
15:        else  
16:          if  $parent[y] = 0$  then  
17:             $parent[y] \leftarrow x$ ;  
18:             $Q.enqueue(mate[y])$ ;
```

take next unexamined  
leaf

**Algorithm 49** BiMatch( $G, match$ )

```
1: for  $x \in V$  do  $mate[x] \leftarrow 0$ ;  
2:  $r \leftarrow 0$ ;  $free \leftarrow n$ ;  
3: while  $free \geq 1$  and  $r < n$  do  
4:    $r \leftarrow r + 1$   
5:   if  $mate[r] = 0$  then  
6:     for  $i = 1$  to  $n$  do  $parent[i'] \leftarrow 0$   
7:      $Q \leftarrow \emptyset$ ;  $Q.append(r)$ ;  $aug \leftarrow false$ ;  
8:     while  $aug = false$  and  $Q \neq \emptyset$  do  
9:        $x \leftarrow Q.dequeue()$ ;  
10:      for  $y \in A_x$  do  
11:        if  $mate[y] = 0$  then  
12:           $augm(mate, parent, y)$ ;  
13:           $aug \leftarrow true$ ;  
14:           $free \leftarrow free - 1$ ;  
15:        else  
16:          if  $parent[y] = 0$  then  
17:             $parent[y] \leftarrow x$ ;  
18:             $Q.enqueue(mate[y])$ ;
```

if  $x$  has unmatched neighbour we found an augmenting path (note that  $y \neq r$  because we are in a bipartite graph)

### Algorithm 49 BiMatch( $G, match$ )

```
1: for  $x \in V$  do  $mate[x] \leftarrow 0$ ;  
2:  $r \leftarrow 0$ ;  $free \leftarrow n$ ;  
3: while  $free \geq 1$  and  $r < n$  do  
4:    $r \leftarrow r + 1$   
5:   if  $mate[r] = 0$  then  
6:     for  $i = 1$  to  $n$  do  $parent[i'] \leftarrow 0$   
7:      $Q \leftarrow \emptyset$ ;  $Q.append(r)$ ;  $aug \leftarrow false$ ;  
8:     while  $aug = false$  and  $Q \neq \emptyset$  do  
9:        $x \leftarrow Q.dequeue()$ ;  
10:      for  $y \in A_x$  do  
11:        if  $mate[y] = 0$  then  
12:           $augm(mate, parent, y)$ ;  
13:           $aug \leftarrow true$ ;  
14:           $free \leftarrow free - 1$ ;  
15:        else  
16:          if  $parent[y] = 0$  then  
17:             $parent[y] \leftarrow x$ ;  
18:             $Q.enqueue(mate[y])$ ;
```

do an augmentation...

**Algorithm 49** BiMatch( $G, match$ )

```
1: for  $x \in V$  do  $mate[x] \leftarrow 0$ ;  
2:  $r \leftarrow 0$ ;  $free \leftarrow n$ ;  
3: while  $free \geq 1$  and  $r < n$  do  
4:    $r \leftarrow r + 1$   
5:   if  $mate[r] = 0$  then  
6:     for  $i = 1$  to  $n$  do  $parent[i'] \leftarrow 0$   
7:      $Q \leftarrow \emptyset$ ;  $Q.append(r)$ ;  $aug \leftarrow false$ ;  
8:     while  $aug = false$  and  $Q \neq \emptyset$  do  
9:        $x \leftarrow Q.dequeue()$ ;  
10:      for  $y \in A_x$  do  
11:        if  $mate[y] = 0$  then  
12:           $augm(mate, parent, y)$ ;  
13:           $aug \leftarrow true$ ;  
14:           $free \leftarrow free - 1$ ;  
15:      else  
16:        if  $parent[y] = 0$  then  
17:           $parent[y] \leftarrow x$ ;  
18:           $Q.enqueue(mate[y])$ ;
```

setting  $aug = true$   
ensures that the tree  
construction will not  
continue

**Algorithm 49 BiMatch( $G, match$ )**

```
1: for  $x \in V$  do  $mate[x] \leftarrow 0$ ;  
2:  $r \leftarrow 0$ ;  $free \leftarrow n$ ;  
3: while  $free \geq 1$  and  $r < n$  do  
4:    $r \leftarrow r + 1$   
5:   if  $mate[r] = 0$  then  
6:     for  $i = 1$  to  $n$  do  $parent[i'] \leftarrow 0$   
7:      $Q \leftarrow \emptyset$ ;  $Q.append(r)$ ;  $aug \leftarrow false$ ;  
8:     while  $aug = false$  and  $Q \neq \emptyset$  do  
9:        $x \leftarrow Q.dequeue()$ ;  
10:      for  $y \in A_x$  do  
11:        if  $mate[y] = 0$  then  
12:           $augm(mate, parent, y)$ ;  
13:           $aug \leftarrow true$ ;  
14:           $free \leftarrow free - 1$ ;  
15:        else  
16:          if  $parent[y] = 0$  then  
17:             $parent[y] \leftarrow x$ ;  
18:             $Q.enqueue(mate[y])$ ;
```

reduce number of free  
nodes



**Algorithm 49 BiMatch( $G, match$ )**

```
1: for  $x \in V$  do  $mate[x] \leftarrow 0$ ;  
2:  $r \leftarrow 0$ ;  $free \leftarrow n$ ;  
3: while  $free \geq 1$  and  $r < n$  do  
4:    $r \leftarrow r + 1$   
5:   if  $mate[r] = 0$  then  
6:     for  $i = 1$  to  $n$  do  $parent[i'] \leftarrow 0$   
7:      $Q \leftarrow \emptyset$ ;  $Q.append(r)$ ;  $aug \leftarrow false$ ;  
8:     while  $aug = false$  and  $Q \neq \emptyset$  do  
9:        $x \leftarrow Q.dequeue()$ ;  
10:      for  $y \in A_x$  do  
11:        if  $mate[y] = 0$  then  
12:           $augm(mate, parent, y)$ ;  
13:           $aug \leftarrow true$ ;  
14:           $free \leftarrow free - 1$ ;  
15:        else  
16:          if  $parent[y] = 0$  then  
17:             $parent[y] \leftarrow x$ ;  
18:             $Q.enqueue(mate[y])$ ;
```

if  $y$  is not in the tree yet

### Algorithm 49 BiMatch( $G, match$ )

```
1: for  $x \in V$  do  $mate[x] \leftarrow 0$ ;  
2:  $r \leftarrow 0$ ;  $free \leftarrow n$ ;  
3: while  $free \geq 1$  and  $r < n$  do  
4:    $r \leftarrow r + 1$   
5:   if  $mate[r] = 0$  then  
6:     for  $i = 1$  to  $n$  do  $parent[i'] \leftarrow 0$   
7:      $Q \leftarrow \emptyset$ ;  $Q.append(r)$ ;  $aug \leftarrow false$ ;  
8:     while  $aug = false$  and  $Q \neq \emptyset$  do  
9:        $x \leftarrow Q.dequeue()$ ;  
10:      for  $y \in A_x$  do  
11:        if  $mate[y] = 0$  then  
12:           $augm(mate, parent, y)$ ;  
13:           $aug \leftarrow true$ ;  
14:           $free \leftarrow free - 1$ ;  
15:        else  
16:          if  $parent[y] = 0$  then  
17:             $parent[y] \leftarrow x$ ;  
18:             $Q.enqueue(mate[y])$ ;
```

...put it into the tree

### Algorithm 49 BiMatch( $G, match$ )

```
1: for  $x \in V$  do  $mate[x] \leftarrow 0$ ;  
2:  $r \leftarrow 0$ ;  $free \leftarrow n$ ;  
3: while  $free \geq 1$  and  $r < n$  do  
4:    $r \leftarrow r + 1$   
5:   if  $mate[r] = 0$  then  
6:     for  $i = 1$  to  $n$  do  $parent[i'] \leftarrow 0$   
7:      $Q \leftarrow \emptyset$ ;  $Q.append(r)$ ;  $aug \leftarrow false$ ;  
8:     while  $aug = false$  and  $Q \neq \emptyset$  do  
9:        $x \leftarrow Q.dequeue()$ ;  
10:      for  $y \in A_x$  do  
11:        if  $mate[y] = 0$  then  
12:           $augm(mate, parent, y)$ ;  
13:           $aug \leftarrow true$ ;  
14:           $free \leftarrow free - 1$ ;  
15:        else  
16:          if  $parent[y] = 0$  then  
17:             $parent[y] \leftarrow x$ ;  
18:             $Q.enqueue(mate[y])$ ;
```

add its buddy to the set  
of unexamined leaves

# 18 Weighted Bipartite Matching

## Weighted Bipartite Matching/Assignment

- ▶ Input: undirected, bipartite graph  $G = L \cup R, E$ .
- ▶ an edge  $e = (\ell, r)$  has weight  $w_e \geq 0$
- ▶ find a matching of maximum weight, where the weight of a matching is the sum of the weights of its edges

## Simplifying Assumptions (wlog [why?]):

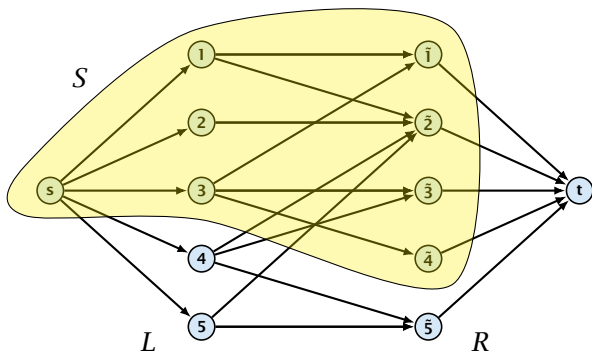
- ▶ assume that  $|L| = |R| = n$
- ▶ assume that there is an edge between every pair of nodes  $(\ell, r) \in V \times V$
- ▶ can assume goal is to construct maximum weight **perfect** matching

# Weighted Bipartite Matching

## Theorem 91 (Halls Theorem)

A bipartite graph  $G = (L \cup R, E)$  has a perfect matching if and only if for all sets  $S \subseteq L$ ,  $|\Gamma(S)| \geq |S|$ , where  $\Gamma(S)$  denotes the set of nodes in  $R$  that have a neighbour in  $S$ .

# 18 Weighted Bipartite Matching



# Halls Theorem

## Proof:

- ← Of course, the condition is necessary as otherwise not all nodes in  $S$  could be matched to different neighbours.

# Halls Theorem

## Proof:

- ⇐ Of course, the condition is necessary as otherwise not all nodes in  $S$  could be matched to different neighbours.
- ⇒ For the other direction we need to argue that the minimum cut in the graph  $G'$  is at least  $|L|$ .



# Halls Theorem

## Proof:

- ⇐ Of course, the condition is necessary as otherwise not all nodes in  $S$  could be matched to different neighbours.
- ⇒ For the other direction we need to argue that the minimum cut in the graph  $G'$  is at least  $|L|$ .
  - ▶ Let  $S$  denote a minimum cut and let  $L_S \stackrel{\text{def}}{=} L \cap S$  and  $R_S \stackrel{\text{def}}{=} R \cap S$  denote the portion of  $S$  inside  $L$  and  $R$ , respectively.

# Halls Theorem

## Proof:

- ⇐ Of course, the condition is necessary as otherwise not all nodes in  $S$  could be matched to different neighbours.
- ⇒ For the other direction we need to argue that the minimum cut in the graph  $G'$  is at least  $|L|$ .
  - ▶ Let  $S$  denote a minimum cut and let  $L_S \stackrel{\text{def}}{=} L \cap S$  and  $R_S \stackrel{\text{def}}{=} R \cap S$  denote the portion of  $S$  inside  $L$  and  $R$ , respectively.
  - ▶ Clearly, all neighbours of nodes in  $L_S$  have to be in  $S$ , as otherwise we would cut an edge of infinite capacity.

# Halls Theorem

## Proof:

- ⇐ Of course, the condition is necessary as otherwise not all nodes in  $S$  could be matched to different neighbours.
- ⇒ For the other direction we need to argue that the minimum cut in the graph  $G'$  is at least  $|L|$ .
  - ▶ Let  $S$  denote a minimum cut and let  $L_S \stackrel{\text{def}}{=} L \cap S$  and  $R_S \stackrel{\text{def}}{=} R \cap S$  denote the portion of  $S$  inside  $L$  and  $R$ , respectively.
  - ▶ Clearly, all neighbours of nodes in  $L_S$  have to be in  $S$ , as otherwise we would cut an edge of infinite capacity.
  - ▶ This gives  $R_S \geq |\Gamma(L_S)|$ .

# Halls Theorem

## Proof:

- ⇐ Of course, the condition is necessary as otherwise not all nodes in  $S$  could be matched to different neighbours.
- ⇒ For the other direction we need to argue that the minimum cut in the graph  $G'$  is at least  $|L|$ .
  - ▶ Let  $S$  denote a minimum cut and let  $L_S \stackrel{\text{def}}{=} L \cap S$  and  $R_S \stackrel{\text{def}}{=} R \cap S$  denote the portion of  $S$  inside  $L$  and  $R$ , respectively.
  - ▶ Clearly, all neighbours of nodes in  $L_S$  have to be in  $S$ , as otherwise we would cut an edge of infinite capacity.
  - ▶ This gives  $R_S \geq |\Gamma(L_S)|$ .
  - ▶ The size of the cut is  $|L| - |L_S| + |R_S|$ .

# Halls Theorem

## Proof:

- ⇐ Of course, the condition is necessary as otherwise not all nodes in  $S$  could be matched to different neighbours.
- ⇒ For the other direction we need to argue that the minimum cut in the graph  $G'$  is at least  $|L|$ .
  - ▶ Let  $S$  denote a minimum cut and let  $L_S \stackrel{\text{def}}{=} L \cap S$  and  $R_S \stackrel{\text{def}}{=} R \cap S$  denote the portion of  $S$  inside  $L$  and  $R$ , respectively.
  - ▶ Clearly, all neighbours of nodes in  $L_S$  have to be in  $S$ , as otherwise we would cut an edge of infinite capacity.
  - ▶ This gives  $R_S \geq |\Gamma(L_S)|$ .
  - ▶ The size of the cut is  $|L| - |L_S| + |R_S|$ .
  - ▶ Using the fact that  $|\Gamma(L_S)| \geq |L_S|$  gives that this is at least  $|L|$ .

# Algorithm Outline

## Idea:

We introduce a node weighting  $\vec{x}$ . Let for a node  $v \in V$ ,  $x_v \in \mathbb{R}$  denote the weight of node  $v$ .

# Algorithm Outline

## Idea:

We introduce a node weighting  $\vec{x}$ . Let for a node  $v \in V$ ,  $x_v \in \mathbb{R}$  denote the weight of node  $v$ .

- ▶ Suppose that the node weights dominate the edge-weights in the following sense:

$$x_u + x_v \geq w_e \text{ for every edge } e = (u, v).$$

# Algorithm Outline

## Idea:

We introduce a node weighting  $\vec{x}$ . Let for a node  $v \in V$ ,  $x_v \in \mathbb{R}$  denote the weight of node  $v$ .

- ▶ Suppose that the node weights dominate the edge-weights in the following sense:

$$x_u + x_v \geq w_e \text{ for every edge } e = (u, v).$$

- ▶ Let  $H(\vec{x})$  denote the subgraph of  $G$  that only contains edges that are **tight** w.r.t. the node weighting  $\vec{x}$ , i.e. edges  $e = (u, v)$  for which  $w_e = x_u + x_v$ .



# Algorithm Outline

## Idea:

We introduce a node weighting  $\vec{x}$ . Let for a node  $v \in V$ ,  $x_v \in \mathbb{R}$  denote the weight of node  $v$ .

- ▶ Suppose that the node weights dominate the edge-weights in the following sense:

$$x_u + x_v \geq w_e \text{ for every edge } e = (u, v).$$

- ▶ Let  $H(\vec{x})$  denote the subgraph of  $G$  that only contains edges that are **tight** w.r.t. the node weighting  $\vec{x}$ , i.e. edges  $e = (u, v)$  for which  $w_e = x_u + x_v$ .
- ▶ Try to compute a perfect matching in the subgraph  $H(\vec{x})$ . If you are successful you found an optimal matching.

# Algorithm Outline

## Reason:

- ▶ The weight of your matching  $M^*$  is

$$\sum_{(u,v) \in M^*} w(u,v) = \sum_{(u,v) \in M^*} (x_u + x_v) = \sum_v x_v .$$

- ▶ Any other perfect matching  $M$  (in  $G$ , not necessarily in  $H(\vec{x})$ ) has

$$\sum_{(u,v) \in M} w(u,v) \leq \sum_{(u,v) \in M} (x_u + x_v) = \sum_v x_v .$$

# Algorithm Outline

## What if you don't find a perfect matching?

Then, Hall's theorem guarantees you that there is a set  $S \subseteq L$ , with  $|\Gamma(S)| < |S|$ , where  $\Gamma$  denotes the neighbourhood w.r.t. the subgraph  $H(\vec{x})$ .

# Algorithm Outline

## What if you don't find a perfect matching?

Then, Hall's theorem guarantees you that there is a set  $S \subseteq L$ , with  $|\Gamma(S)| < |S|$ , where  $\Gamma$  denotes the neighbourhood w.r.t. the subgraph  $H(\vec{x})$ .

**Idea:** reweight such that:

- ▶ the total weight assigned to nodes decreases
- ▶ the weight function still dominates the edge-weights

# Algorithm Outline

## What if you don't find a perfect matching?

Then, Hall's theorem guarantees you that there is a set  $S \subseteq L$ , with  $|\Gamma(S)| < |S|$ , where  $\Gamma$  denotes the neighbourhood w.r.t. the subgraph  $H(\vec{x})$ .

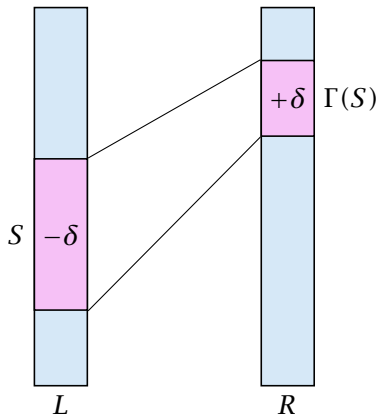
**Idea:** reweight such that:

- ▶ the total weight assigned to nodes decreases
- ▶ the weight function still dominates the edge-weights

If we can do this we have an algorithm that terminates with an optimal solution (we analyze the running time later).

## Changing Node Weights

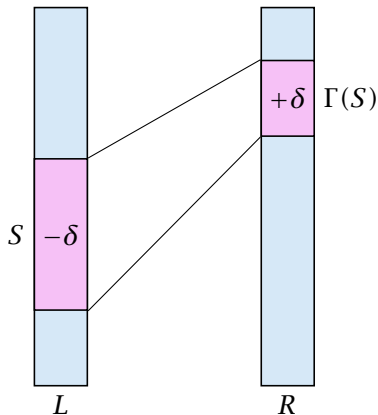
Increase node-weights in  $\Gamma(S)$  by  $+\delta$ , and decrease the node-weights in  $S$  by  $-\delta$ .



# Changing Node Weights

Increase node-weights in  $\Gamma(S)$  by  $+\delta$ , and decrease the node-weights in  $S$  by  $-\delta$ .

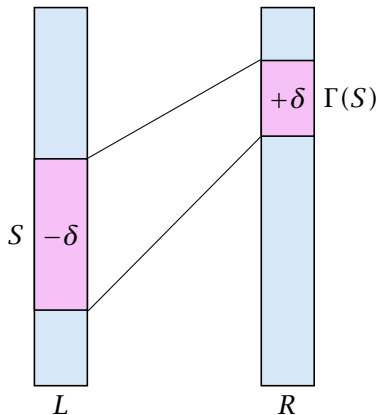
- ▶ Total node-weight decreases.



# Changing Node Weights

Increase node-weights in  $\Gamma(S)$  by  $+\delta$ , and decrease the node-weights in  $S$  by  $-\delta$ .

- ▶ Total node-weight decreases.
- ▶ Only edges from  $S$  to  $R - \Gamma(S)$  decrease in their weight.

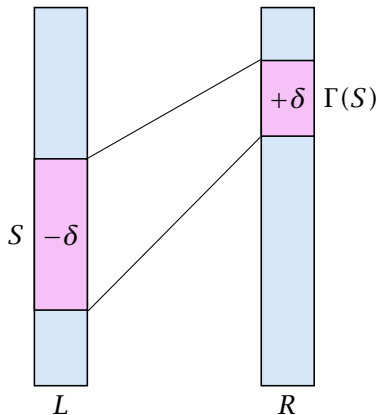




# Changing Node Weights

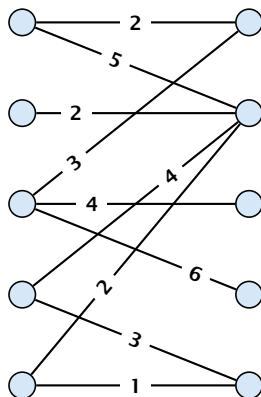
Increase node-weights in  $\Gamma(S)$  by  $+\delta$ , and decrease the node-weights in  $S$  by  $-\delta$ .

- ▶ Total node-weight decreases.
- ▶ Only edges from  $S$  to  $R - \Gamma(S)$  decrease in their weight.
- ▶ Since, none of these edges is tight (otw. the edge would be contained in  $H(\vec{x})$ , and hence would go between  $S$  and  $\Gamma(S)$ ) we can do this decrement for small enough  $\delta > 0$  until a new edge gets tight.



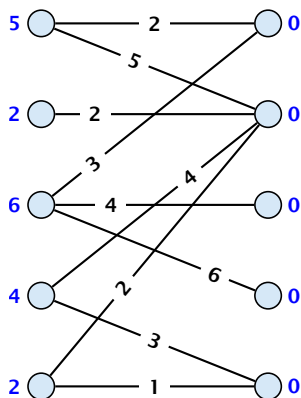
# Weighted Bipartite Matching

Edges not drawn have weight 0.



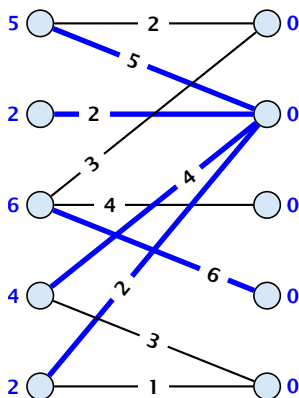
# Weighted Bipartite Matching

Edges not drawn have weight 0.



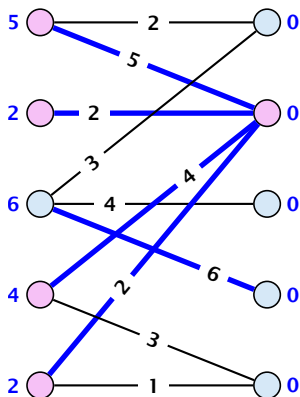
# Weighted Bipartite Matching

Edges not drawn have weight 0.



# Weighted Bipartite Matching

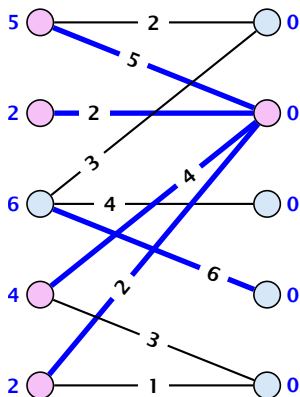
Edges not drawn have weight 0.



# Weighted Bipartite Matching

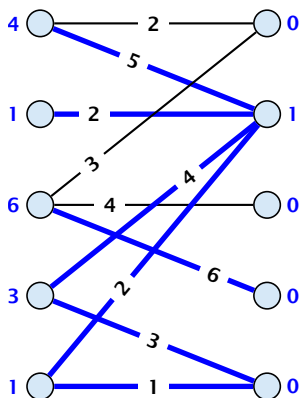
Edges not drawn have weight 0.

$$\delta = 1$$



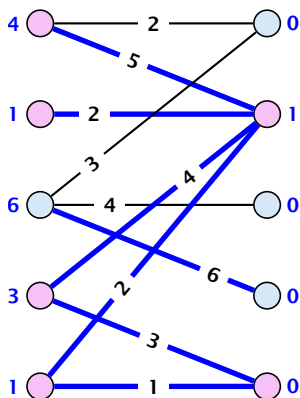
# Weighted Bipartite Matching

Edges not drawn have weight 0.



# Weighted Bipartite Matching

Edges not drawn have weight 0.

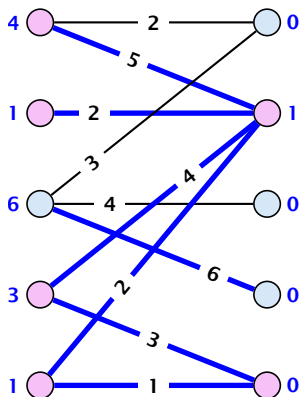




# Weighted Bipartite Matching

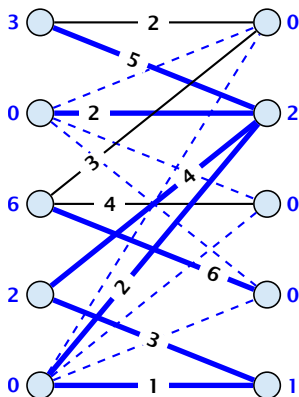
Edges not drawn have weight 0.

$$\delta = 1$$



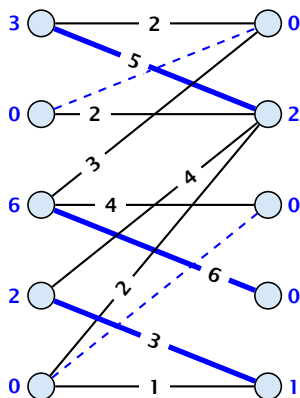
# Weighted Bipartite Matching

Edges not drawn have weight 0.



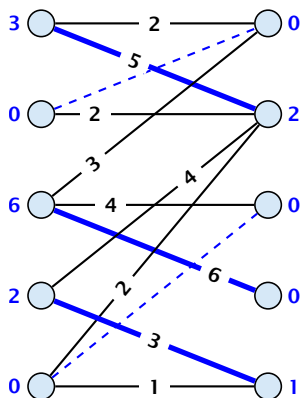
# Weighted Bipartite Matching

Edges not drawn have weight 0.



# Weighted Bipartite Matching

Edges not drawn have weight 0.



# Analysis

## How many iterations do we need?

- ▶ One reweighting step increases the number of edges out of  $S$  by at least one.

# Analysis

## How many iterations do we need?

- ▶ One reweighting step increases the number of edges out of  $S$  by at least one.
- ▶ Assume that we have a maximum matching that saturates the set  $\Gamma(S)$ , in the sense that every node in  $\Gamma(S)$  is matched to a node in  $S$  (we will show that we can always find  $S$  and a matching such that this holds).

## How many iterations do we need?

- ▶ One reweighting step increases the number of edges out of  $S$  by at least one.
- ▶ Assume that we have a maximum matching that saturates the set  $\Gamma(S)$ , in the sense that every node in  $\Gamma(S)$  is matched to a node in  $S$  (we will show that we can always find  $S$  and a matching such that this holds).
- ▶ This matching is still contained in the new graph, because all its edges either go between  $\Gamma(S)$  and  $S$  or between  $L - S$  and  $R - \Gamma(S)$ .

## How many iterations do we need?

- ▶ One reweighting step increases the number of edges out of  $S$  by at least one.
- ▶ Assume that we have a maximum matching that saturates the set  $\Gamma(S)$ , in the sense that every node in  $\Gamma(S)$  is matched to a node in  $S$  (we will show that we can always find  $S$  and a matching such that this holds).
- ▶ This matching is still contained in the new graph, because all its edges either go between  $\Gamma(S)$  and  $S$  or between  $L - S$  and  $R - \Gamma(S)$ .
- ▶ Hence, reweighting does not decrease the size of a maximum matching in the tight sub-graph.

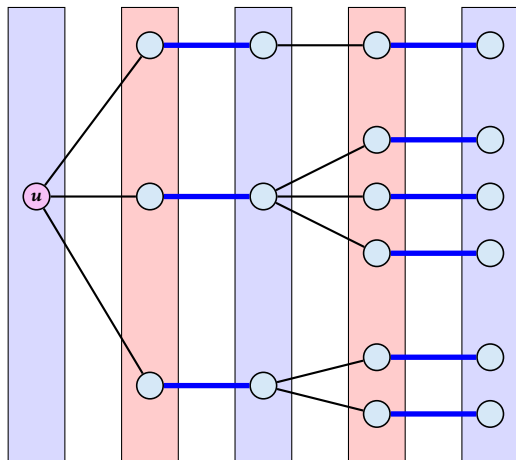


# Analysis

- ▶ We will show that after at most  $n$  reweighting steps the size of the maximum matching can be increased by finding an augmenting path.
- ▶ This gives a polynomial running time.

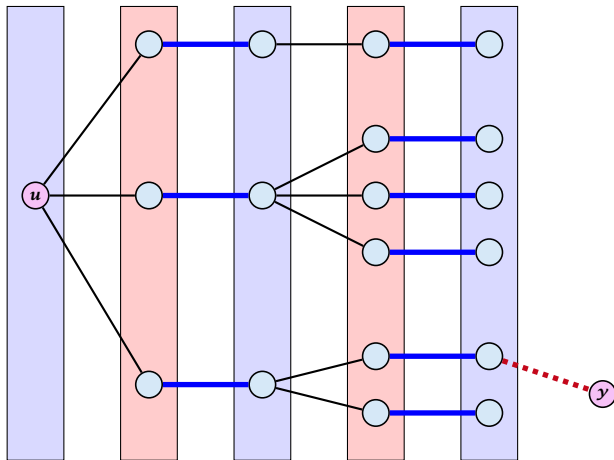
# How to find an augmenting path?

Construct an alternating tree.



# How to find an augmenting path?

Construct an alternating tree.



# Analysis

## How do we find $S$ ?

- ▶ Start on the left and compute an alternating tree, starting at any free node  $u$ .

# Analysis

## How do we find $S$ ?

- ▶ Start on the left and compute an alternating tree, starting at any free node  $u$ .
- ▶ If this construction stops, there is no perfect matching in the tight subgraph (because for a perfect matching we need to find an augmenting path starting at  $u$ ).

# Analysis

## How do we find $S$ ?

- ▶ Start on the left and compute an alternating tree, starting at any free node  $u$ .
- ▶ If this construction stops, there is no perfect matching in the tight subgraph (because for a perfect matching we need to find an augmenting path starting at  $u$ ).
- ▶ The set of even vertices is on the left and the set of odd vertices is on the right **and** contains all neighbours of even nodes.

# Analysis

## How do we find $S$ ?

- ▶ Start on the left and compute an alternating tree, starting at any free node  $u$ .
- ▶ If this construction stops, there is no perfect matching in the tight subgraph (because for a perfect matching we need to find an augmenting path starting at  $u$ ).
- ▶ The set of even vertices is on the left and the set of odd vertices is on the right **and** contains all neighbours of even nodes.
- ▶ All odd vertices are matched to even vertices. Furthermore, the even vertices additionally contain the free vertex  $u$ . Hence,  $|V_{\text{odd}}| = |\Gamma(V_{\text{even}})| < |V_{\text{even}}|$ , and all odd vertices are saturated in the current matching.

# Analysis

- ▶ The current matching does not have any edges from  $V_{\text{odd}}$  to  $L \setminus V_{\text{even}}$  (edges that may possibly be deleted by changing weights).



# Analysis

- ▶ The current matching does not have any edges from  $V_{\text{odd}}$  to  $L \setminus V_{\text{even}}$  (edges that may possibly be deleted by changing weights).
- ▶ After changing weights, there is at least one more edge connecting  $V_{\text{even}}$  to a node outside of  $V_{\text{odd}}$ . After at most  $n$  reweights we can do an augmentation.

# Analysis

- ▶ The current matching does not have any edges from  $V_{\text{odd}}$  to  $L \setminus V_{\text{even}}$  (edges that may possibly be deleted by changing weights).
- ▶ After changing weights, there is at least one more edge connecting  $V_{\text{even}}$  to a node outside of  $V_{\text{odd}}$ . After at most  $n$  reweightings we can do an augmentation.
- ▶ A reweighting can be trivially performed in time  $\mathcal{O}(n^2)$  (keeping track of the tight edges).

# Analysis

- ▶ The current matching does not have any edges from  $V_{\text{odd}}$  to  $L \setminus V_{\text{even}}$  (edges that may possibly be deleted by changing weights).
- ▶ After changing weights, there is at least one more edge connecting  $V_{\text{even}}$  to a node outside of  $V_{\text{odd}}$ . After at most  $n$  reweightings we can do an augmentation.
- ▶ A reweighting can be trivially performed in time  $\mathcal{O}(n^2)$  (keeping track of the tight edges).
- ▶ An augmentation takes at most  $\mathcal{O}(n)$  time.

# Analysis

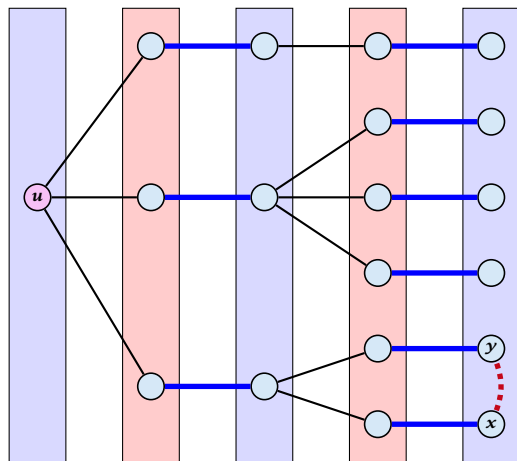
- ▶ The current matching does not have any edges from  $V_{\text{odd}}$  to  $L \setminus V_{\text{even}}$  (edges that may possibly be deleted by changing weights).
- ▶ After changing weights, there is at least one more edge connecting  $V_{\text{even}}$  to a node outside of  $V_{\text{odd}}$ . After at most  $n$  reweights we can do an augmentation.
- ▶ A reweighting can be trivially performed in time  $\mathcal{O}(n^2)$  (keeping track of the tight edges).
- ▶ An augmentation takes at most  $\mathcal{O}(n)$  time.
- ▶ In total we obtain a running time of  $\mathcal{O}(n^4)$ .

# Analysis

- ▶ The current matching does not have any edges from  $V_{\text{odd}}$  to  $L \setminus V_{\text{even}}$  (edges that may possibly be deleted by changing weights).
- ▶ After changing weights, there is at least one more edge connecting  $V_{\text{even}}$  to a node outside of  $V_{\text{odd}}$ . After at most  $n$  reweights we can do an augmentation.
- ▶ A reweighting can be trivially performed in time  $\mathcal{O}(n^2)$  (keeping track of the tight edges).
- ▶ An augmentation takes at most  $\mathcal{O}(n)$  time.
- ▶ In total we obtain a running time of  $\mathcal{O}(n^4)$ .
- ▶ A more careful implementation of the algorithm obtains a running time of  $\mathcal{O}(n^3)$ .

# How to find an augmenting path?

Construct an alternating tree.



even nodes

odd nodes

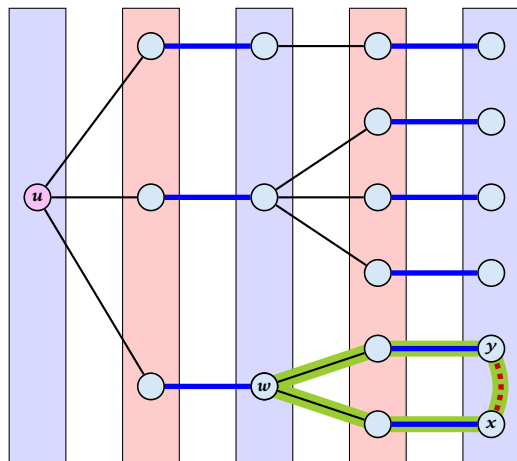
**Case 4:**

$y$  is already contained  
in  $T$  as an even vertex

can't ignore  $y$

# How to find an augmenting path?

Construct an alternating tree.



even nodes

odd nodes

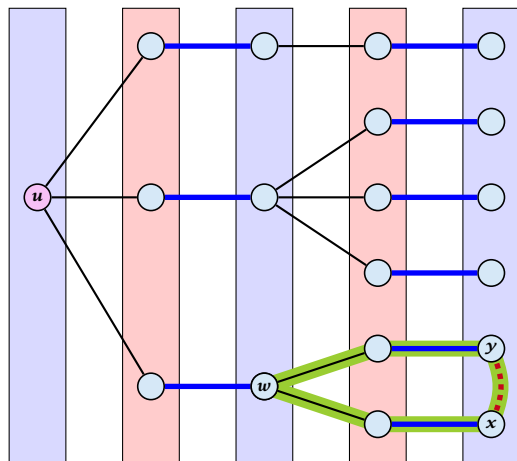
**Case 4:**

$y$  is already contained  
in  $T$  as an even vertex

can't ignore  $y$

# How to find an augmenting path?

Construct an alternating tree.



even nodes

odd nodes

**Case 4:**

$y$  is already contained  
in  $T$  as an even vertex

**can't ignore  $y$**

The cycle  $w \leftrightarrow y - x \leftrightarrow w$   
is called a **blossom**.  
 $w$  is called the **base** of the  
blossom (even node!!!).  
The path  $u-w$  is called the  
**stem** of the blossom.



# Flowers and Blossoms

## Definition 92

A **flower** in a graph  $G = (V, E)$  w.r.t. a matching  $M$  and a (free) root node  $r$ , is a subgraph with two components:

# Flowers and Blossoms

## Definition 92

A **flower** in a graph  $G = (V, E)$  w.r.t. a matching  $M$  and a (free) root node  $r$ , is a subgraph with two components:

- ▶ A **stem** is an even length alternating path that starts at the root node  $r$  and terminates at some node  $w$ . We permit the possibility that  $r = w$  (empty stem).

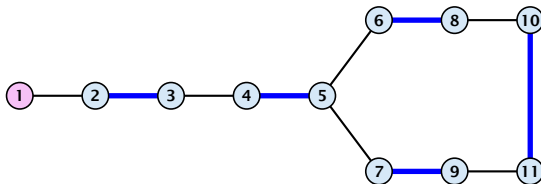
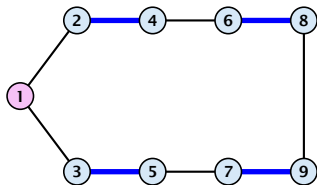
# Flowers and Blossoms

## Definition 92

A **flower** in a graph  $G = (V, E)$  w.r.t. a matching  $M$  and a (free) root node  $r$ , is a subgraph with two components:

- ▶ A **stem** is an even length alternating path that starts at the root node  $r$  and terminates at some node  $w$ . We permit the possibility that  $r = w$  (empty stem).
- ▶ A **blossom** is an odd length alternating cycle that starts and terminates at the terminal node  $w$  of a stem and has no other node in common with the stem.  $w$  is called the **base** of the blossom.

# Flowers and Blossoms



# Flowers and Blossoms

## Properties:

1. A stem spans  $2\ell + 1$  nodes and contains  $\ell$  matched edges for some integer  $\ell \geq 0$ .

# Flowers and Blossoms

## Properties:

1. A stem spans  $2\ell + 1$  nodes and contains  $\ell$  matched edges for some integer  $\ell \geq 0$ .
2. A blossom spans  $2k + 1$  nodes and contains  $k$  matched edges for some integer  $k \geq 1$ . The matched edges match all nodes of the blossom except the base.

# Flowers and Blossoms

## Properties:

1. A stem spans  $2\ell + 1$  nodes and contains  $\ell$  matched edges for some integer  $\ell \geq 0$ .
2. A blossom spans  $2k + 1$  nodes and contains  $k$  matched edges for some integer  $k \geq 1$ . The matched edges match all nodes of the blossom except the base.
3. The base of a blossom is an even node (if the stem is part of an alternating tree starting at  $r$ ).

# Flowers and Blossoms

## Properties:

4. Every node  $x$  in the blossom (except its base) is reachable from the root (or from the base of the blossom) through two distinct alternating paths; one with even and one with odd length.

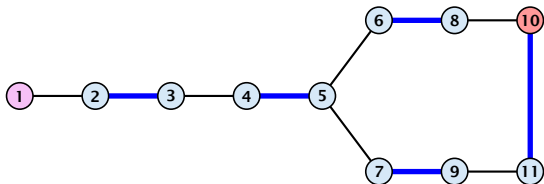


# Flowers and Blossoms

## Properties:

4. Every node  $x$  in the blossom (except its base) is reachable from the root (or from the base of the blossom) through two distinct alternating paths; one with even and one with odd length.
5. The even alternating path to  $x$  terminates with a matched edge and the odd path with an unmatched edge.

# Flowers and Blossoms



# Shrinking Blossoms

When during the alternating tree construction we discover a blossom  $B$  we replace the graph  $G$  by  $G' = G/B$ , which is obtained from  $G$  by contracting the blossom  $B$ .

# Shrinking Blossoms

When during the alternating tree construction we discover a blossom  $B$  we replace the graph  $G$  by  $G' = G/B$ , which is obtained from  $G$  by contracting the blossom  $B$ .

- ▶ Delete all vertices in  $B$  (and its incident edges) from  $G$ .

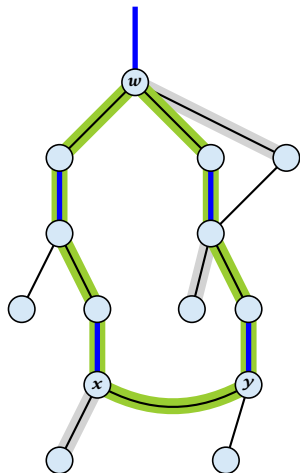
# Shrinking Blossoms

When during the alternating tree construction we discover a blossom  $B$  we replace the graph  $G$  by  $G' = G/B$ , which is obtained from  $G$  by contracting the blossom  $B$ .

- ▶ Delete all vertices in  $B$  (and its incident edges) from  $G$ .
- ▶ Add a new (pseudo-)vertex  $b$ . The new vertex  $b$  is connected to all vertices in  $V \setminus B$  that had at least one edge to a vertex from  $B$ .

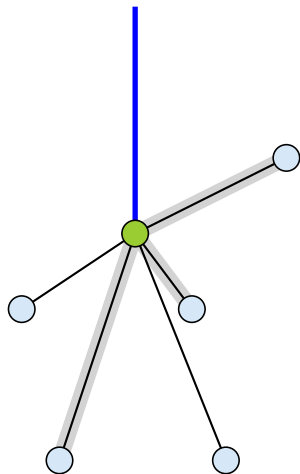
# Shrinking Blossoms

- ▶ Edges of  $T$  that connect a node  $u$  not in  $B$  to a node in  $B$  become tree edges in  $T'$  connecting  $u$  to  $b$ .
- ▶ Matching edges (there is at most one) that connect a node  $u$  not in  $B$  to a node in  $B$  become matching edges in  $M'$ .
- ▶ Nodes that are connected in  $G$  to at least one node in  $B$  become connected to  $b$  in  $G'$ .

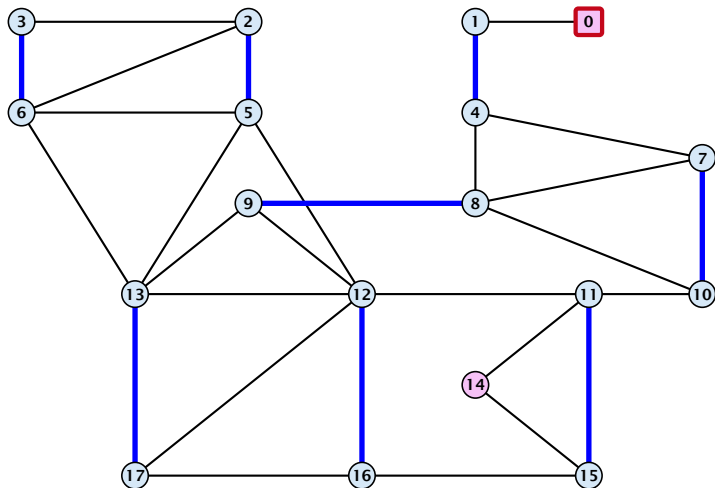


# Shrinking Blossoms

- ▶ Edges of  $T$  that connect a node  $u$  not in  $B$  to a node in  $B$  become tree edges in  $T'$  connecting  $u$  to  $b$ .
- ▶ Matching edges (there is at most one) that connect a node  $u$  not in  $B$  to a node in  $B$  become matching edges in  $M'$ .
- ▶ Nodes that are connected in  $G$  to at least one node in  $B$  become connected to  $b$  in  $G'$ .

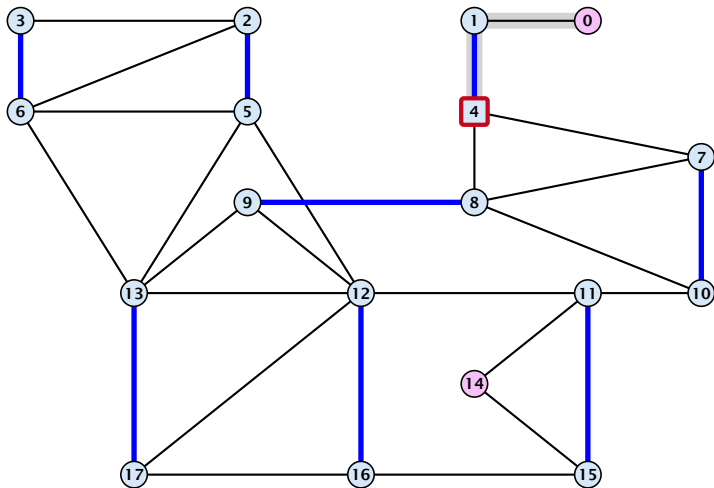


# Example: Blossom Algorithm

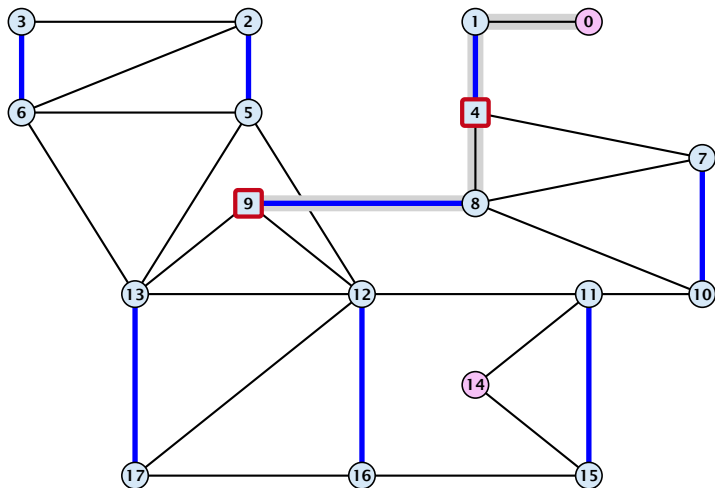




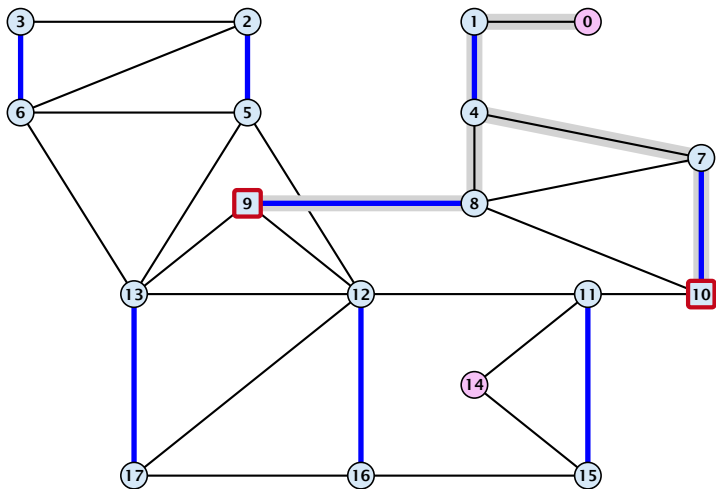
# Example: Blossom Algorithm



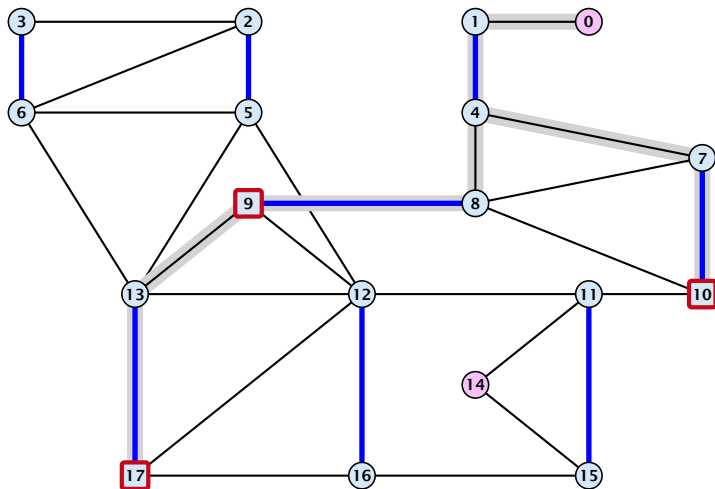
# Example: Blossom Algorithm



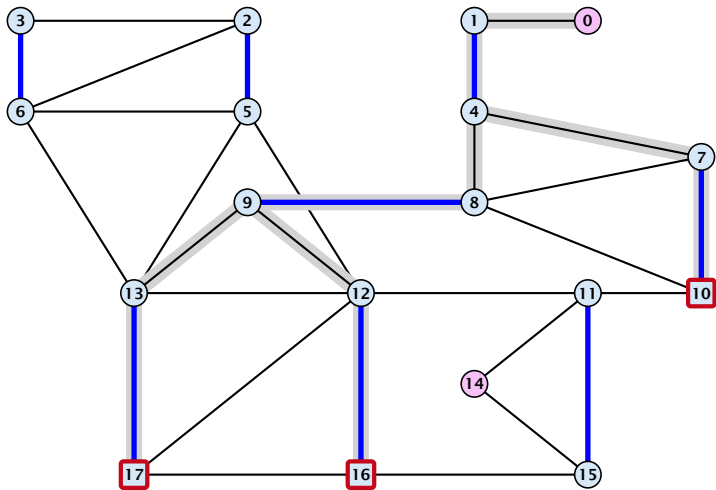
# Example: Blossom Algorithm



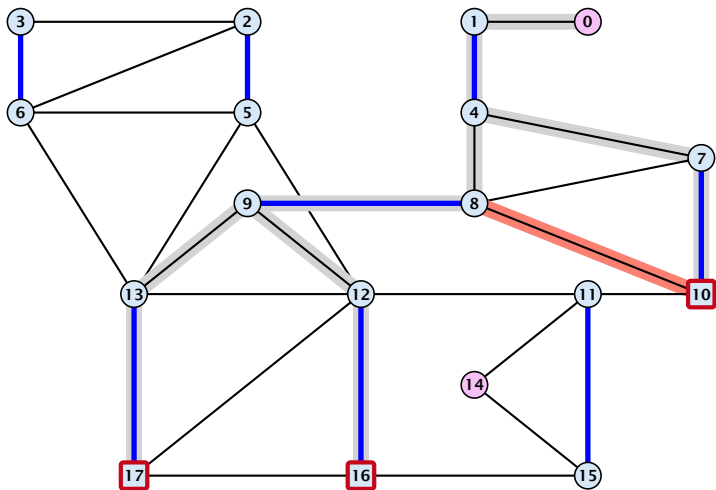
# Example: Blossom Algorithm



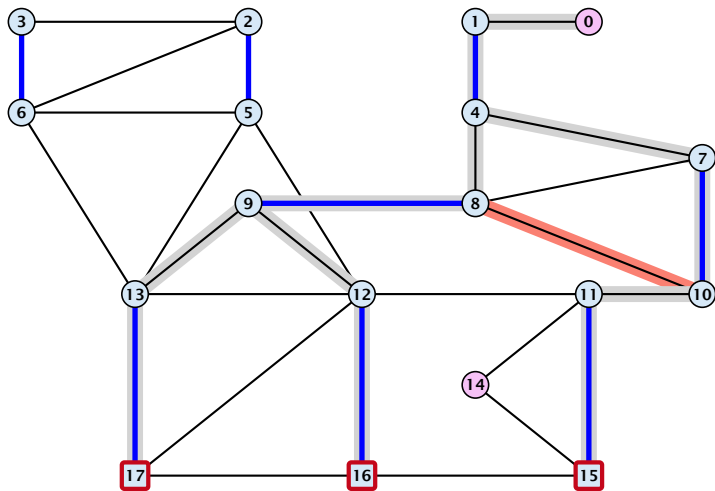
# Example: Blossom Algorithm



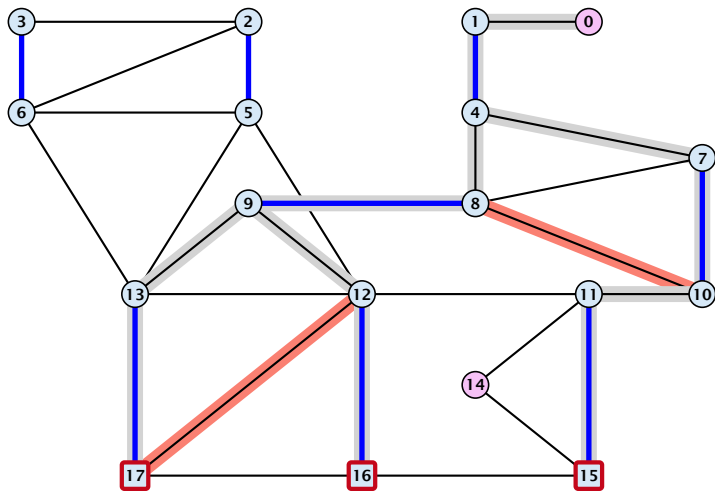
# Example: Blossom Algorithm



# Example: Blossom Algorithm

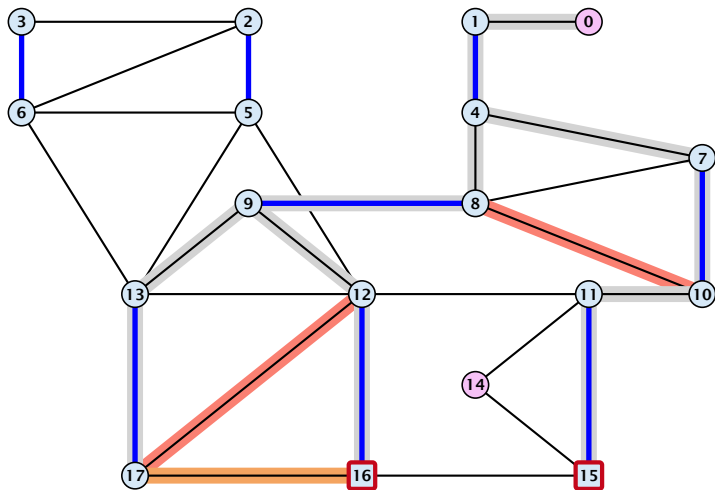


# Example: Blossom Algorithm

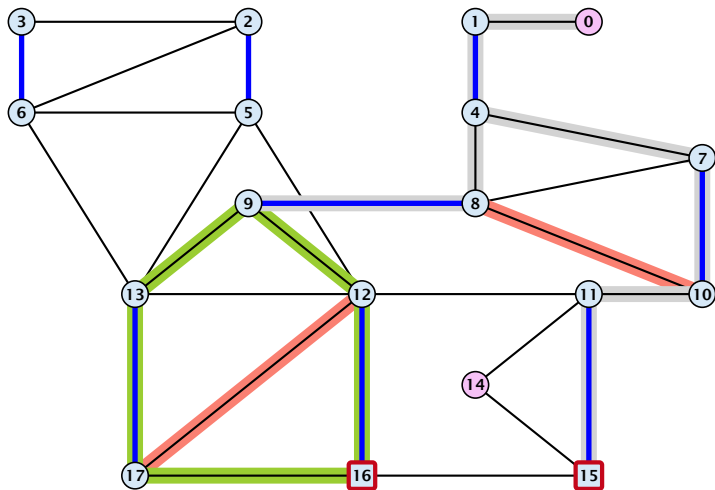




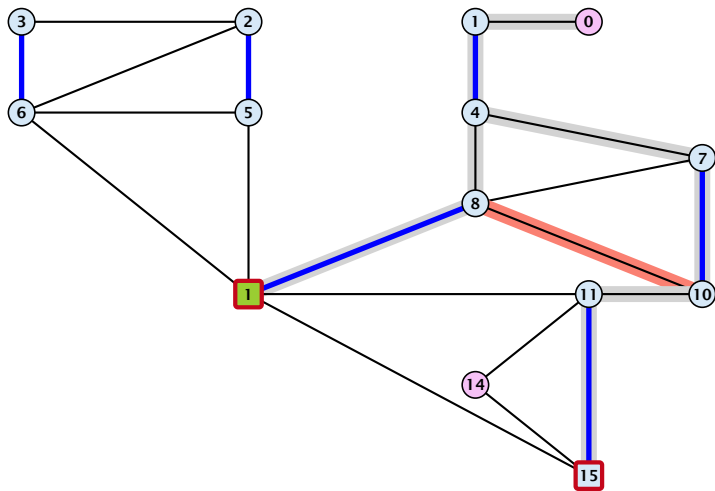
# Example: Blossom Algorithm



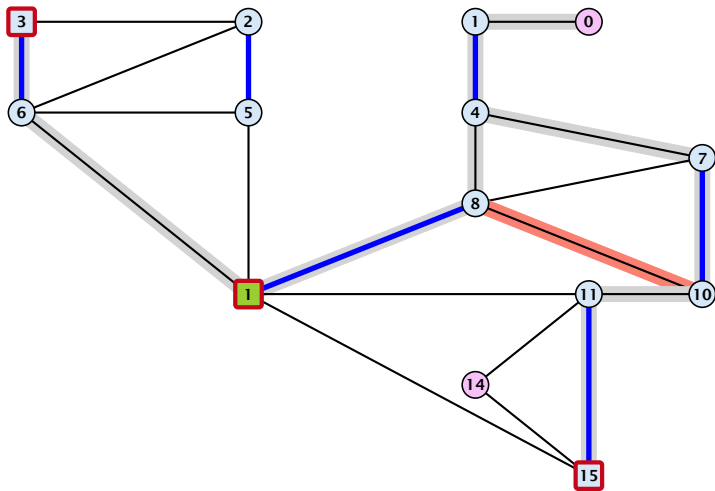
# Example: Blossom Algorithm



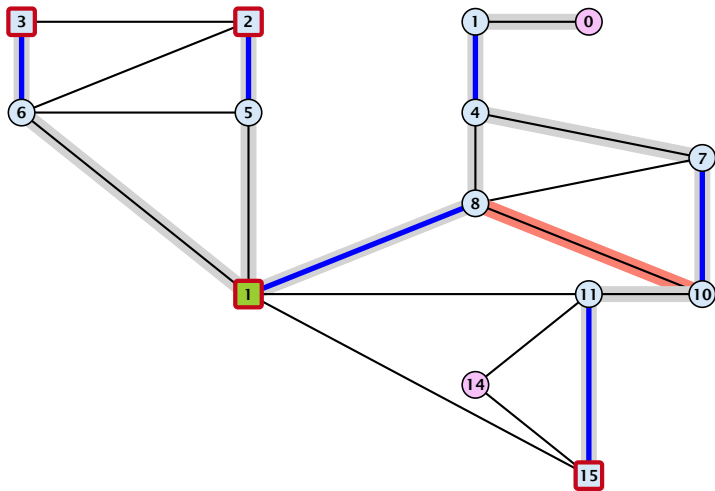
# Example: Blossom Algorithm



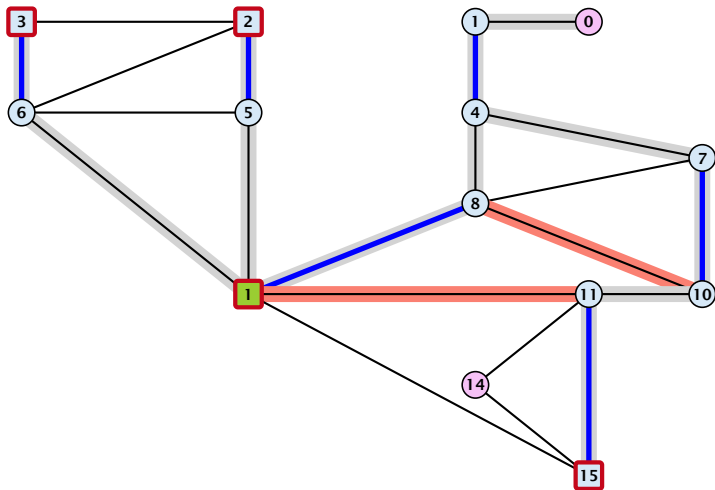
# Example: Blossom Algorithm



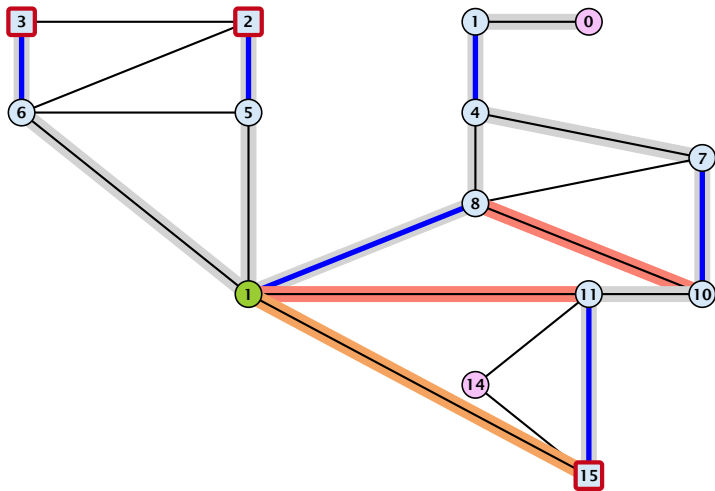
# Example: Blossom Algorithm



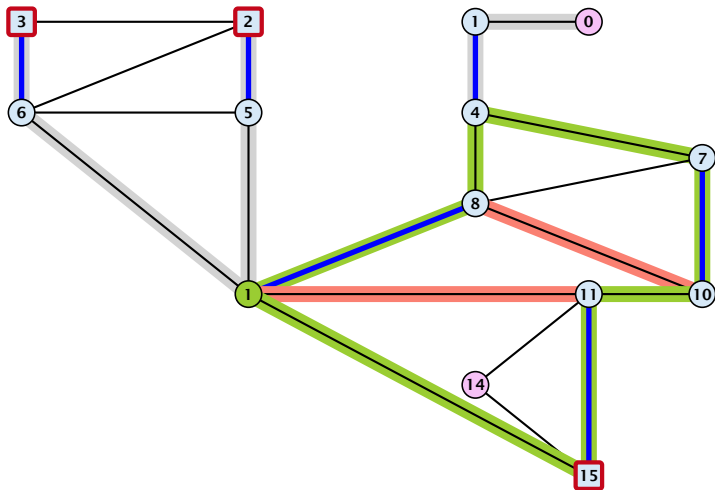
# Example: Blossom Algorithm



# Example: Blossom Algorithm

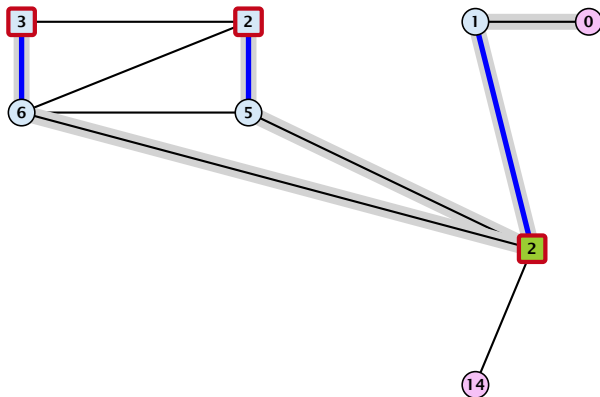


# Example: Blossom Algorithm

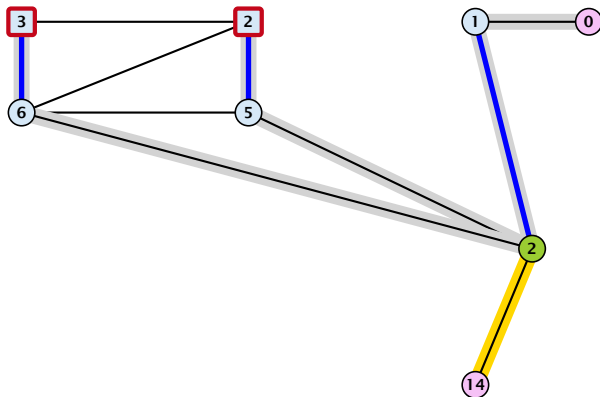




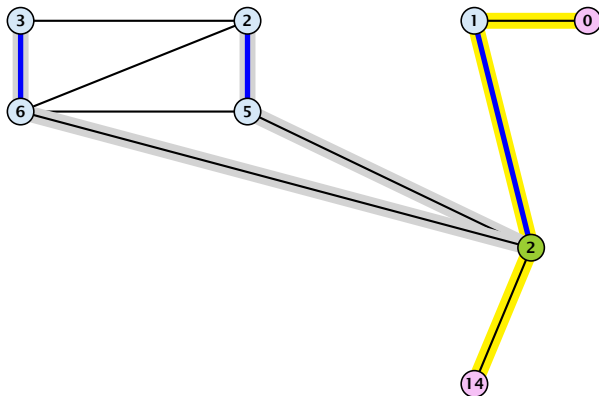
# Example: Blossom Algorithm



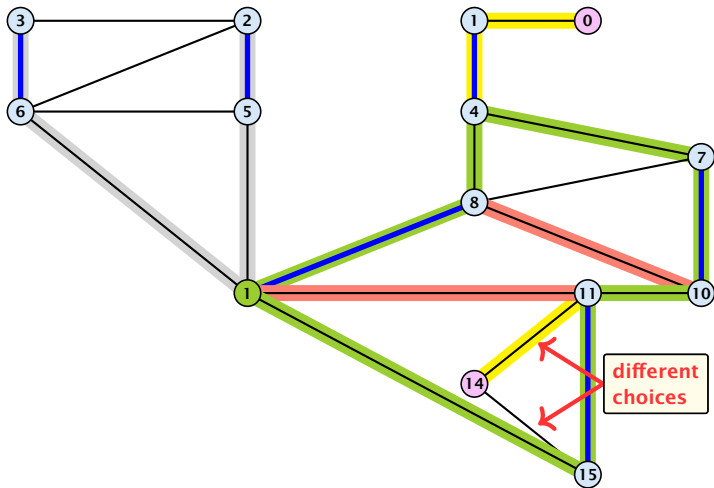
# Example: Blossom Algorithm



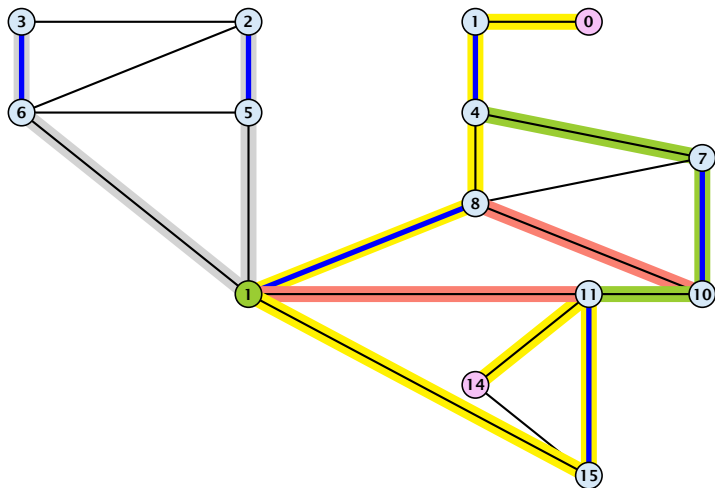
# Example: Blossom Algorithm



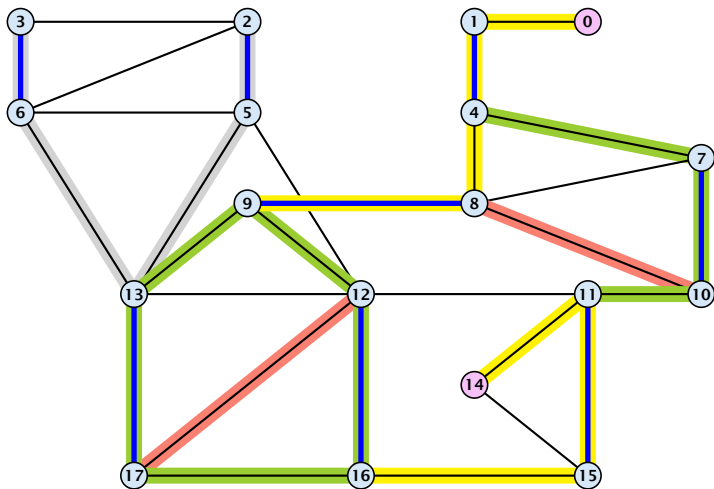
# Example: Blossom Algorithm



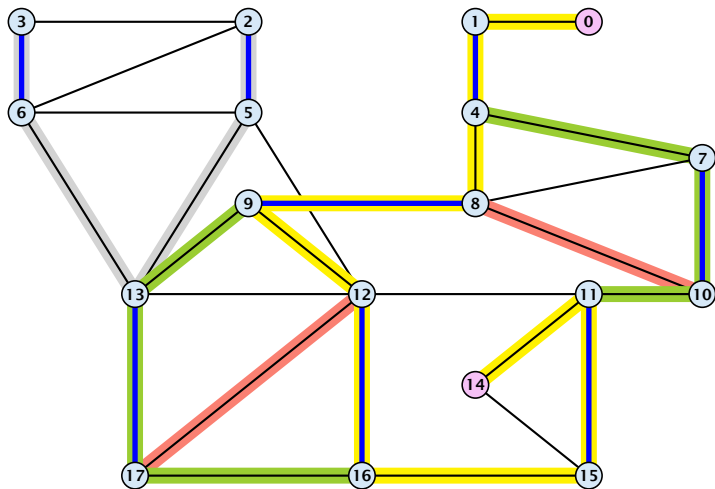
# Example: Blossom Algorithm



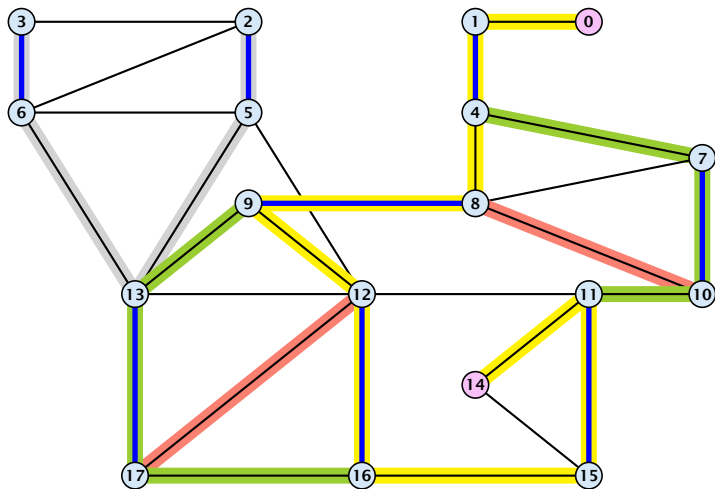
# Example: Blossom Algorithm



# Example: Blossom Algorithm



# Example: Blossom Algorithm





# Correctness

Assume that in  $G$  we have a flower w.r.t. matching  $M$ . Let  $r$  be the root,  $B$  the blossom, and  $w$  the base. Let graph  $G' = G/B$  with pseudonode  $b$ . Let  $M'$  be the matching in the contracted graph.

Assume that in  $G$  we have a flower w.r.t. matching  $M$ . Let  $r$  be the root,  $B$  the blossom, and  $w$  the base. Let graph  $G' = G/B$  with pseudonode  $b$ . Let  $M'$  be the matching in the contracted graph.

## Lemma 93

*If  $G'$  contains an augmenting path  $P'$  starting at  $r$  (or the pseudo-node containing  $r$ ) w.r.t. the matching  $M'$  then  $G$  contains an augmenting path starting at  $r$  w.r.t. matching  $M$ .*

# Correctness

**Proof.**

If  $P'$  does not contain  $b$  it is also an augmenting path in  $G$ .

# Correctness

## Proof.

If  $P'$  does not contain  $b$  it is also an augmenting path in  $G$ .

## Case 1: non-empty stem

- ▶ Next suppose that the stem is non-empty.

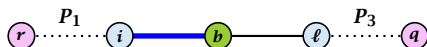
# Correctness

**Proof.**

If  $P'$  does not contain  $b$  it is also an augmenting path in  $G$ .

**Case 1: non-empty stem**

- ▶ Next suppose that the stem is non-empty.



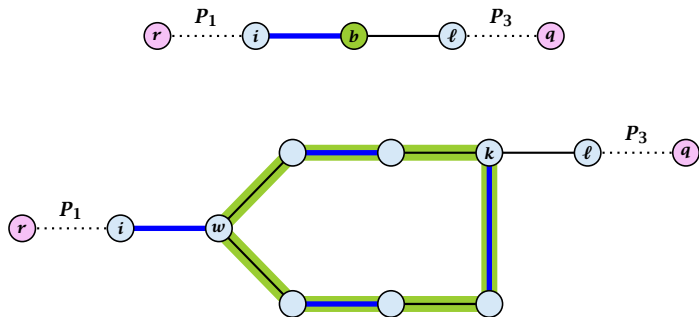
# Correctness

## Proof.

If  $P'$  does not contain  $b$  it is also an augmenting path in  $G$ .

### Case 1: non-empty stem

- ▶ Next suppose that the stem is non-empty.



# Correctness

- ▶ After the expansion  $\ell$  must be incident to some node in the blossom. Let this node be  $k$ .
- ▶ If  $k \neq w$  there is an alternating path  $P_2$  from  $w$  to  $k$  that ends in a matching edge.
- ▶  $P_1 \circ (i, w) \circ P_2 \circ (k, \ell) \circ P_3$  is an alternating path.
- ▶ If  $k = w$  then  $P_1 \circ (i, w) \circ (w, \ell) \circ P_3$  is an alternating path.

# Correctness

**Proof.**

**Case 2: empty stem**

- ▶ If the stem is empty then after expanding the blossom,  
 $w = r$ .



# Correctness

**Proof.**

**Case 2: empty stem**

- ▶ If the stem is empty then after expanding the blossom,  $w = r$ .

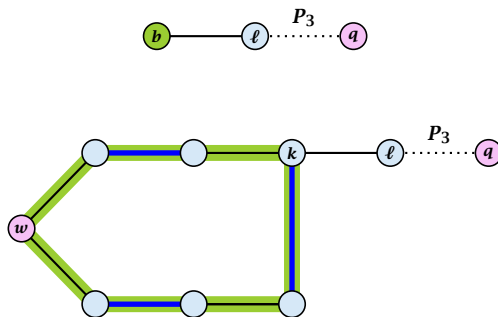


# Correctness

**Proof.**

**Case 2: empty stem**

- ▶ If the stem is empty then after expanding the blossom,  $w = r$ .

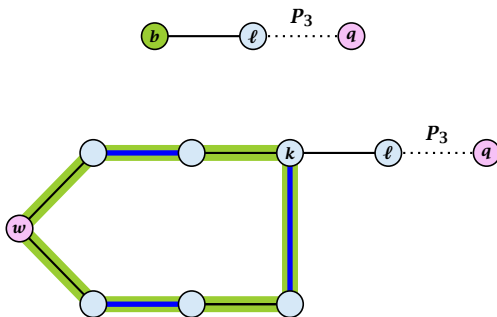


# Correctness

**Proof.**

**Case 2: empty stem**

- ▶ If the stem is empty then after expanding the blossom,  $w = r$ .



- ▶ The path  $r \circ P_2 \circ (k, l) \circ P_3$  is an alternating path.

## Lemma 94

*If  $G$  contains an augmenting path  $P$  from  $r$  to  $q$  w.r.t. matching  $M$  then  $G'$  contains an augmenting path from  $r$  (or the pseudo-node containing  $r$ ) to  $q$  w.r.t.  $M'$ .*

# Correctness

## Proof.

- ▶ If  $P$  does not contain a node from  $B$  there is nothing to prove.

# Correctness

## Proof.

- ▶ If  $P$  does not contain a node from  $B$  there is nothing to prove.
- ▶ We can assume that  $r$  and  $q$  are the only free nodes in  $G$ .

# Correctness

## Proof.

- ▶ If  $P$  does not contain a node from  $B$  there is nothing to prove.
- ▶ We can assume that  $r$  and  $q$  are the only free nodes in  $G$ .

## Case 1: empty stem

# Correctness

## Proof.

- ▶ If  $P$  does not contain a node from  $B$  there is nothing to prove.
- ▶ We can assume that  $r$  and  $q$  are the only free nodes in  $G$ .

## Case 1: empty stem

Let  $i$  be the last node on the path  $P$  that is part of the blossom.



# Correctness

## Proof.

- ▶ If  $P$  does not contain a node from  $B$  there is nothing to prove.
- ▶ We can assume that  $r$  and  $q$  are the only free nodes in  $G$ .

## Case 1: empty stem

Let  $i$  be the last node on the path  $P$  that is part of the blossom.

$P$  is of the form  $P_1 \circ (i, j) \circ P_2$ , for some node  $j$  and  $(i, j)$  is unmatched.

# Correctness

## Proof.

- ▶ If  $P$  does not contain a node from  $B$  there is nothing to prove.
- ▶ We can assume that  $r$  and  $q$  are the only free nodes in  $G$ .

## Case 1: empty stem

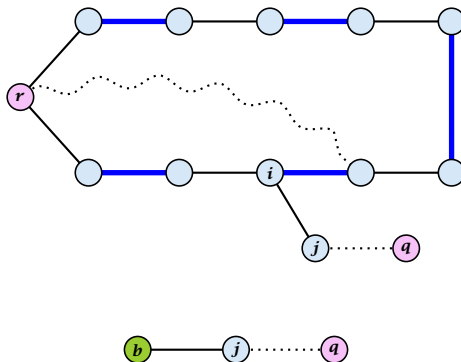
Let  $i$  be the last node on the path  $P$  that is part of the blossom.

$P$  is of the form  $P_1 \circ (i, j) \circ P_2$ , for some node  $j$  and  $(i, j)$  is unmatched.

$(b, j) \circ P_2$  is an augmenting path in the contracted network.

# Correctness

## Illustration for Case 1:



# Correctness

**Case 2: non-empty stem**

## Correctness

### Case 2: non-empty stem

Let  $P_3$  be alternating path from  $r$  to  $w$ ; this exists because  $r$  and  $w$  are root and base of a blossom. Define  $M_+ = M \oplus P_3$ .

## Correctness

### Case 2: non-empty stem

Let  $P_3$  be alternating path from  $r$  to  $w$ ; this exists because  $r$  and  $w$  are root and base of a blossom. Define  $M_+ = M \oplus P_3$ .

In  $M_+$ ,  $r$  is matched and  $w$  is unmatched.

## Correctness

### Case 2: non-empty stem

Let  $P_3$  be alternating path from  $r$  to  $w$ ; this exists because  $r$  and  $w$  are root and base of a blossom. Define  $M_+ = M \oplus P_3$ .

In  $M_+$ ,  $r$  is matched and  $w$  is unmatched.

$G$  must contain an augmenting path w.r.t. matching  $M_+$ , since  $M$  and  $M_+$  have same cardinality.

## Correctness

### Case 2: non-empty stem

Let  $P_3$  be alternating path from  $r$  to  $w$ ; this exists because  $r$  and  $w$  are root and base of a blossom. Define  $M_+ = M \oplus P_3$ .

In  $M_+$ ,  $r$  is matched and  $w$  is unmatched.

$G$  must contain an augmenting path w.r.t. matching  $M_+$ , since  $M$  and  $M_+$  have same cardinality.

This path must go between  $w$  and  $q$  as these are the only unmatched vertices w.r.t.  $M_+$ .



## Correctness

### Case 2: non-empty stem

Let  $P_3$  be alternating path from  $r$  to  $w$ ; this exists because  $r$  and  $w$  are root and base of a blossom. Define  $M_+ = M \oplus P_3$ .

In  $M_+$ ,  $r$  is matched and  $w$  is unmatched.

$G$  must contain an augmenting path w.r.t. matching  $M_+$ , since  $M$  and  $M_+$  have same cardinality.

This path must go between  $w$  and  $q$  as these are the only unmatched vertices w.r.t.  $M_+$ .

For  $M'_+$  the blossom has an empty stem. Case 1 applies.

## Correctness

### Case 2: non-empty stem

Let  $P_3$  be alternating path from  $r$  to  $w$ ; this exists because  $r$  and  $w$  are root and base of a blossom. Define  $M_+ = M \oplus P_3$ .

In  $M_+$ ,  $r$  is matched and  $w$  is unmatched.

$G$  must contain an augmenting path w.r.t. matching  $M_+$ , since  $M$  and  $M_+$  have same cardinality.

This path must go between  $w$  and  $q$  as these are the only unmatched vertices w.r.t.  $M_+$ .

For  $M'_+$  the blossom has an empty stem. Case 1 applies.

$G'$  has an augmenting path w.r.t.  $M'_+$ . It must also have an augmenting path w.r.t.  $M'$ , as both matchings have the same cardinality.

## Correctness

### Case 2: non-empty stem

Let  $P_3$  be alternating path from  $r$  to  $w$ ; this exists because  $r$  and  $w$  are root and base of a blossom. Define  $M_+ = M \oplus P_3$ .

In  $M_+$ ,  $r$  is matched and  $w$  is unmatched.

$G$  must contain an augmenting path w.r.t. matching  $M_+$ , since  $M$  and  $M_+$  have same cardinality.

This path must go between  $w$  and  $q$  as these are the only unmatched vertices w.r.t.  $M_+$ .

For  $M'_+$  the blossom has an empty stem. Case 1 applies.

$G'$  has an augmenting path w.r.t.  $M'_+$ . It must also have an augmenting path w.r.t.  $M'$ , as both matchings have the same cardinality.

This path must go between  $r$  and  $q$ .

**Algorithm 50**  $\text{search}(r, \text{found})$

- 1: set  $\bar{A}(i) \leftarrow A(i)$  for all nodes  $i$
- 2:  $\text{found} \leftarrow \text{false}$
- 3: unlabel all nodes;
- 4: give an even label to  $r$  and initialize  $\text{list} \leftarrow \{r\}$
- 5: **while**  $\text{list} \neq \emptyset$  **do**
- 6:     delete a node  $i$  from  $\text{list}$
- 7:     examine( $i, \text{found}$ )
- 8:     **if**  $\text{found} = \text{true}$  **then return**

Search for an augmenting path  
starting at  $r$ .

**Algorithm 50**  $\text{search}(r, \text{found})$

- 1: set  $\bar{A}(i) \leftarrow A(i)$  for all nodes  $i$
- 2:  $\text{found} \leftarrow \text{false}$
- 3: unlabel all nodes;
- 4: give an even label to  $r$  and initialize  $\text{list} \leftarrow \{r\}$
- 5: **while**  $\text{list} \neq \emptyset$  **do**
- 6:     delete a node  $i$  from  $\text{list}$
- 7:     examine( $i, \text{found}$ )
- 8:     **if**  $\text{found} = \text{true}$  **then return**

$A(i)$  contains neighbours of node  $i$ .

We create a copy  $\bar{A}(i)$  so that we later  
can shrink blossoms.

**Algorithm 50**  $\text{search}(r, \text{found})$

1: set  $\bar{A}(i) \leftarrow A(i)$  for all nodes  $i$

2:  $\text{found} \leftarrow \text{false}$

3: unlabel all nodes;

4: give an even label to  $r$  and initialize  $\text{list} \leftarrow \{r\}$

5: **while**  $\text{list} \neq \emptyset$  **do**

6:     delete a node  $i$  from  $\text{list}$

7:     examine( $i, \text{found}$ )

8:     **if**  $\text{found} = \text{true}$  **then return**

*found* is just a Boolean that allows  
to abort the search process...

**Algorithm 50**  $\text{search}(r, \text{found})$

1: set  $\bar{A}(i) \leftarrow A(i)$  for all nodes  $i$

2:  $\text{found} \leftarrow \text{false}$

3: **unlabel all nodes;**

4: give an even label to  $r$  and initialize  $\text{list} \leftarrow \{r\}$

5: **while**  $\text{list} \neq \emptyset$  **do**

6:     delete a node  $i$  from  $\text{list}$

7:      $\text{examine}(i, \text{found})$

8:     **if**  $\text{found} = \text{true}$  **then return**

In the beginning no node is in the tree.

**Algorithm 50** search( $r$ ,  $found$ )

1: set  $\bar{A}(i) \leftarrow A(i)$  for all nodes  $i$

2:  $found \leftarrow \text{false}$

3: unlabel all nodes;

4: give an even label to  $r$  and initialize  $list \leftarrow \{r\}$

5: **while**  $list \neq \emptyset$  **do**

6:     delete a node  $i$  from  $list$

7:     examine( $i$ ,  $found$ )

8:     **if**  $found = \text{true}$  **then return**

Put the root in the tree.

*list* could also be a set or a stack.



**Algorithm 50**  $\text{search}(r, \text{found})$

1: set  $\bar{A}(i) \leftarrow A(i)$  for all nodes  $i$

2:  $\text{found} \leftarrow \text{false}$

3: unlabel all nodes;

4: give an even label to  $r$  and initialize  $\text{list} \leftarrow \{r\}$

5: **while**  $\text{list} \neq \emptyset$  **do**

6:     delete a node  $i$  from  $\text{list}$

7:     examine( $i, \text{found}$ )

8:     **if**  $\text{found} = \text{true}$  **then return**

As long as there are nodes with  
unexamined neighbours...

**Algorithm 50**  $\text{search}(r, \text{found})$

- 1: set  $\bar{A}(i) \leftarrow A(i)$  for all nodes  $i$
- 2:  $\text{found} \leftarrow \text{false}$
- 3: unlabel all nodes;
- 4: give an even label to  $r$  and initialize  $\text{list} \leftarrow \{r\}$
- 5: **while**  $\text{list} \neq \emptyset$  **do**
- 6:     delete a node  $i$  from  $\text{list}$
- 7:     examine( $i, \text{found}$ )
- 8:     **if**  $\text{found} = \text{true}$  **then return**

...examine the next one

**Algorithm 50**  $\text{search}(r, \text{found})$

- 1: set  $\bar{A}(i) \leftarrow A(i)$  for all nodes  $i$
- 2:  $\text{found} \leftarrow \text{false}$
- 3: unlabel all nodes;
- 4: give an even label to  $r$  and initialize  $\text{list} \leftarrow \{r\}$
- 5: **while**  $\text{list} \neq \emptyset$  **do**
- 6:     delete a node  $i$  from  $\text{list}$
- 7:     examine( $i, \text{found}$ )
- 8:     **if**  $\text{found} = \text{true}$  **then return**

If you found augmenting path  
abort and start from next root.

**Algorithm 51** examine( $i$ ,  $found$ )

```
1: for all  $j \in \bar{A}(i)$  do  
2:   if  $j$  is even then contract( $i$ ,  $j$ ) and return  
3:   if  $j$  is unmatched then  
4:      $q \leftarrow j$ ;  
5:     pred( $q$ )  $\leftarrow i$ ;  
6:      $found \leftarrow \text{true}$ ;  
7:     return  
8:   if  $j$  is matched and unlabeled then  
9:     pred( $j$ )  $\leftarrow i$ ;  
10:    pred(mate( $j$ ))  $\leftarrow j$ ;  
11:    add mate( $j$ ) to  $list$ 
```

Examine the neighbours of a node  $i$

**Algorithm 51** examine( $i, found$ )

```
1: for all  $j \in \bar{A}(i)$  do
2:   if  $j$  is even then contract( $i, j$ ) and return
3:   if  $j$  is unmatched then
4:      $q \leftarrow j$ ;
5:     pred( $q$ )  $\leftarrow i$ ;
6:      $found \leftarrow \text{true}$ ;
7:     return
8:   if  $j$  is matched and unlabeled then
9:     pred( $j$ )  $\leftarrow i$ ;
10:    pred(mate( $j$ ))  $\leftarrow j$ ;
11:    add mate( $j$ ) to  $list$ 
```

For all neighbours  $j$  do...

**Algorithm 51** examine( $i, found$ )

```
1: for all  $j \in \bar{A}(i)$  do
2:   if  $j$  is even then contract( $i, j$ ) and return
3:   if  $j$  is unmatched then
4:      $q \leftarrow j$ ;
5:     pred( $q$ )  $\leftarrow i$ ;
6:      $found \leftarrow \text{true}$ ;
7:     return
8:   if  $j$  is matched and unlabeled then
9:     pred( $j$ )  $\leftarrow i$ ;
10:    pred(mate( $j$ ))  $\leftarrow j$ ;
11:    add mate( $j$ ) to  $list$ 
```

You have found a blossom...

**Algorithm 51**  $\text{examine}(i, \text{found})$ 

```
1: for all  $j \in \bar{A}(i)$  do  
2:   if  $j$  is even then  $\text{contract}(i, j)$  and return  
3:   if  $j$  is unmatched then  
4:      $q \leftarrow j$ ;  
5:      $\text{pred}(q) \leftarrow i$ ;  
6:      $\text{found} \leftarrow \text{true}$ ;  
7:     return  
8:   if  $j$  is matched and unlabeled then  
9:      $\text{pred}(j) \leftarrow i$ ;  
10:     $\text{pred}(\text{mate}(j)) \leftarrow j$ ;  
11:    add  $\text{mate}(j)$  to  $\text{list}$ 
```

You have found a free node which gives you an augmenting path.

**Algorithm 51**  $\text{examine}(i, \text{found})$ 

```
1: for all  $j \in \bar{A}(i)$  do  
2:   if  $j$  is even then  $\text{contract}(i, j)$  and return  
3:   if  $j$  is unmatched then  
4:      $q \leftarrow j$ ;  
5:      $\text{pred}(q) \leftarrow i$ ;  
6:      $\text{found} \leftarrow \text{true}$ ;  
7:     return  
8:   if  $j$  is matched and unlabeled then  
9:      $\text{pred}(j) \leftarrow i$ ;  
10:     $\text{pred}(\text{mate}(j)) \leftarrow j$ ;  
11:    add  $\text{mate}(j)$  to list
```

If you find a matched node that is not  
in the tree you grow...



**Algorithm 51** examine( $i, found$ )

```
1: for all  $j \in \bar{A}(i)$  do
2:   if  $j$  is even then contract( $i, j$ ) and return
3:   if  $j$  is unmatched then
4:      $q \leftarrow j$ ;
5:     pred( $q$ )  $\leftarrow i$ ;
6:      $found \leftarrow \text{true}$ ;
7:     return
8:   if  $j$  is matched and unlabeled then
9:     pred( $j$ )  $\leftarrow i$ ;
10:    pred(mate( $j$ ))  $\leftarrow j$ ;
11:    add mate( $j$ ) to  $list$ 
```

$mate(j)$  is a new node from  
which you can grow further.

### Algorithm 52 contract( $i, j$ )

- 1: trace pred-indices of  $i$  and  $j$  to identify a blossom  $B$
- 2: create new node  $b$  and set  $\bar{A}(b) \leftarrow \cup_{x \in B} \bar{A}(x)$
- 3: label  $b$  even and add to *list*
- 4: update  $\bar{A}(j) \leftarrow \bar{A}(j) \cup \{b\}$  for each  $j \in \bar{A}(b)$
- 5: form a circular double linked list of nodes in  $B$
- 6: delete nodes in  $B$  from the graph

Contract blossom identified by  
nodes  $i$  and  $j$

### Algorithm 52 contract( $i, j$ )

- 1: trace pred-indices of  $i$  and  $j$  to identify a blossom  $B$
- 2: create new node  $b$  and set  $\bar{A}(b) \leftarrow \cup_{x \in B} \bar{A}(x)$
- 3: label  $b$  even and add to *list*
- 4: update  $\bar{A}(j) \leftarrow \bar{A}(j) \cup \{b\}$  for each  $j \in \bar{A}(b)$
- 5: form a circular double linked list of nodes in  $B$
- 6: delete nodes in  $B$  from the graph

Get all nodes of the blossom.

Time:  $\mathcal{O}(m)$

### Algorithm 52 contract( $i, j$ )

- 1: trace pred-indices of  $i$  and  $j$  to identify a blossom  $B$
- 2: create new node  $b$  and set  $\bar{A}(b) \leftarrow \cup_{x \in B} \bar{A}(x)$
- 3: label  $b$  even and add to *list*
- 4: update  $\bar{A}(j) \leftarrow \bar{A}(j) \cup \{b\}$  for each  $j \in \bar{A}(b)$
- 5: form a circular double linked list of nodes in  $B$
- 6: delete nodes in  $B$  from the graph

Identify all neighbours of  $b$ .

Time:  $\mathcal{O}(m)$  (how?)

### Algorithm 52 contract( $i, j$ )

- 1: trace pred-indices of  $i$  and  $j$  to identify a blossom  $B$
- 2: create new node  $b$  and set  $\bar{A}(b) \leftarrow \cup_{x \in B} \bar{A}(x)$
- 3: label  $b$  even and add to *list*
- 4: update  $\bar{A}(j) \leftarrow \bar{A}(j) \cup \{b\}$  for each  $j \in \bar{A}(b)$
- 5: form a circular double linked list of nodes in  $B$
- 6: delete nodes in  $B$  from the graph

$b$  will be an even node, and it has unexamined neighbours.

### Algorithm 52 contract( $i, j$ )

- 1: trace pred-indices of  $i$  and  $j$  to identify a blossom  $B$
- 2: create new node  $b$  and set  $\bar{A}(b) \leftarrow \cup_{x \in B} \bar{A}(x)$
- 3: label  $b$  even and add to *list*
- 4: update  $\bar{A}(j) \leftarrow \bar{A}(j) \cup \{b\}$  for each  $j \in \bar{A}(b)$
- 5: form a circular double linked list of nodes in  $B$
- 6: delete nodes in  $B$  from the graph

Every node that was adjacent to a node  
in  $B$  is now adjacent to  $b$

### Algorithm 52 contract( $i, j$ )

- 1: trace pred-indices of  $i$  and  $j$  to identify a blossom  $B$
- 2: create new node  $b$  and set  $\bar{A}(b) \leftarrow \cup_{x \in B} \bar{A}(x)$
- 3: label  $b$  even and add to *list*
- 4: update  $\bar{A}(j) \leftarrow \bar{A}(j) \cup \{b\}$  for each  $j \in \bar{A}(b)$
- 5: form a circular double linked list of nodes in  $B$
- 6: delete nodes in  $B$  from the graph

Only for making a blossom expansion easier.

### Algorithm 52 contract( $i, j$ )

- 1: trace pred-indices of  $i$  and  $j$  to identify a blossom  $B$
- 2: create new node  $b$  and set  $\bar{A}(b) \leftarrow \cup_{x \in B} \bar{A}(x)$
- 3: label  $b$  even and add to *list*
- 4: update  $\bar{A}(j) \leftarrow \bar{A}(j) \cup \{b\}$  for each  $j \in \bar{A}(b)$
- 5: form a circular double linked list of nodes in  $B$
- 6: delete nodes in  $B$  from the graph

Only delete links from nodes not in  $B$  to  $B$ .

When expanding the blossom again we can recreate these links in time  $\mathcal{O}(m)$ .



## Analysis

- ▶ A contraction operation can be performed in time  $\mathcal{O}(m)$ .  
Note, that any graph created will have at most  $m$  edges.

## Analysis

- ▶ A contraction operation can be performed in time  $\mathcal{O}(m)$ . Note, that any graph created will have at most  $m$  edges.
- ▶ The time between two contraction-operation is basically a BFS/DFS on a graph. Hence takes time  $\mathcal{O}(m)$ .

# Analysis

- ▶ A contraction operation can be performed in time  $\mathcal{O}(m)$ . Note, that any graph created will have at most  $m$  edges.
- ▶ The time between two contraction-operation is basically a BFS/DFS on a graph. Hence takes time  $\mathcal{O}(m)$ .
- ▶ There are at most  $n$  contractions as each contraction reduces the number of vertices.

# Analysis

- ▶ A contraction operation can be performed in time  $\mathcal{O}(m)$ . Note, that any graph created will have at most  $m$  edges.
- ▶ The time between two contraction-operation is basically a BFS/DFS on a graph. Hence takes time  $\mathcal{O}(m)$ .
- ▶ There are at most  $n$  contractions as each contraction reduces the number of vertices.
- ▶ The expansion can trivially be done in the same time as needed for all contractions.

# Analysis

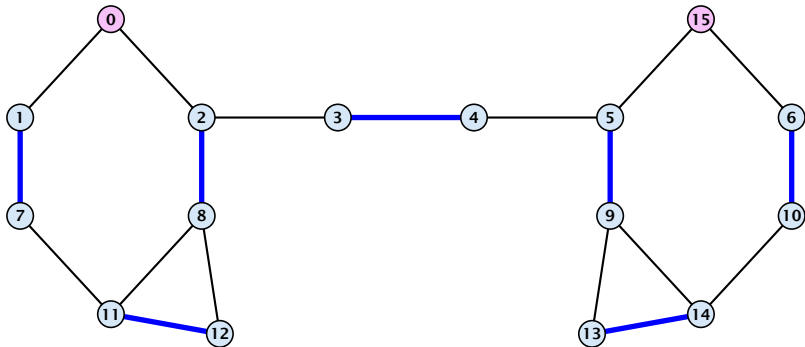
- ▶ A contraction operation can be performed in time  $\mathcal{O}(m)$ . Note, that any graph created will have at most  $m$  edges.
- ▶ The time between two contraction-operation is basically a BFS/DFS on a graph. Hence takes time  $\mathcal{O}(m)$ .
- ▶ There are at most  $n$  contractions as each contraction reduces the number of vertices.
- ▶ The expansion can trivially be done in the same time as needed for all contractions.
- ▶ An augmentation requires time  $\mathcal{O}(n)$ . There are at most  $n$  of them.

# Analysis

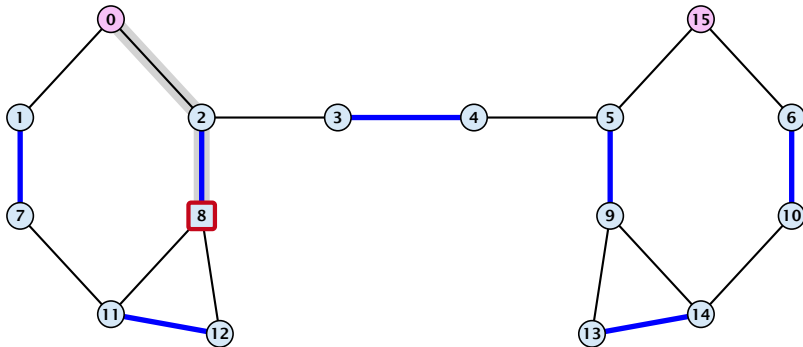
- ▶ A contraction operation can be performed in time  $\mathcal{O}(m)$ . Note, that any graph created will have at most  $m$  edges.
- ▶ The time between two contraction-operation is basically a BFS/DFS on a graph. Hence takes time  $\mathcal{O}(m)$ .
- ▶ There are at most  $n$  contractions as each contraction reduces the number of vertices.
- ▶ The expansion can trivially be done in the same time as needed for all contractions.
- ▶ An augmentation requires time  $\mathcal{O}(n)$ . There are at most  $n$  of them.
- ▶ In total the running time is at most

$$n \cdot (\mathcal{O}(mn) + \mathcal{O}(n)) = \mathcal{O}(mn^2) .$$

# Example: Blossom Algorithm

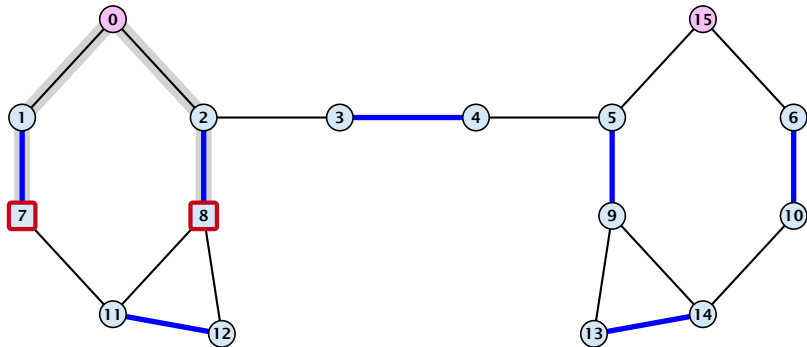


# Example: Blossom Algorithm

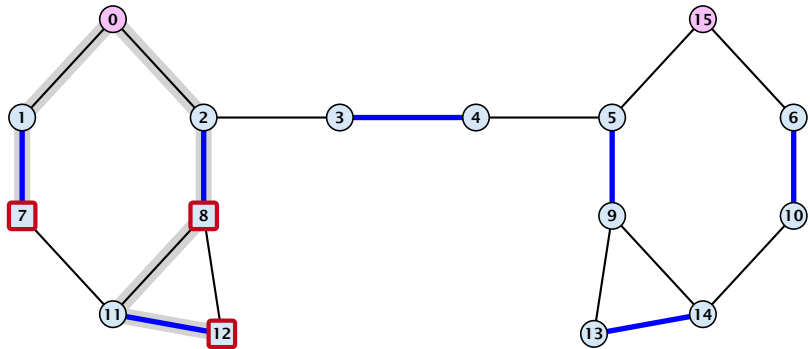




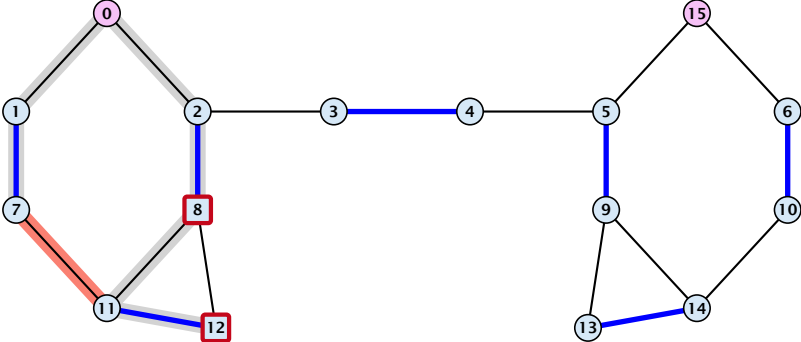
# Example: Blossom Algorithm



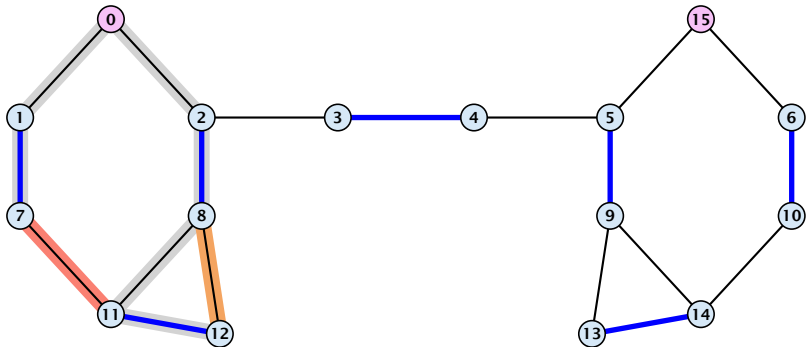
# Example: Blossom Algorithm



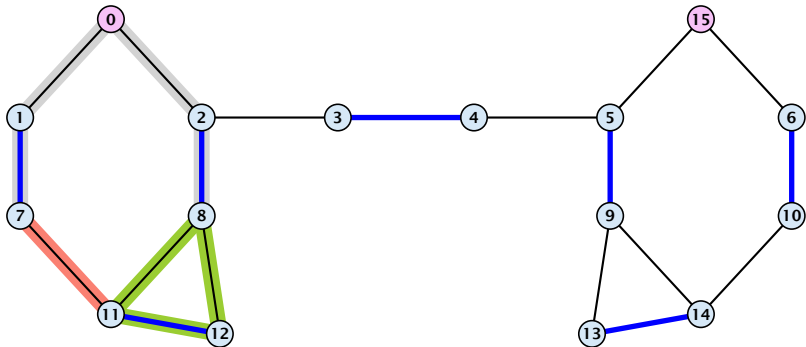
# Example: Blossom Algorithm



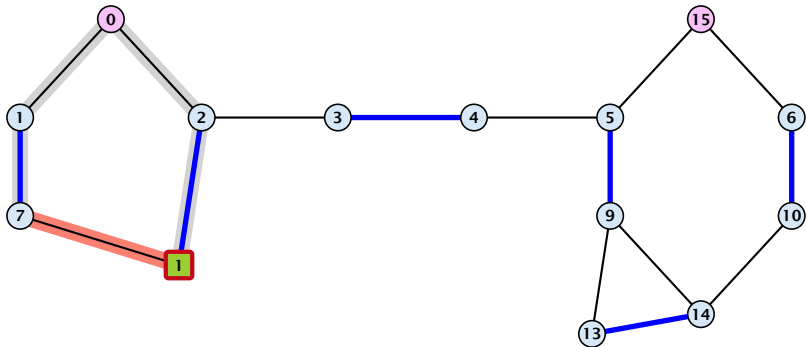
# Example: Blossom Algorithm



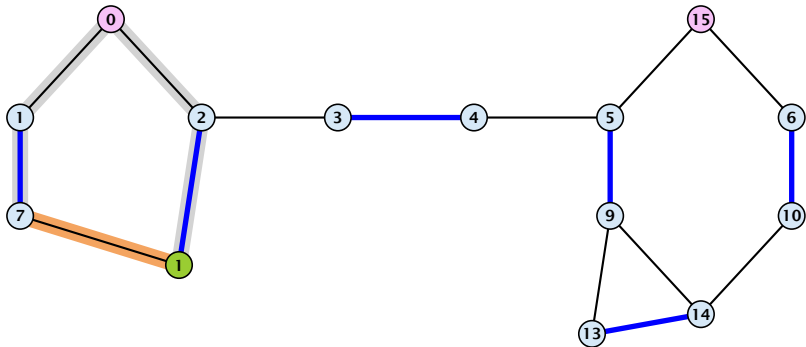
# Example: Blossom Algorithm



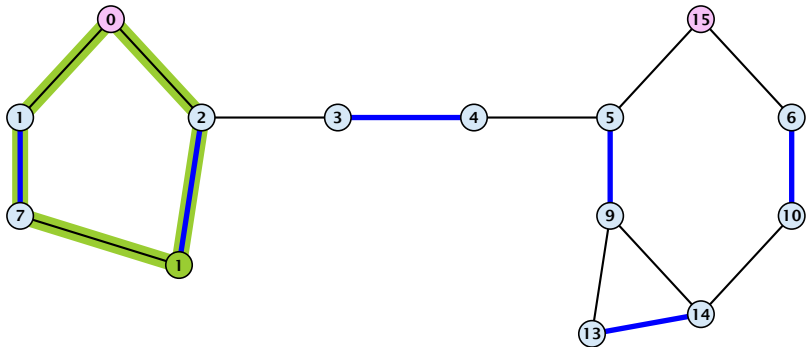
# Example: Blossom Algorithm



# Example: Blossom Algorithm

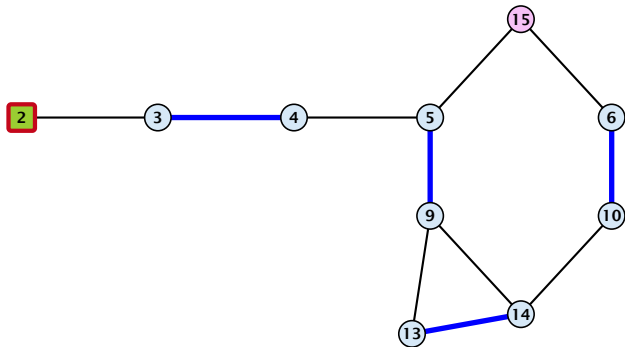


# Example: Blossom Algorithm

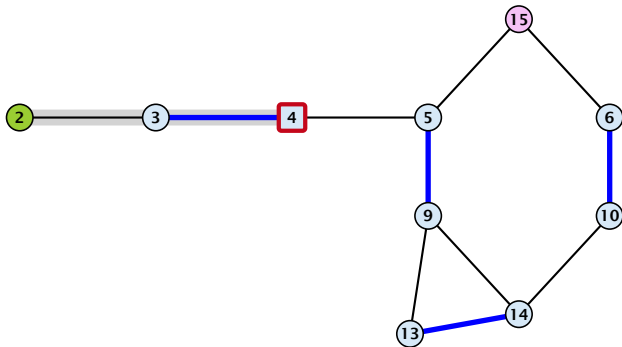




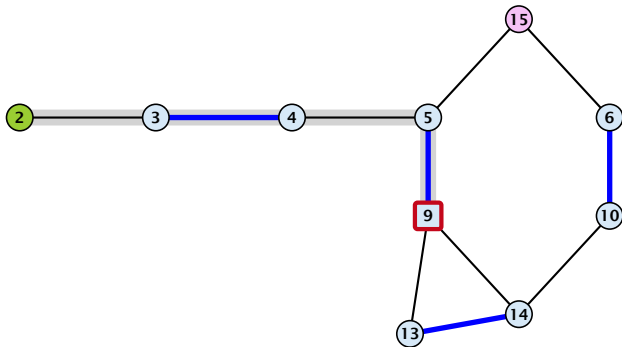
# Example: Blossom Algorithm



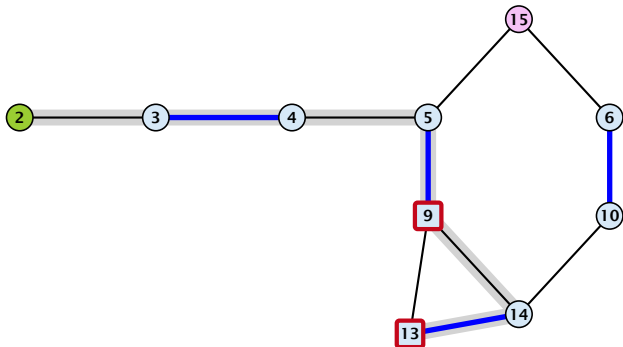
# Example: Blossom Algorithm



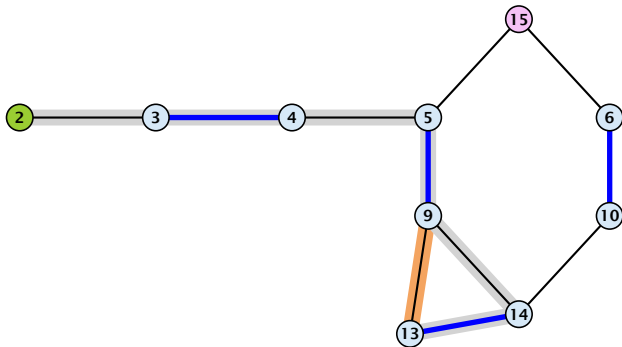
# Example: Blossom Algorithm



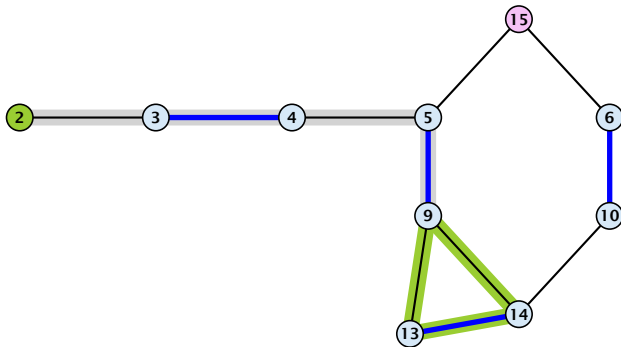
# Example: Blossom Algorithm



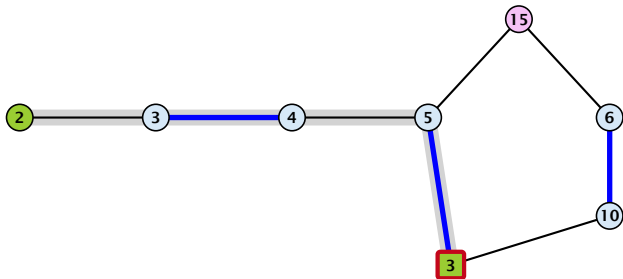
# Example: Blossom Algorithm



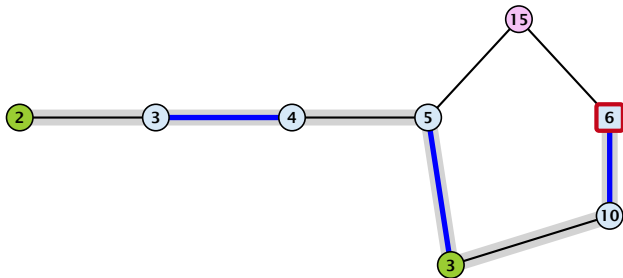
# Example: Blossom Algorithm



# Example: Blossom Algorithm

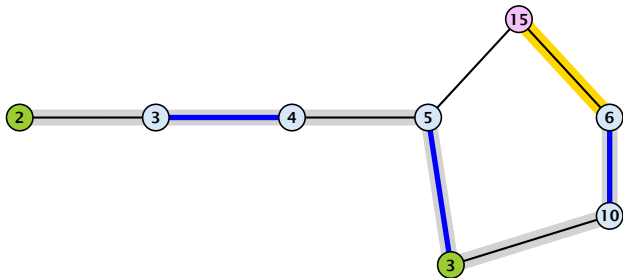


# Example: Blossom Algorithm

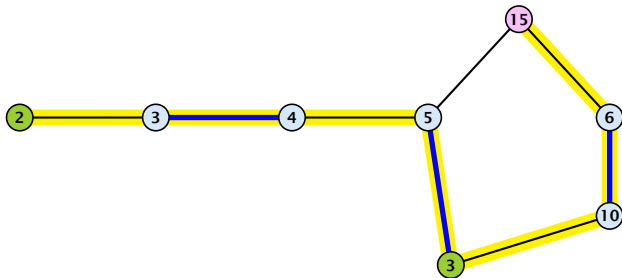




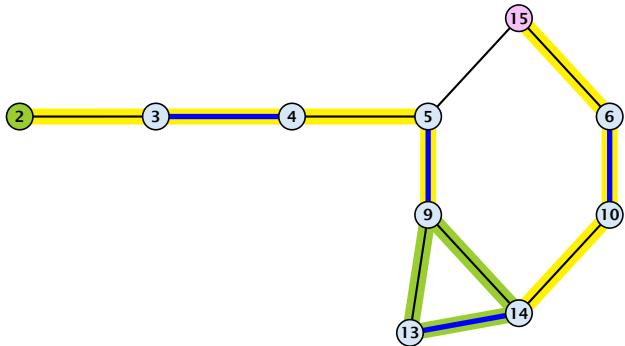
# Example: Blossom Algorithm



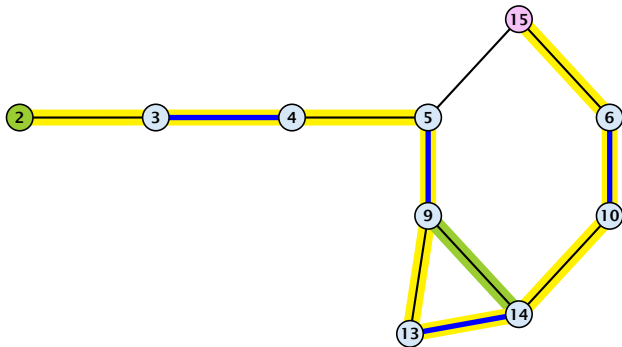
# Example: Blossom Algorithm



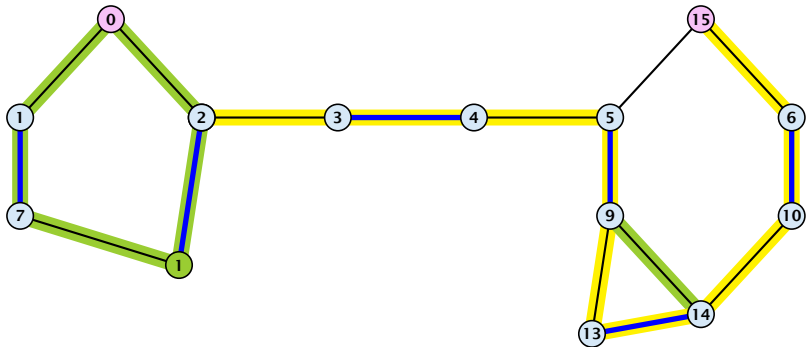
# Example: Blossom Algorithm



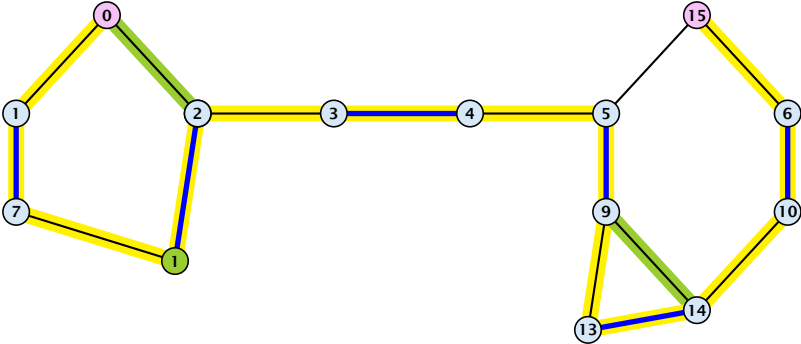
# Example: Blossom Algorithm



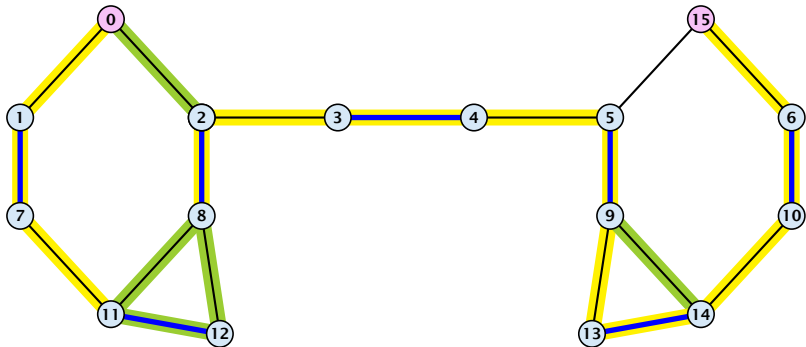
# Example: Blossom Algorithm



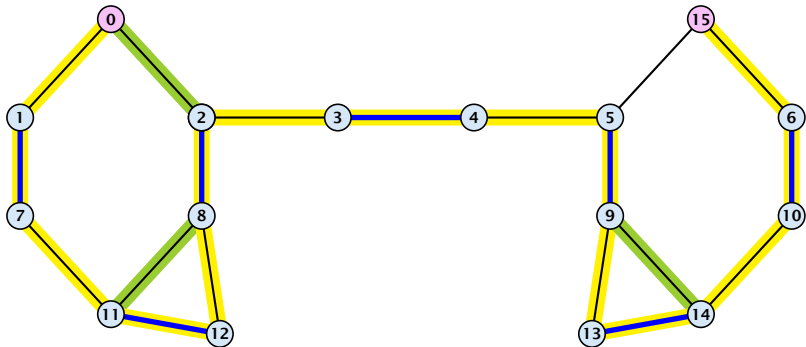
# Example: Blossom Algorithm



# Example: Blossom Algorithm



# Example: Blossom Algorithm





# Example: Blossom Algorithm

