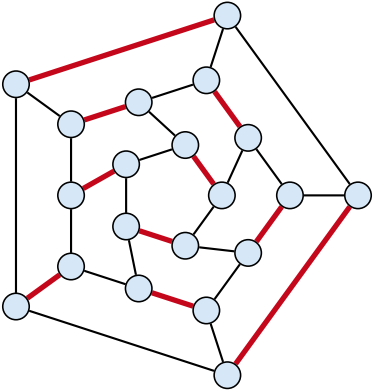
# Part V

# 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: $\mathcal{O}(m\,\mathrm{val}(f^*)) = \mathcal{O}(mn)$.
- 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.

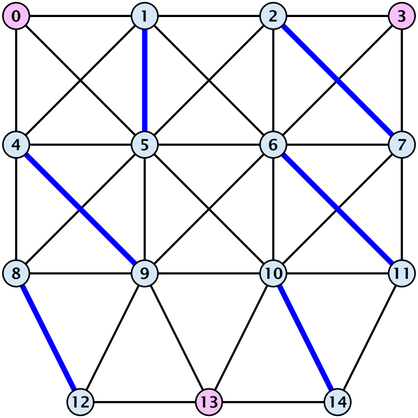
# 17 Augmenting Paths for Matchings

**Definitions.**

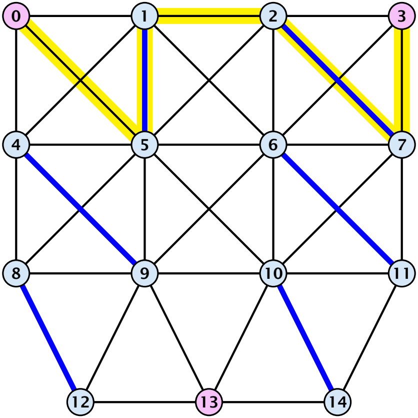
- Given a matching $M$ in a graph $G$, a vertex that is not incident to any edge of $M$ is called a free vertex w. r. .t. $M$.
- For a matching $M$ a path $P$ in $G$ is called an alternating path if edges in $M$ alternate with edges not in $M$.
- An alternating path is called an augmenting path for matching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

**Theorem 6**
*A matching $M$ is a maximum matching if and only if there is no augmenting path w. r. t. $M$.*

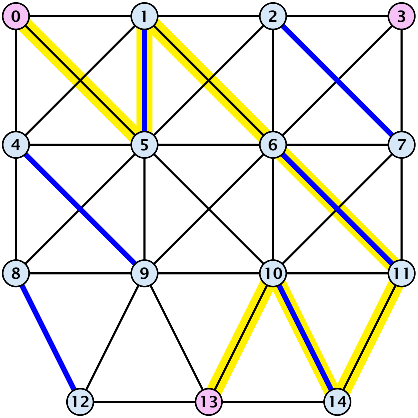
# Augmenting Paths in Action



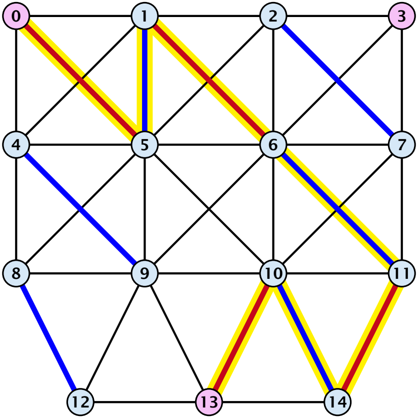
# Augmenting Paths in Action



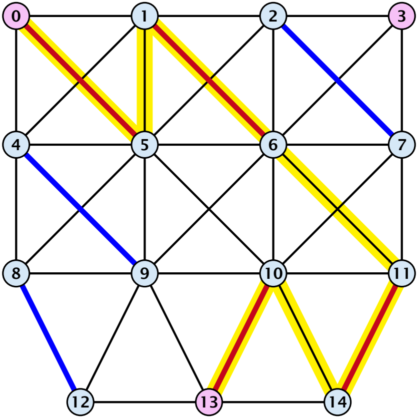
# Augmenting Paths in Action



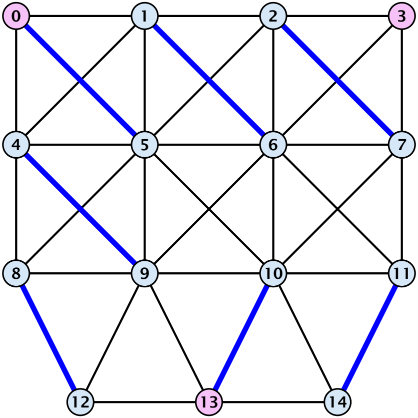
# Augmenting Paths in Action



# Augmenting Paths in Action



# Augmenting Paths in Action



# 17 Augmenting Paths for Matchings

**Proof.**

- $\Rightarrow$ 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.**

- $\Rightarrow$ 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.
- $\Leftarrow$ 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.**

- $\Rightarrow$ 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.
- $\Leftarrow$ 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.**

- $\Rightarrow$ 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.
- $\Leftarrow$ 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 alternating 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 7**

*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 $M'$.*

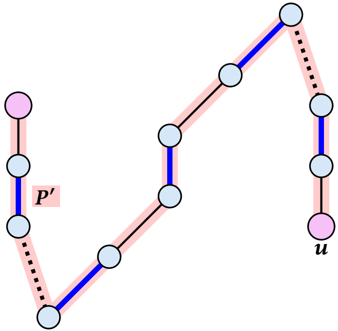
# 17 Augmenting Paths for Matchings

**Proof**

# 17 Augmenting Paths for Matchings

## Proof

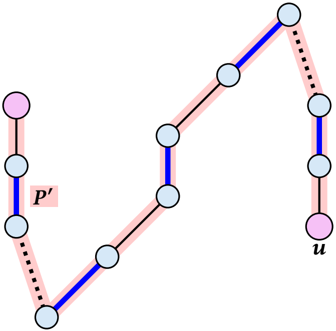
- Assume there is an augmenting path $P'$ w.r.t. $M'$ starting at $u$.



# 17 Augmenting Paths for Matchings

## Proof

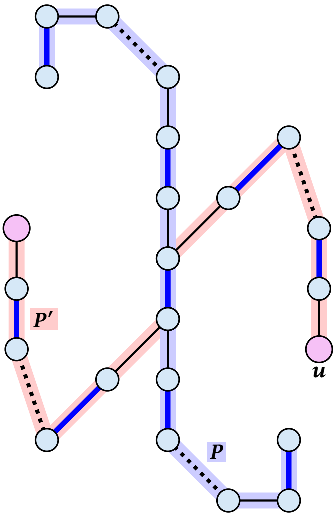
- Assume there is an augmenting path $P'$ w.r.t. $M'$ starting at $u$.
- If $P'$ and $P$ are node-disjoint, $P'$ is also augmenting path w.r.t. $M$ ($\lightning$).



# 17 Augmenting Paths for Matchings

## Proof

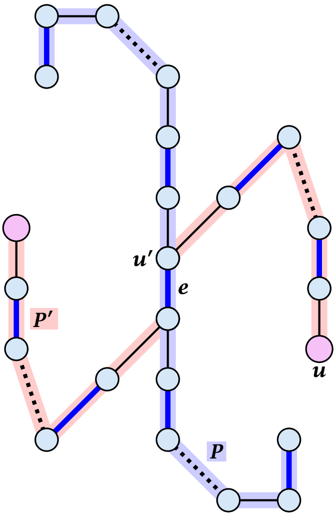
- Assume there is an augmenting path $P'$ w.r.t. $M'$ starting at $u$.
- If $P'$ and $P$ are node-disjoint, $P'$ is also augmenting path w.r.t. $M$ (↯).



# 17 Augmenting Paths for Matchings

**Proof**

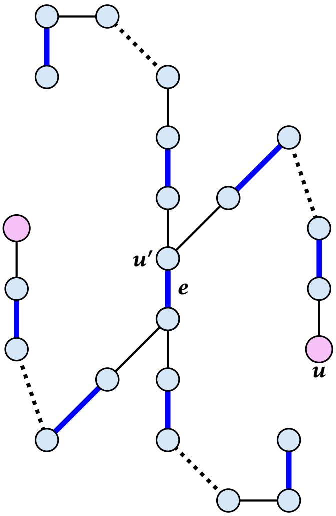
- Assume there is an augmenting path $P'$ w.r.t. $M'$ starting at $u$.
- If $P'$ and $P$ are node-disjoint, $P'$ is also augmenting path w.r.t. $M$ (↯).
- Let $u'$ be the first node on $P'$ that is in $P$, and let $e$ be the matching edge from $M'$ incident to $u'$.



# 17 Augmenting Paths for Matchings

**Proof**

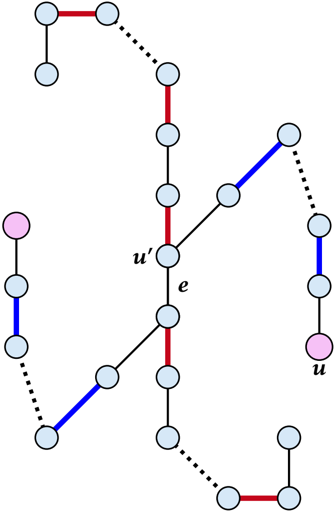
- Assume there is an augmenting path $P'$ w.r.t. $M'$ starting at $u$.
- If $P'$ and $P$ are node-disjoint, $P'$ is also augmenting path w.r.t. $M$ (↯).
- Let $u'$ be the first node on $P'$ that is in $P$, and let $e$ be the matching edge from $M'$ incident to $u'$.



# 17 Augmenting Paths for Matchings

**Proof**

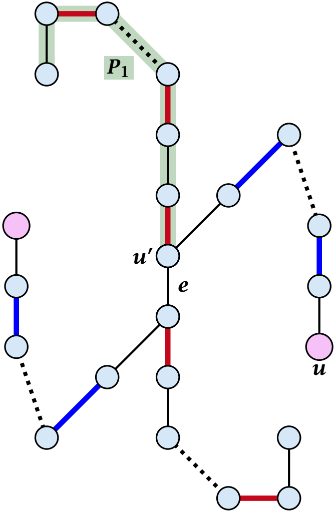
- Assume there is an augmenting path $P'$ w.r.t. $M'$ starting at $u$.
- If $P'$ and $P$ are node-disjoint, $P'$ is also augmenting path w.r.t. $M$ (↯).
- Let $u'$ be the first node on $P'$ that is in $P$, and let $e$ be the matching edge from $M'$ incident to $u'$.



# 17 Augmenting Paths for Matchings

## Proof

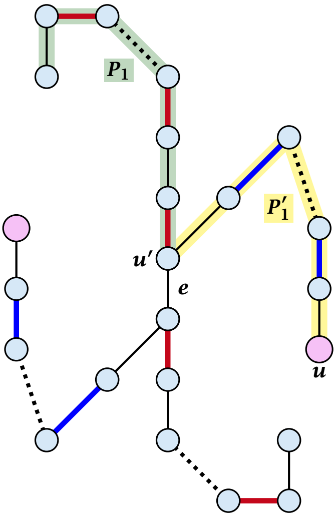
- Assume there is an augmenting path $P'$ w.r.t. $M'$ starting at $u$.
- If $P'$ and $P$ are node-disjoint, $P'$ is also augmenting path w.r.t. $M$ (↯).
- Let $u'$ be the first node on $P'$ that is in $P$, and let $e$ be the matching edge from $M'$ incident to $u'$.
- $u'$ splits $P$ into two parts one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

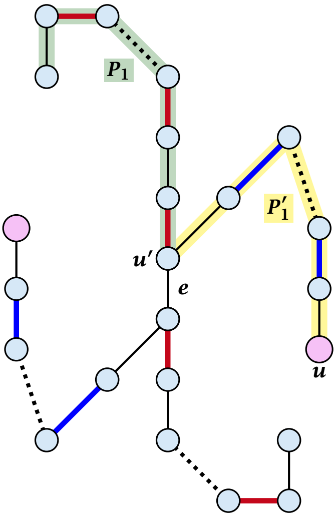
- Assume there is an augmenting path $P'$ w.r.t. $M'$ starting at $u$.
- If $P'$ and $P$ are node-disjoint, $P'$ is also augmenting path w.r.t. $M$ (↯).
- Let $u'$ be the first node on $P'$ that is in $P$, and let $e$ be the matching edge from $M'$ incident to $u'$.
- $u'$ splits $P$ into two parts one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

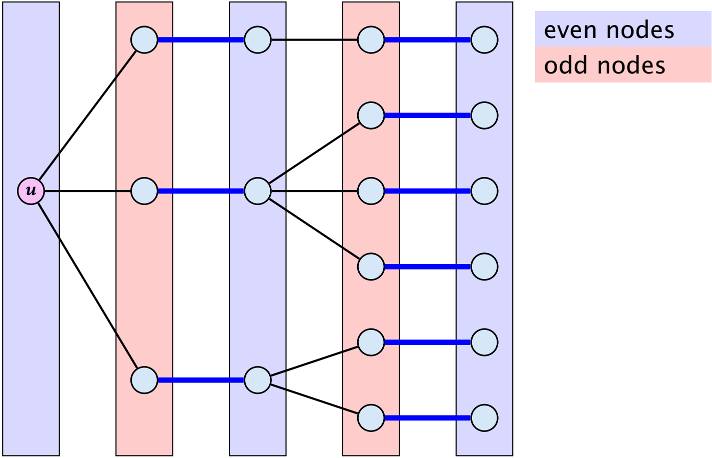
**Proof**

- 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. $M$ ($\lightning$).
- 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 $M$ ($\lightning$).



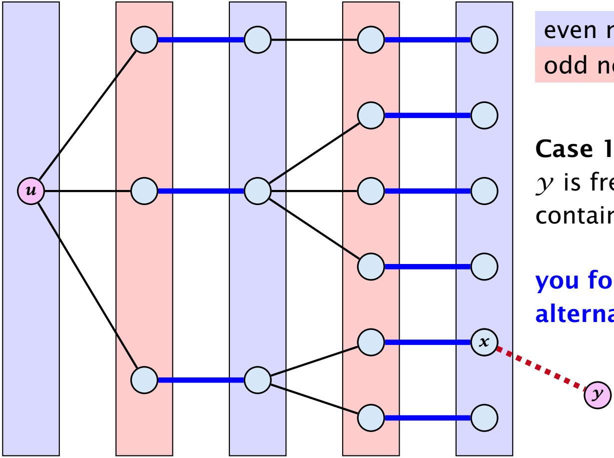
# How to find an augmenting path?

**Construct an alternating tree.**



# How to find an augmenting path?

**Construct an alternating tree.**




even nodes

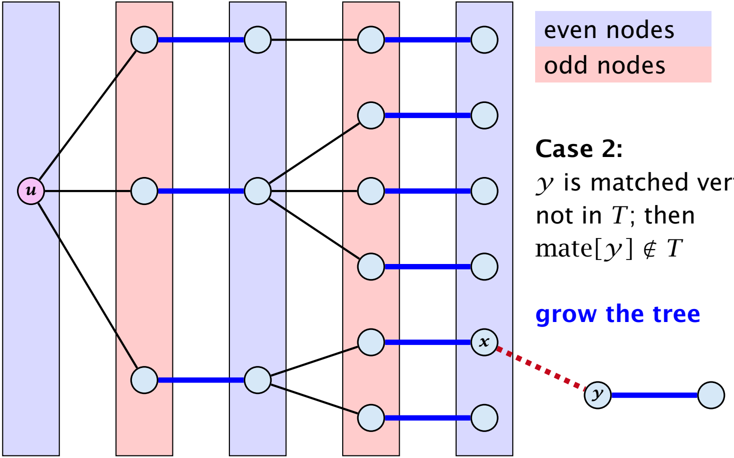
odd nodes

**Case 1:**
$y$ is free vertex not contained in $T$

**you found alternating path**

# How to find an augmenting path?

**Construct an alternating tree.**

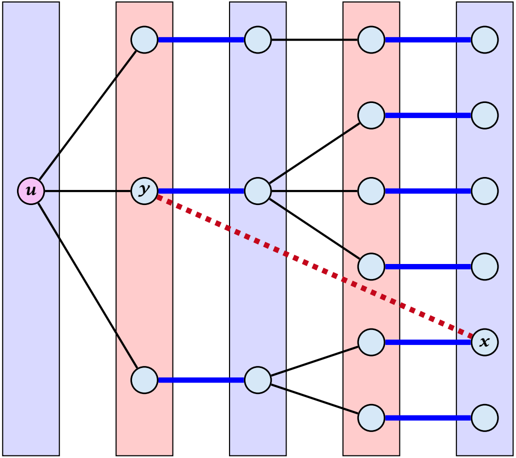



**Case 2:**
$y$ is matched vertex not in $T$; then $\text{mate}[y] \notin T$

**grow the tree**

# How to find an augmenting path?

**Construct an alternating tree.**



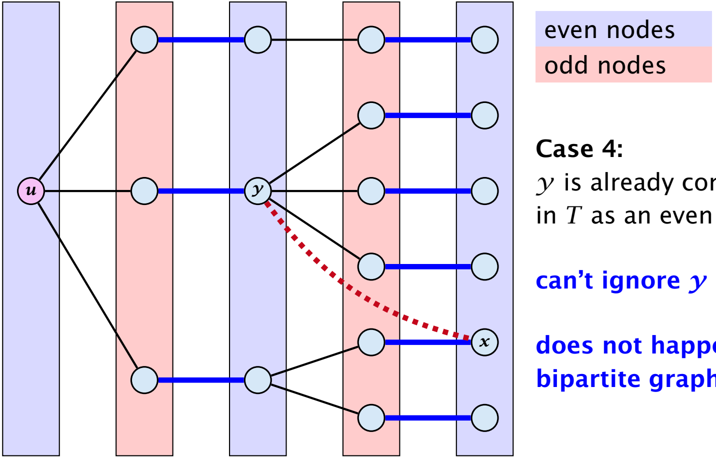

even nodes
odd nodes

**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.**




**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($G$, $match$)

```
for x ∈ V do mate[x] ← 0;
r ← 0; free ← n;
while free ≥ 1 and r < n do
    r ← r + 1
    if mate[r] = 0 then
        for i = 1 to n do parent[i′] ← 0
        Q ← ∅; Q.append(r); aug ← false;
        while aug = false and Q ≠ ∅ do
            x ← Q.dequeue();
            for y ∈ A_x do
                if mate[y] = 0 then
                    augm(mate, parent, y);
                    aug ← true;
                    free ← free − 1;
                else
                    if parent[y] = 0 then
                        parent[y] ← x;
                        Q.enqueue(mate[y]);
```

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

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

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

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

```
for x ∈ V do mate[x] ← 0;
r ← 0; free ← n;
while free ≥ 1 and r < n do
    r ← r + 1
    if mate[r] = 0 then
        for i = 1 to n do parent[i′] ← 0
        Q ← ∅; Q.append(r); aug ← false;
        while aug = false and Q ≠ ∅ do
            x ← Q.dequeue();
            for y ∈ A_x do
                if mate[y] = 0 then
                    augm(mate, parent, y);
                    aug ← true;
                    free ← free − 1;
                else
                    if parent[y] = 0 then
                        parent[y] ← x;
                        Q.enqueue(mate[y]);
```

start with an empty matching

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

```
for x ∈ V do mate[x] ← 0;
r ← 0; free ← n;
while free ≥ 1 and r < n do
    r ← r + 1
    if mate[r] = 0 then
        for i = 1 to n do parent[i′] ← 0
        Q ← ∅; Q.append(r); aug ← false;
        while aug = false and Q ≠ ∅ do
            x ← Q.dequeue();
            for y ∈ A_x do
                if mate[y] = 0 then
                    augm(mate, parent, y);
                    aug ← true;
                    free ← free − 1;
                else
                    if parent[y] = 0 then
                        parent[y] ← x;
                        Q.enqueue(mate[y]);
```

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

$r$: root of current tree

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

```
for x ∈ V do mate[x] ← 0;
r ← 0; free ← n;
while free ≥ 1 and r < n do
    r ← r + 1
    if mate[r] = 0 then
        for i = 1 to n do parent[i′] ← 0
        Q ← ∅; Q.append(r); aug ← false;
        while aug = false and Q ≠ ∅ do
            x ← Q.dequeue();
            for y ∈ A_x do
                if mate[y] = 0 then
                    augm(mate, parent, y);
                    aug ← true;
                    free ← free − 1;
                else
                    if parent[y] = 0 then
                        parent[y] ← x;
                        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$(G, match)$

```
for x ∈ V do mate[x] ← 0;
r ← 0; free ← n;
while free ≥ 1 and r < n do
    r ← r + 1
    if mate[r] = 0 then
        for i = 1 to n do parent[i'] ← 0
        Q ← ∅; Q.append(r); aug ← false;
        while aug = false and Q ≠ ∅ do
            x ← Q.dequeue();
            for y ∈ A_x do
                if mate[y] = 0 then
                    augm(mate, parent, y);
                    aug ← true;
                    free ← free − 1;
                else
                    if parent[y] = 0 then
                        parent[y] ← x;
                        Q.enqueue(mate[y]);
```

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

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

```
for x ∈ V do mate[x] ← 0;
r ← 0; free ← n;
while free ≥ 1 and r < n do
    r ← r + 1
    if mate[r] = 0 then
        for i = 1 to n do parent[i′] ← 0
        Q ← ∅; Q.append(r); aug ← false;
        while aug = false and Q ≠ ∅ do
            x ← Q.dequeue();
            for y ∈ A_x do
                if mate[y] = 0 then
                    augm(mate, parent, y);
                    aug ← true;
                    free ← free − 1;
                else
                    if parent[y] = 0 then
                        parent[y] ← x;
                        Q.enqueue(mate[y]);
```

If $r$ is free start tree construction

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

**for** $x \in V$ **do** $mate[x] \leftarrow 0$;<br>
$r \leftarrow 0$; $free \leftarrow n$;<br>
**while** $free \geq 1$ **and** $r < n$ **do**<br>
    $r \leftarrow r + 1$<br>
    **if** $mate[r] = 0$ **then**<br>
        **for** $i = 1$ **to** $n$ **do** $parent[i'] \leftarrow 0$<br>
        $Q \leftarrow \emptyset$; $Q.\mathrm{append}(r)$; $aug \leftarrow \mathrm{false}$;<br>
        **while** $aug = \mathrm{false}$ **and** $Q \neq \emptyset$ **do**<br>
            $x \leftarrow Q.\mathrm{dequeue}()$;<br>
            **for** $y \in A_x$ **do**<br>
                **if** $mate[y] = 0$ **then**<br>
                    $\mathrm{augm}(mate, parent, y)$;<br>
                    $aug \leftarrow \mathrm{true}$;<br>
                    $free \leftarrow free - 1$;<br>
                **else**<br>
                    **if** $parent[y] = 0$ **then**<br>
                        $parent[y] \leftarrow x$;<br>
                        $Q.\mathrm{enqueue}(mate[y])$;

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

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

```
for x ∈ V do mate[x] ← 0;
r ← 0; free ← n;
while free ≥ 1 and r < n do
    r ← r + 1
    if mate[r] = 0 then
        for i = 1 to n do parent[i′] ← 0
        Q ← ∅; Q.append(r); aug ← false;
        while aug = false and Q ≠ ∅ do
            x ← Q.dequeue();
            for y ∈ A_x do
                if mate[y] = 0 then
                    augm(mate, parent, y);
                    aug ← true;
                    free ← free − 1;
                else
                    if parent[y] = 0 then
                        parent[y] ← x;
                        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($G$, $match$)

```
for x ∈ V do mate[x] ← 0;
r ← 0; free ← n;
while free ≥ 1 and r < n do
    r ← r + 1
    if mate[r] = 0 then
        for i = 1 to n do parent[i′] ← 0
        Q ← ∅; Q.append(r); aug ← false;
        while aug = false and Q ≠ ∅ do
            x ← Q.dequeue();
            for y ∈ A_x do
                if mate[y] = 0 then
                    augm(mate, parent, y);
                    aug ← true;
                    free ← free − 1;
                else
                    if parent[y] = 0 then
                        parent[y] ← x;
                        Q.enqueue(mate[y]);
```

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

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

```
for x ∈ V do mate[x] ← 0;
r ← 0; free ← n;
while free ≥ 1 and r < n do
    r ← r + 1
    if mate[r] = 0 then
        for i = 1 to n do parent[i′] ← 0
        Q ← ∅; Q.append(r); aug ← false;
        while aug = false and Q ≠ ∅ do
            x ← Q.dequeue();
            for y ∈ A_x do
                if mate[y] = 0 then
                    augm(mate, parent, y);
                    aug ← true;
                    free ← free − 1;
                else
                    if parent[y] = 0 then
                        parent[y] ← x;
                        Q.enqueue(mate[y]);
```

take next unexamined leaf

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

```
for x ∈ V do mate[x] ← 0;
r ← 0; free ← n;
while free ≥ 1 and r < n do
    r ← r + 1
    if mate[r] = 0 then
        for i = 1 to n do parent[i′] ← 0
        Q ← ∅; Q.append(r); aug ← false;
        while aug = false and Q ≠ ∅ do
            x ← Q.dequeue();
            for y ∈ A_x do
                if mate[y] = 0 then
                    augm(mate, parent, y);
                    aug ← true;
                    free ← free − 1;
                else
                    if parent[y] = 0 then
                        parent[y] ← x;
                        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($G$, $match$)

```
for x ∈ V do mate[x] ← 0;
r ← 0; free ← n;
while free ≥ 1 and r < n do
    r ← r + 1
    if mate[r] = 0 then
        for i = 1 to n do parent[i′] ← 0
        Q ← ∅; Q.append(r); aug ← false;
        while aug = false and Q ≠ ∅ do
            x ← Q.dequeue();
            for y ∈ A_x do
                if mate[y] = 0 then
                    augm(mate, parent, y);
                    aug ← true;
                    free ← free − 1;
                else
                    if parent[y] = 0 then
                        parent[y] ← x;
                        Q.enqueue(mate[y]);
```

do an augmentation...

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

```
for x ∈ V do mate[x] ← 0;
r ← 0; free ← n;
while free ≥ 1 and r < n do
    r ← r + 1
    if mate[r] = 0 then
        for i = 1 to n do parent[i′] ← 0
        Q ← ∅; Q.append(r); aug ← false;
        while aug = false and Q ≠ ∅ do
            x ← Q.dequeue();
            for y ∈ A_x do
                if mate[y] = 0 then
                    augm(mate, parent, y);
                    aug ← true;
                    free ← free − 1;
                else
                    if parent[y] = 0 then
                        parent[y] ← x;
                        Q.enqueue(mate[y]);
```

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

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

```
for x ∈ V do mate[x] ← 0;
r ← 0; free ← n;
while free ≥ 1 and r < n do
    r ← r + 1
    if mate[r] = 0 then
        for i = 1 to n do parent[i′] ← 0
        Q ← ∅; Q.append(r); aug ← false;
        while aug = false and Q ≠ ∅ do
            x ← Q.dequeue();
            for y ∈ A_x do
                if mate[y] = 0 then
                    augm(mate, parent, y);
                    aug ← true;
                    free ← free − 1;
                else
                    if parent[y] = 0 then
                        parent[y] ← x;
                        Q.enqueue(mate[y]);
```

reduce number of free nodes

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

```
for x ∈ V do mate[x] ← 0;
r ← 0; free ← n;
while free ≥ 1 and r < n do
    r ← r + 1
    if mate[r] = 0 then
        for i = 1 to n do parent[i′] ← 0
        Q ← ∅; Q.append(r); aug ← false;
        while aug = false and Q ≠ ∅ do
            x ← Q.dequeue();
            for y ∈ A_x do
                if mate[y] = 0 then
                    augm(mate, parent, y);
                    aug ← true;
                    free ← free − 1;
                else
                    if parent[y] = 0 then
                        parent[y] ← x;
                        Q.enqueue(mate[y]);
```

if $y$ is not in the tree yet

**Algorithm 49** BiMatch$(G, \mathit{match})$

```
for x ∈ V do mate[x] ← 0;
r ← 0; free ← n;
while free ≥ 1 and r < n do
    r ← r + 1
    if mate[r] = 0 then
        for i = 1 to n do parent[i′] ← 0
        Q ← ∅; Q.append(r); aug ← false;
        while aug = false and Q ≠ ∅ do
            x ← Q.dequeue();
            for y ∈ A_x do
                if mate[y] = 0 then
                    augm(mate, parent, y);
                    aug ← true;
                    free ← free − 1;
                else
                    if parent[y] = 0 then
                        parent[y] ← x;
                        Q.enqueue(mate[y]);
```

...put it into the tree

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

```
for x ∈ V do mate[x] ← 0;
r ← 0; free ← n;
while free ≥ 1 and r < n  do
    r ← r + 1
    if mate[r] = 0  then
        for i = 1 to n do parent[i′] ← 0
        Q ← ∅; Q.append(r); aug ← false;
        while aug = false and Q ≠ ∅  do
            x ← Q.dequeue();
            for y ∈ A_x do
                if mate[y] = 0  then
                    augm(mate, parent, y);
                    aug ← true;
                    free ← free − 1;
                else
                    if parent[y] = 0  then
                        parent[y] ← x;
                        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?]):**

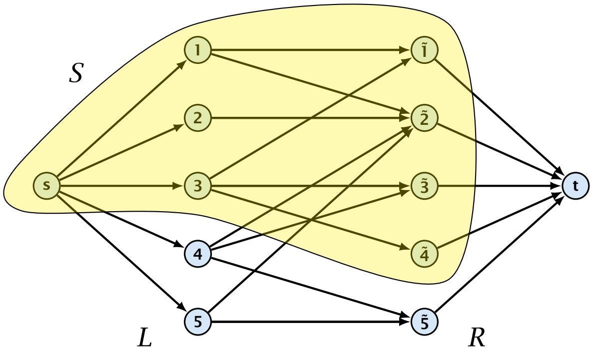
- assume that $|L| = |R| = n$
- assume that there is an edge between every pair of nodes $(\ell, r) \in V \times V$
- can assume goal is to construct maximum weight perfect matching

# Weighted Bipartite Matching

**Theorem 8 (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:**

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

# 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:**

- $\Leftarrow$ Of course, the condition is necessary as otherwise not all nodes in $S$ could be matched to different neigbhours.
- $\Rightarrow$ 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 $L_S \overset{\text{def}}{=} L \cap S$ and $R_S \overset{\text{def}}{=} R \cap S$ denote the portion of $S$ inside $L$ and $R$, respectively.

# Halls Theorem

**Proof:**

- $\Leftarrow$ Of course, the condition is necessary as otherwise not all nodes in $S$ could be matched to different neighbours.
- $\Rightarrow$ 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:**

- $\Leftarrow$ Of course, the condition is necessary as otherwise not all nodes in $S$ could be matched to different neighbours.
- $\Rightarrow$ 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:**

- $\Leftarrow$ Of course, the condition is necessary as otherwise not all nodes in $S$ could be matched to different neighbours.
- $\Rightarrow$ 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:**

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

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

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

$$\sum_{(u,v)\in M} w_{(u,v)} \le \sum_{(u,v)\in M} (x_u + x_v) = \sum_v x_v \enspace .$$

# Algorithm Outline

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

Then, Halls 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, Halls 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, Halls 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 $H(\vec{x})$.

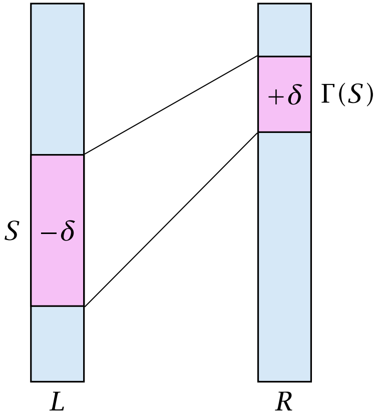
**Idea:** reweight such that:

- 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

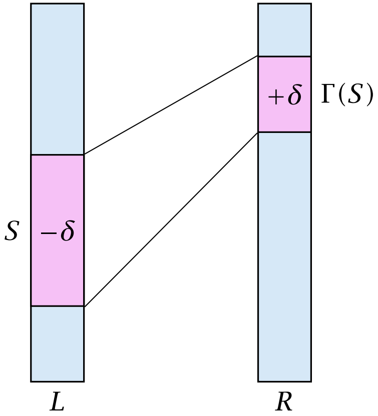
Increase node-weights in $\Gamma(S)$ by $+\delta$, and decrease the node-weights in $S$ by $-\delta$.



# Changing Node Weights

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

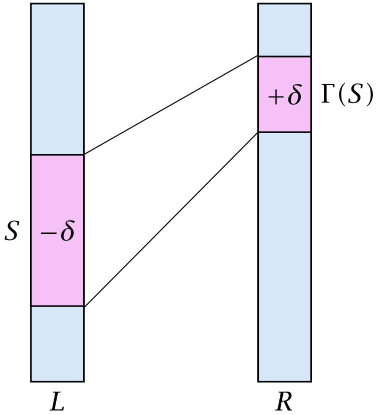
- Total node-weight decreases.



# Changing Node Weights

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

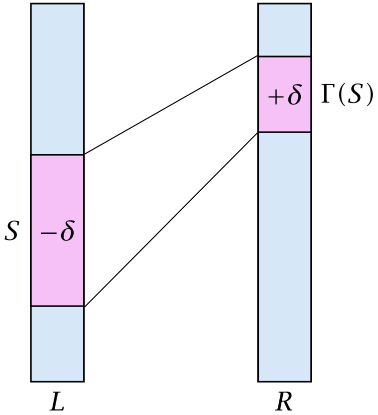
- Total node-weight decreases.
- Only edges from $S$ to $R - \Gamma(S)$ decrease in their weight.



# Changing Node Weights

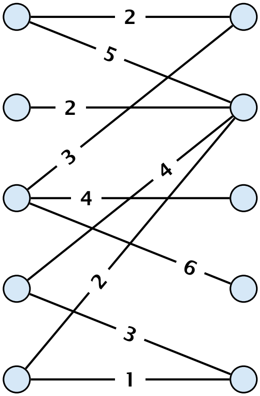
Increase node-weights in $\Gamma(S)$ by $+\delta$, and decrease the node-weights in $S$ by $-\delta$.

- 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.



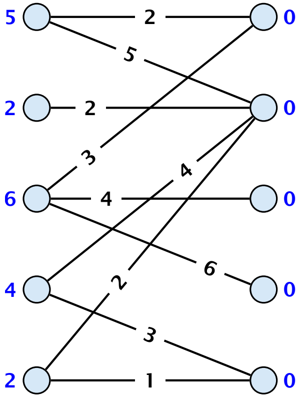
# Weighted Bipartite Matching

Edges not drawn have weight $0$.



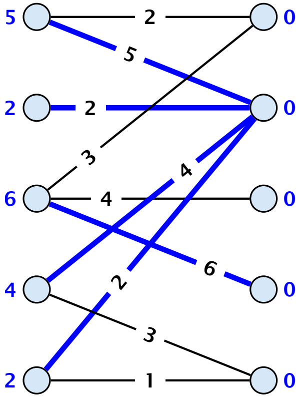
# Weighted Bipartite Matching

Edges not drawn have weight $0$.



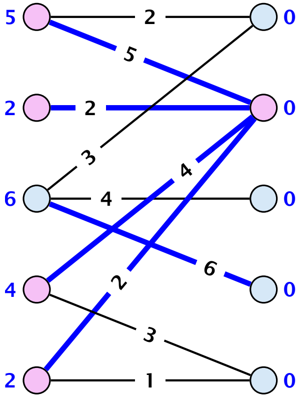
# Weighted Bipartite Matching

Edges not drawn have weight $0$.



# Weighted Bipartite Matching

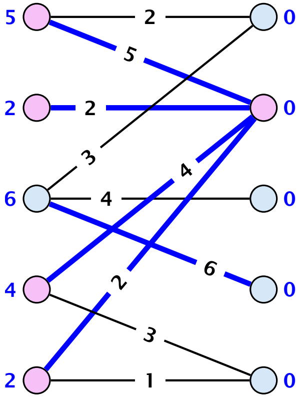
Edges not drawn have weight $0$.



# Weighted Bipartite Matching

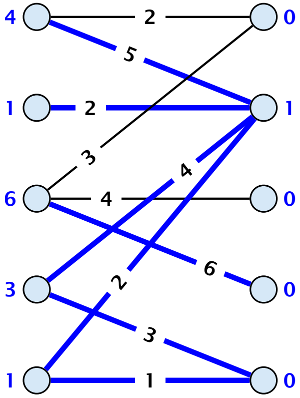
Edges not drawn have weight $0$.

$\delta = 1$



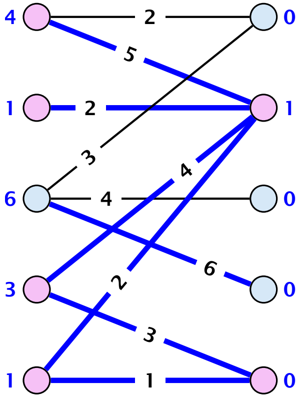
# Weighted Bipartite Matching

Edges not drawn have weight $0$.



# Weighted Bipartite Matching

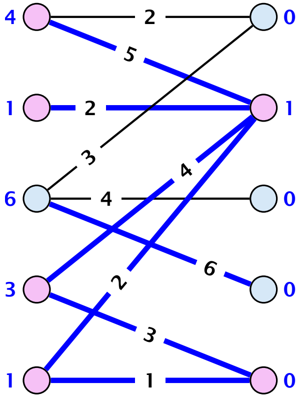
Edges not drawn have weight $0$.



# Weighted Bipartite Matching

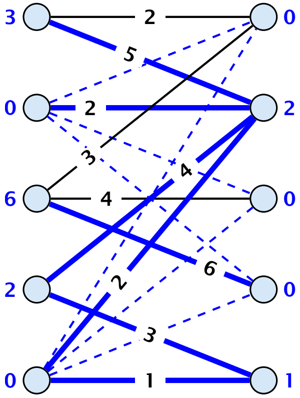
Edges not drawn have weight $0$.

$\delta = 1$



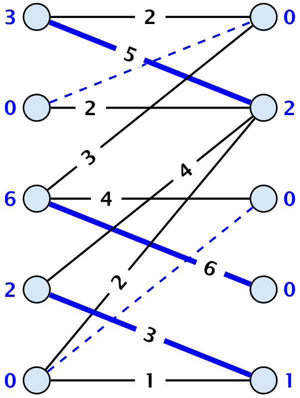
# Weighted Bipartite Matching

Edges not drawn have weight $0$.



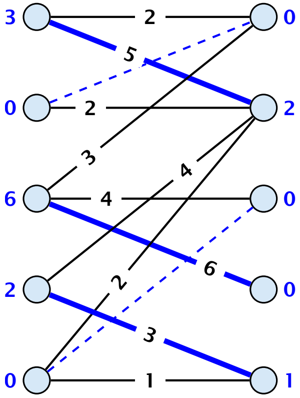
# Weighted Bipartite Matching

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).

# 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).
- 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 $R - \Gamma(S)$.

# Analysis

**How many iterations do we need?**

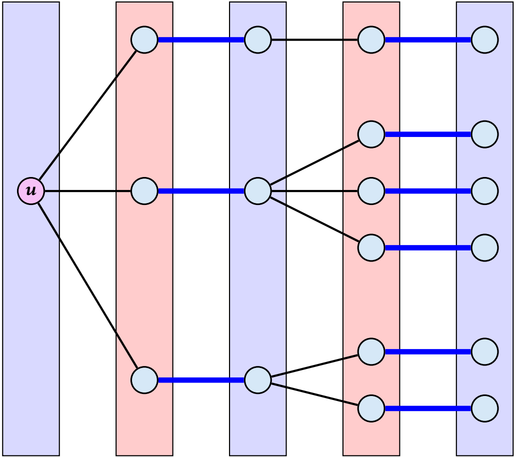
- One reweighting step increases the number of edges out of $S$ by at least one.
- Assume that we have a maximum matching that saturates the set $\Gamma(S)$, in the sense that every node in $\Gamma(S)$ is matched to a node in $S$ (we will show that we can always find $S$ and a matching such that this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

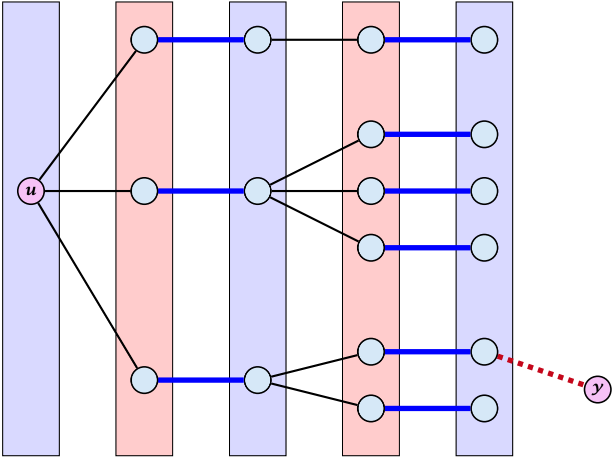
# How to find an augmenting path?

**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 $n$ reweights 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 $n$ 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.

# Analysis

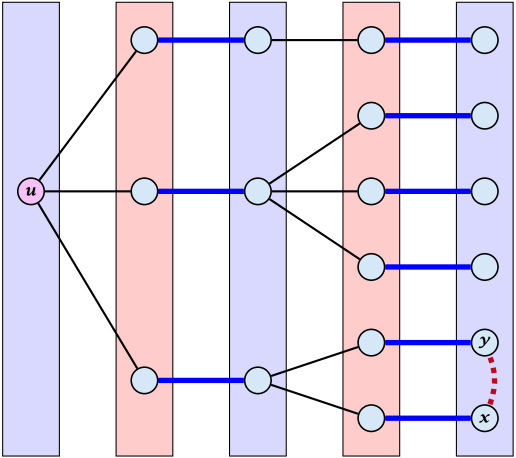
- The current matching does not have any edges from $V_{\text{odd}}$ to $L \setminus V_{\text{even}}$ (edges that may possibly be deleted by changing weights).
- After changing weights, there is at least one more edge connecting $V_{\text{even}}$ to a node outside of $V_{\text{odd}}$. After at most $n$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



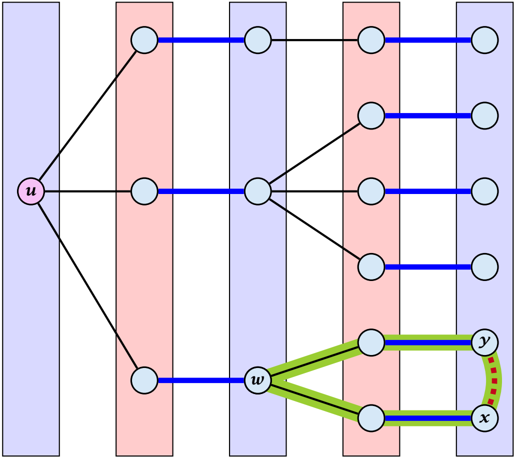

even nodes
odd nodes

**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

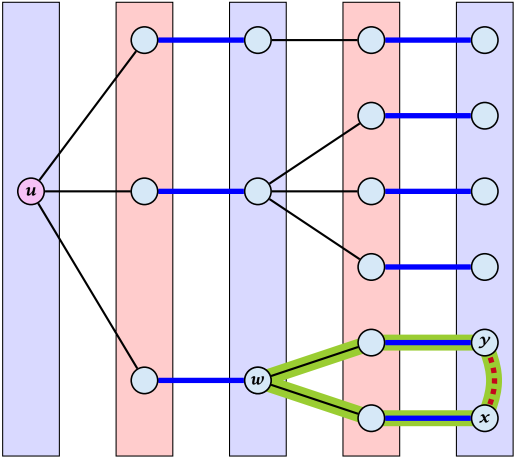
odd nodes

**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 9**
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 9**

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).

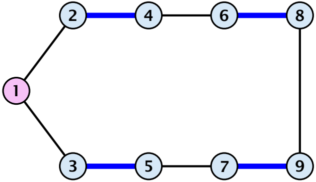
# Flowers and Blossoms

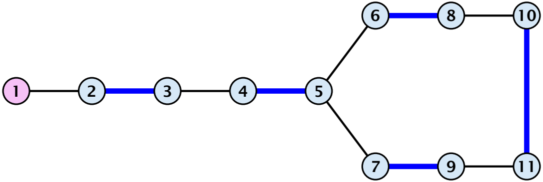
**Definition 9**

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:**

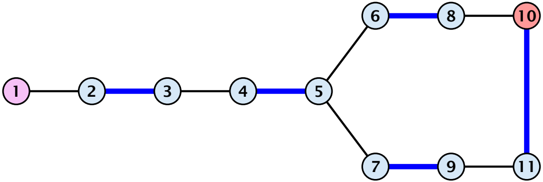
4. Every node $x$ in the blossom (except its base) is reachable from the root (or from the base of the blossom) through two distinct alternating paths; one with even and one with odd length.

# 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 $G$.

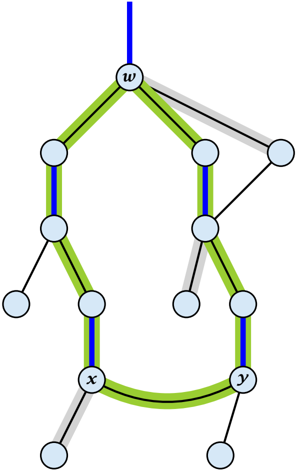
# Shrinking Blossoms

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 $B$.

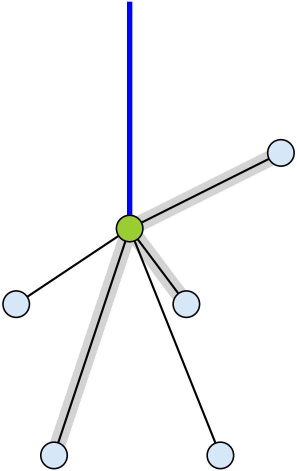
# Shrinking Blossoms

- 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 $G'$.

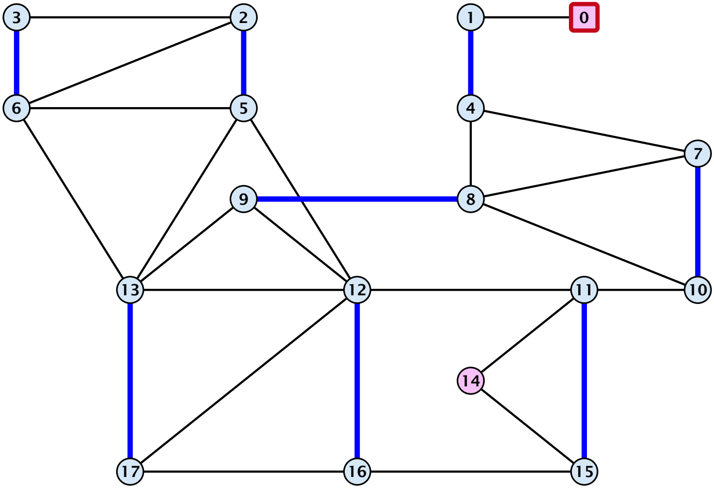


# Shrinking Blossoms

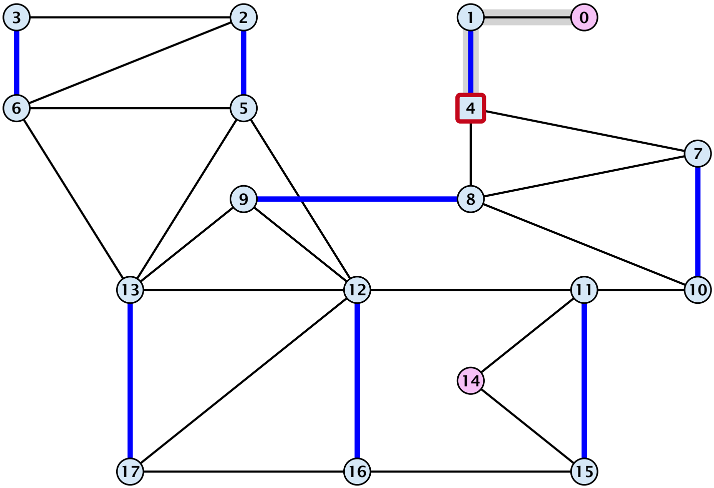
- Edges of $T$ that connect a node $u$ not in $B$ to a node in $B$ become tree edges in $T'$ connecting $u$ to $b$.
- Matching edges (there is at most one) that connect a node $u$ not in $B$ to a node in $B$ become matching edges in $M'$.
- Nodes that are connected in $G$ to at least one node in $B$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



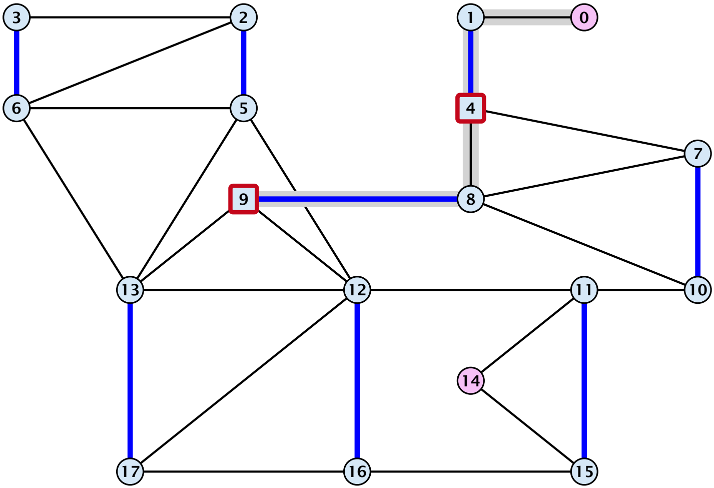
# Example: Blossom Algorithm



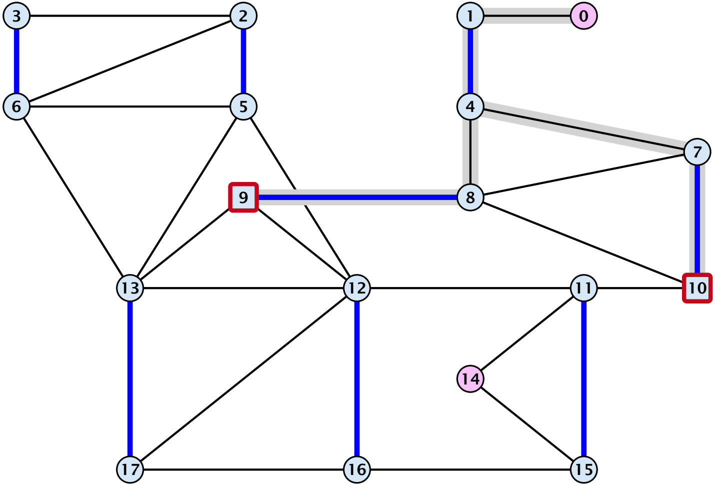
# Example: Blossom Algorithm



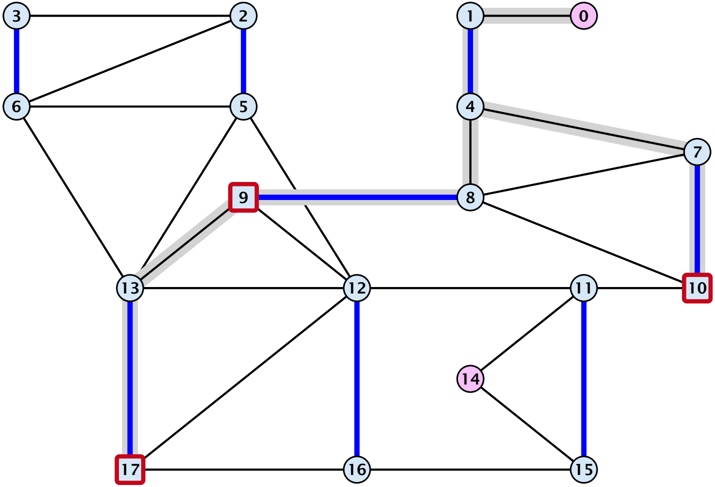
# Example: Blossom Algorithm



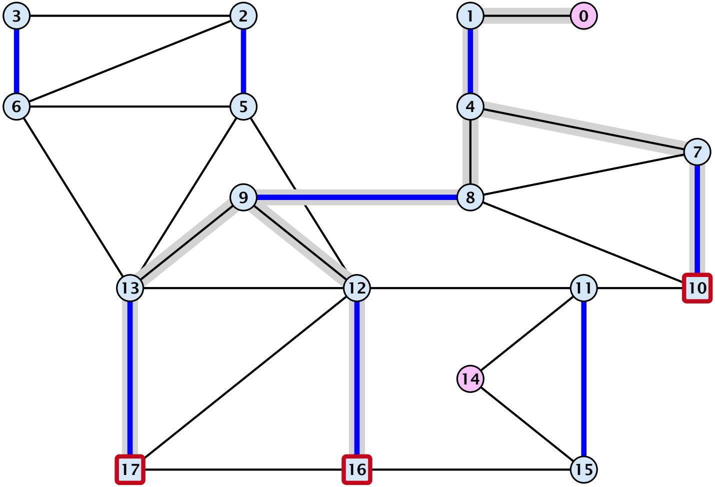
# Example: Blossom Algorithm



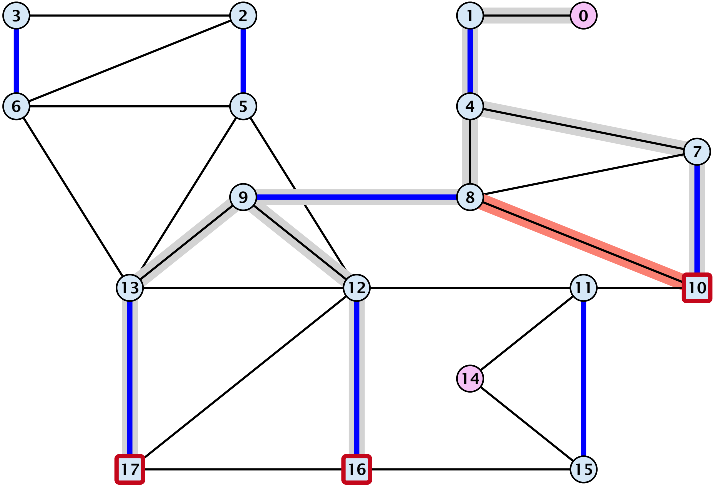
# Example: Blossom Algorithm



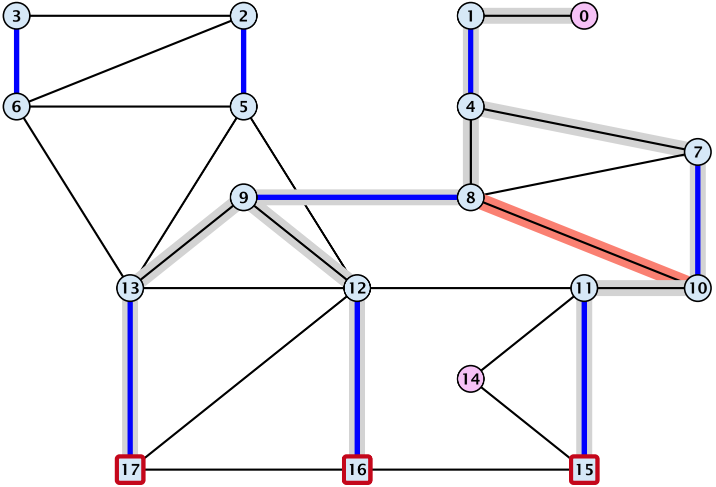
# Example: Blossom Algorithm



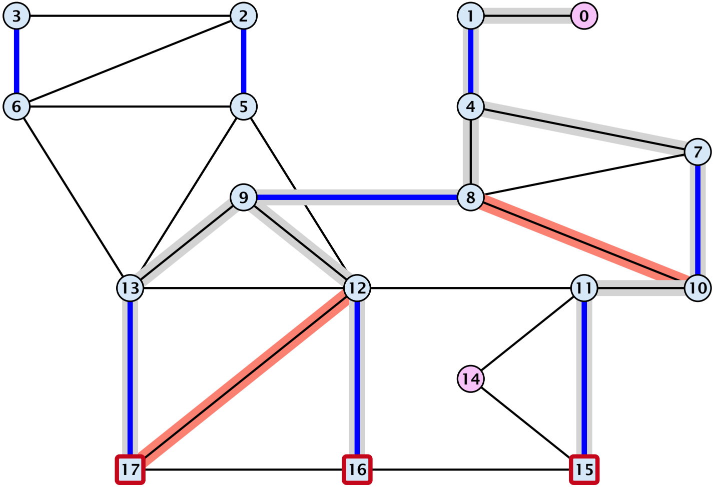
# Example: Blossom Algorithm



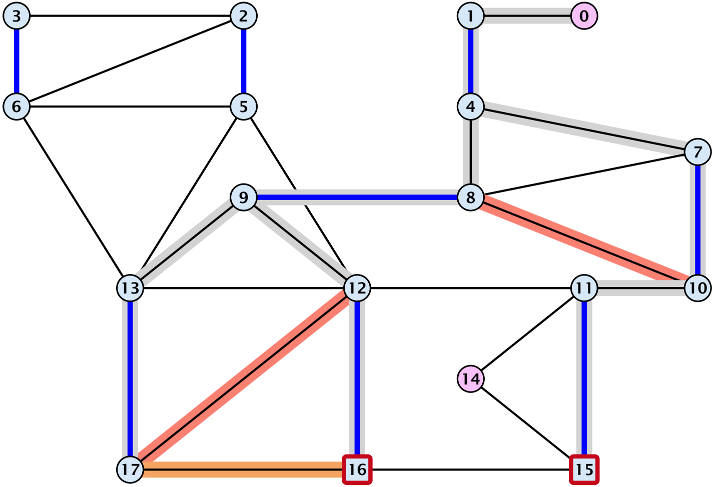
# Example: Blossom Algorithm



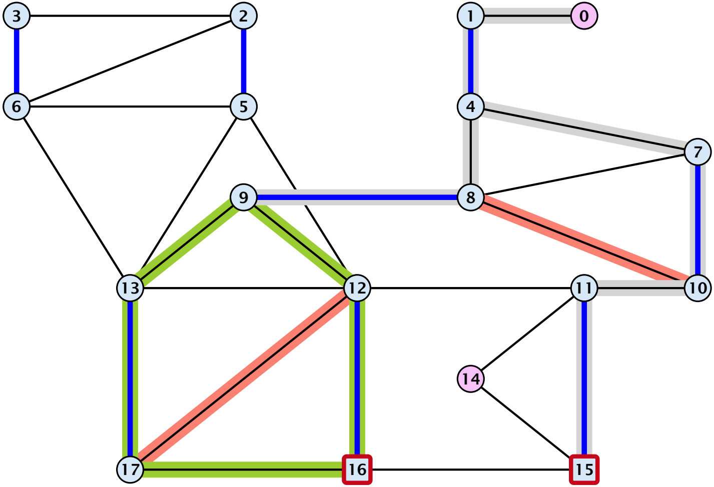
# Example: Blossom Algorithm



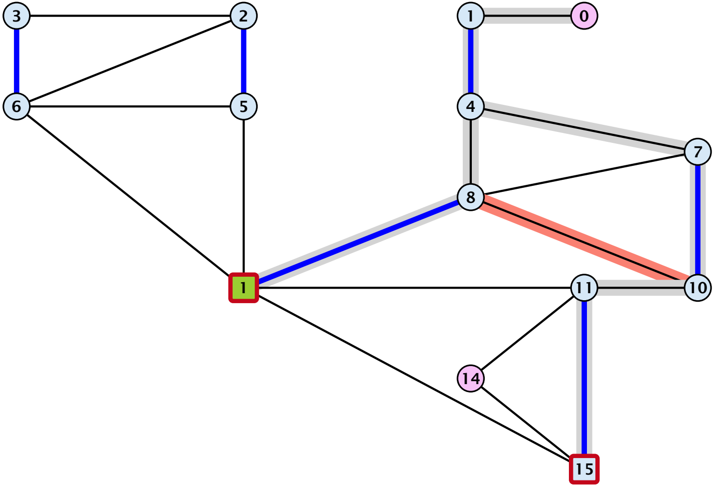
# Example: Blossom Algorithm



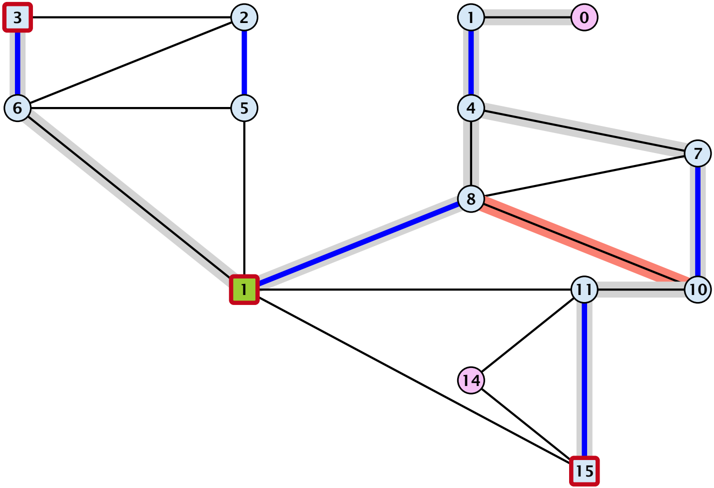
# Example: Blossom Algorithm



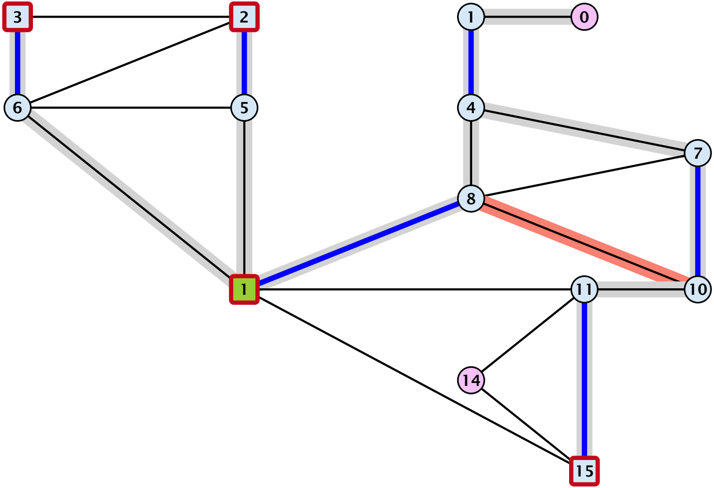
# Example: Blossom Algorithm



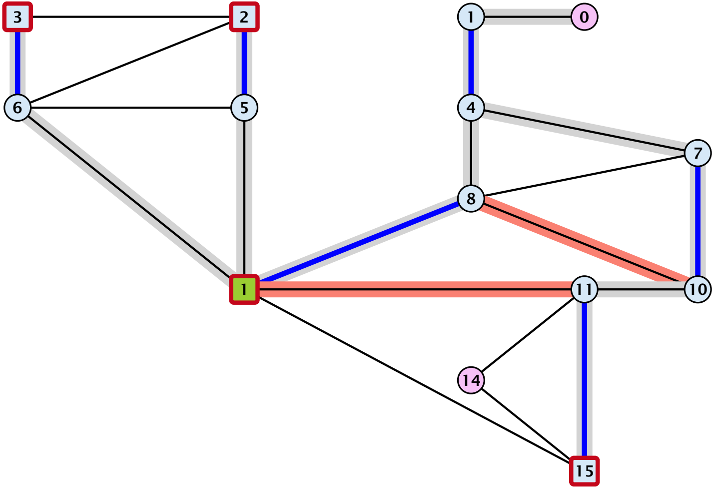
# Example: Blossom Algorithm



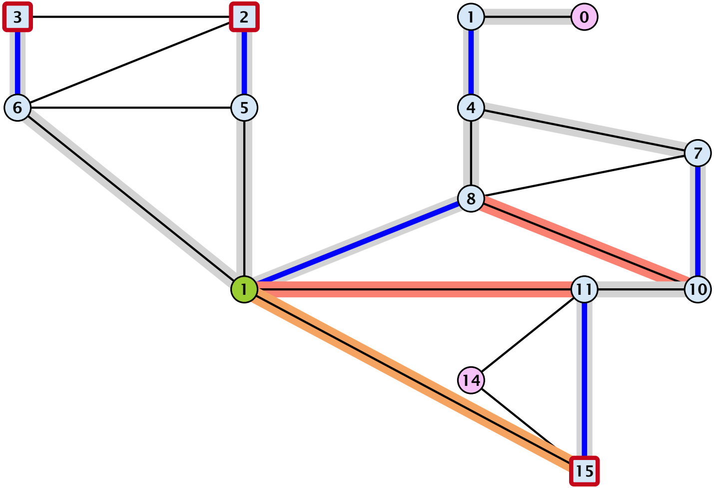
# Example: Blossom Algorithm



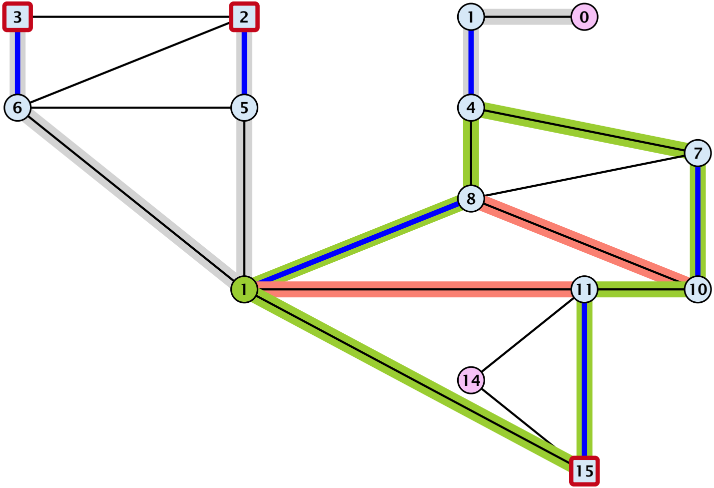
# Example: Blossom Algorithm



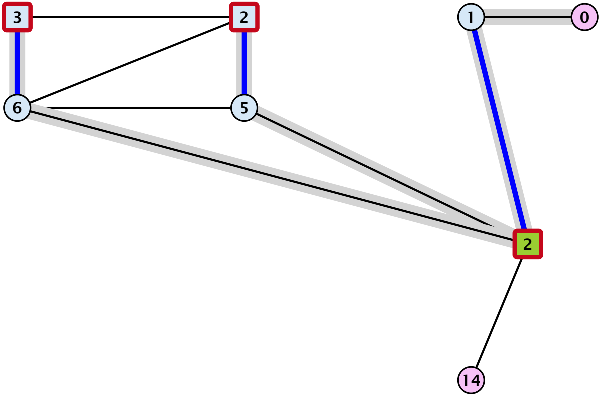
# Example: Blossom Algorithm



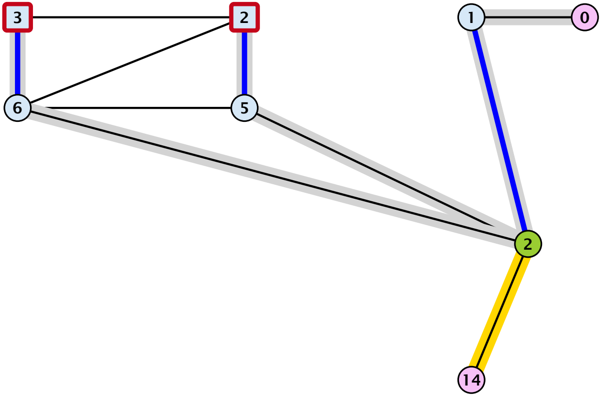
# Example: Blossom Algorithm



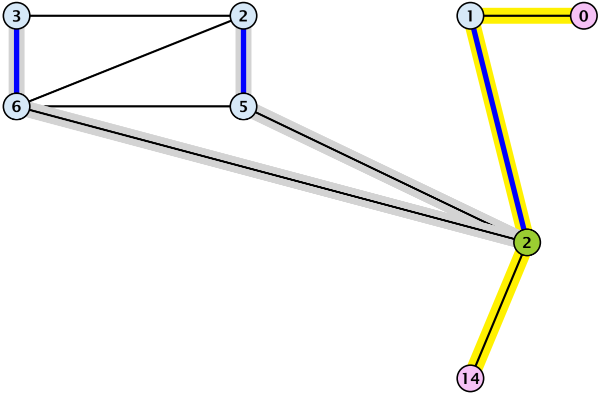
# Example: Blossom Algorithm



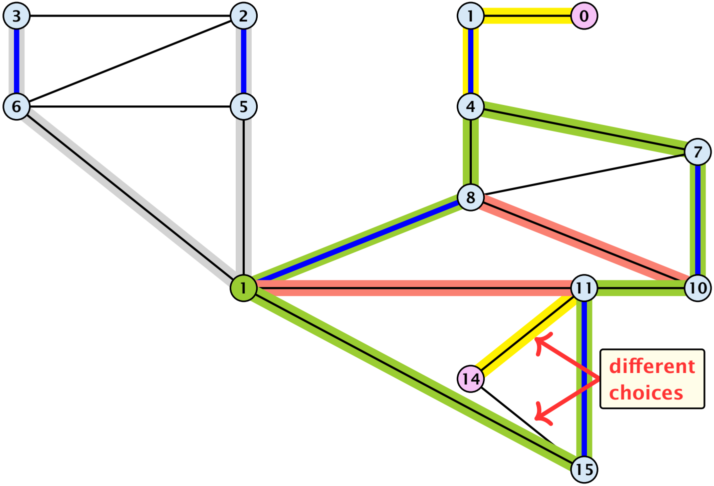
# Example: Blossom Algorithm



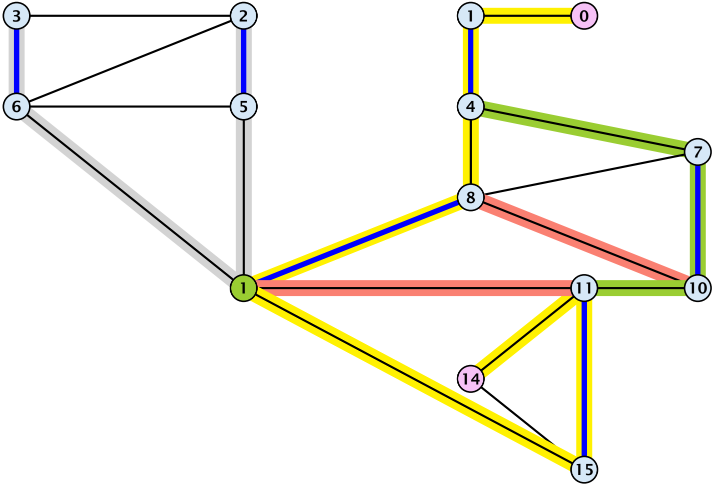
# Example: Blossom Algorithm



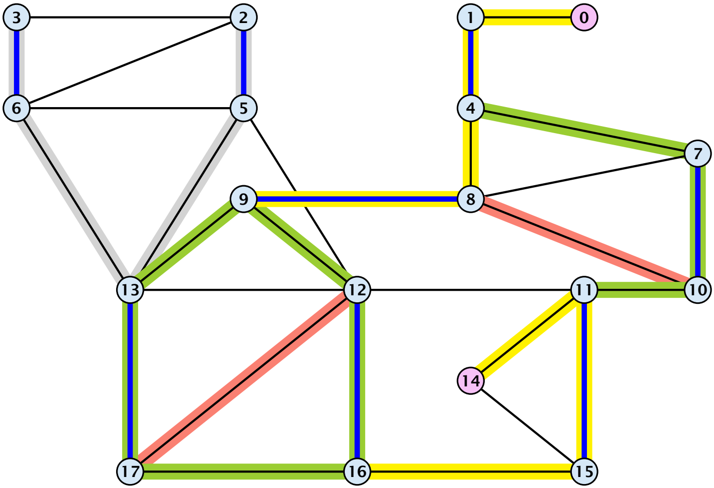
# Example: Blossom Algorithm



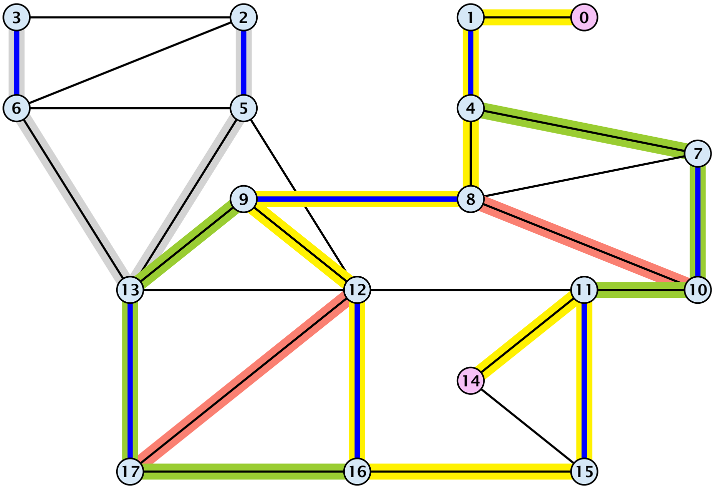
# Example: Blossom Algorithm



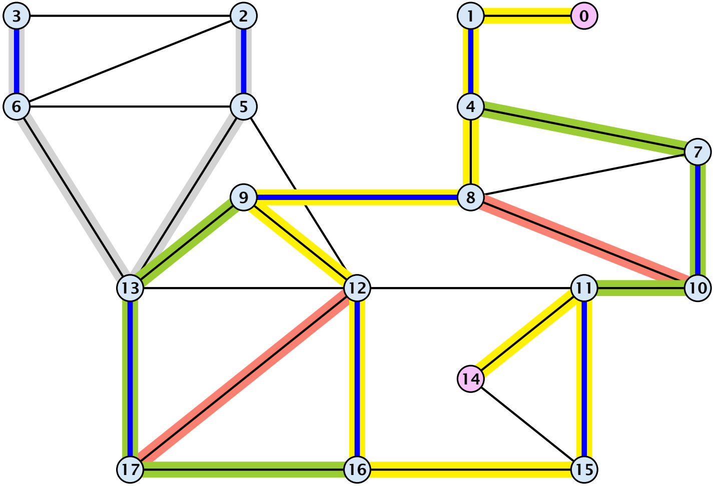
# Example: Blossom Algorithm



# 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.

# 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.

**Lemma 10**

*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.

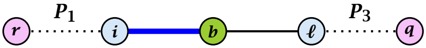
# Correctness

**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.



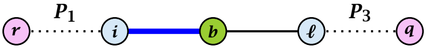
# Correctness

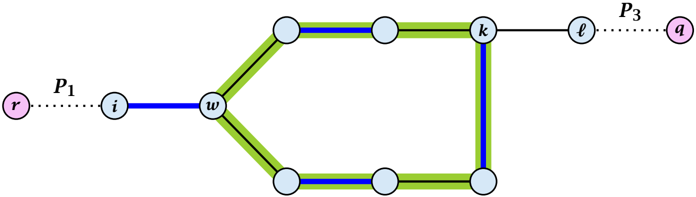
**Proof.**

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**

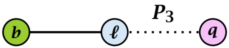
- If the stem is empty then after expanding the blossom, $w = r$.

# Correctness

**Proof.**

**Case 2: empty stem**

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

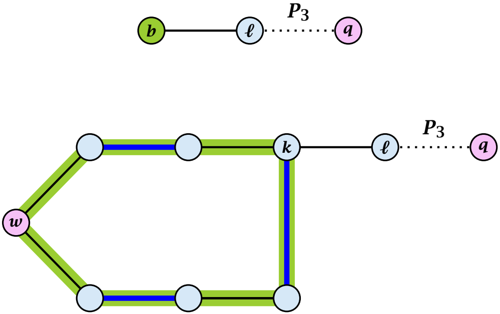


# Correctness

**Proof.**

**Case 2: empty stem**

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

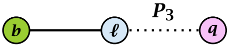


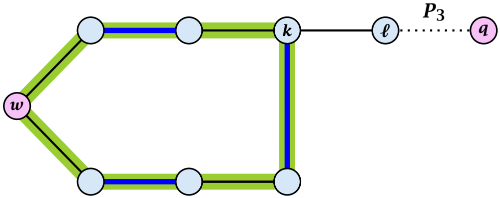
# Correctness

**Proof.**

**Case 2: empty stem**

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




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

# Correctness

**Lemma 11**
*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**

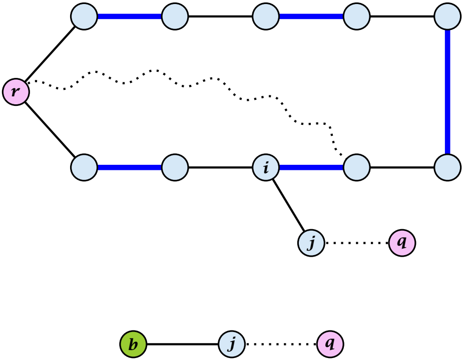
Let $i$ be the last node on the path $P$ that is part of the blossom.

$P$ 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** search($r$, *found*)

set $\bar{A}(i) \leftarrow A(i)$ for all nodes $i$
*found* $\leftarrow$ false
unlabel all nodes;
give an even label to $r$ and initialize *list* $\leftarrow \{r\}$
**while** *list* $\neq \emptyset$ **do**
    delete a node $i$ from *list*
    examine($i$, *found*)
    **if** *found* $=$ true **then return**

Search for an augmenting path starting at $r$.

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

```
set Ā(i) ← A(i) for all nodes i
found ← false
unlabel all nodes;
give an even label to r and initialize list ← {r}
while list ≠ ∅ do
    delete a node i from list
    examine(i, found)
    if found = 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** search($r$, *found*)

set $\bar{A}(i) \leftarrow A(i)$ for all nodes $i$
*found* $\leftarrow$ false
unlabel all nodes;
give an even label to $r$ and initialize *list* $\leftarrow \{r\}$
**while** *list* $\neq \emptyset$ **do**
    delete a node $i$ from *list*
    examine($i$, *found*)
    **if** *found* $=$ true **then return**

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

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

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

In the beginning no node is in the tree.

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

set $\bar{A}(i) \leftarrow A(i)$ for all nodes $i$
*found* $\leftarrow$ false
unlabel all nodes;
give an even label to $r$ and initialize *list* $\leftarrow \{r\}$
**while** *list* $\neq \emptyset$ **do**
    delete a node $i$ from *list*
    examine($i$, *found*)
    **if** *found* $=$ true **then return**

Put the root in the tree.

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

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

set $\bar{A}(i) \leftarrow A(i)$ for all nodes $i$
*found* $\leftarrow$ false
unlabel all nodes;
give an even label to $r$ and initialize *list* $\leftarrow \{r\}$
**while** *list* $\neq \emptyset$ **do**
  delete a node $i$ from *list*
  examine($i$, *found*)
  **if** *found* $=$ true **then return**

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

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

```
set Ā(i) ← A(i) for all nodes i
found ← false
unlabel all nodes;
give an even label to r and initialize list ← {r}
while list ≠ ∅ do
    delete a node i from list
    examine(i, found)
    if found = true then return
```

...examine the next one

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

```
set Ā(i) ← A(i) for all nodes i
found ← false
unlabel all nodes;
give an even label to r and initialize list ← {r}
while list ≠ ∅ do
    delete a node i from list
    examine(i, found)
    if found = true then return
```

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

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

**for all** $j \in \bar{A}(i)$ **do**
    **if** $j$ is even **then** contract($i, j$) and **return**
    **if** $j$ is unmatched **then**
        $q \leftarrow j$;
        $\text{pred}(q) \leftarrow i$;
        $found \leftarrow \text{true}$;
        **return**
    **if** $j$ is matched and unlabeled **then**
        $\text{pred}(j) \leftarrow i$;
        $\text{pred}(\text{mate}(j)) \leftarrow j$;
        add $\text{mate}(j)$ to $list$

Examine the neighbours of a node $i$

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

```
for all j ∈ Ā(i) do
    if j is even then contract(i, j) and return
    if j is unmatched then
        q ← j;
        pred(q) ← i;
        found ← true;
        return
    if j is matched and unlabeled then
        pred(j) ← i;
        pred(mate(j)) ← j;
        add mate(j) to list
```

For all neighbours $j$ do...

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

```
for all j ∈ Ā(i) do
    if j is even then contract(i, j) and return
    if j is unmatched then
        q ← j;
        pred(q) ← i;
        found ← true;
        return
    if j is matched and unlabeled then
        pred(j) ← i;
        pred(mate(j)) ← j;
        add mate(j) to list
```

You have found a blossom...

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

**for all** $j \in \bar{A}(i)$ **do**
    **if** $j$ is even **then** contract($i, j$) and **return**
    **if** $j$ is unmatched **then**
        $q \leftarrow j$;
        $\text{pred}(q) \leftarrow i$;
        $found \leftarrow \text{true}$;
        **return**
    **if** $j$ is matched and unlabeled **then**
        $\text{pred}(j) \leftarrow i$;
        $\text{pred}(\text{mate}(j)) \leftarrow j$;
        add mate($j$) to $list$

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

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

```
for all j ∈ Ā(i) do
    if j is even then contract(i, j) and return
    if j is unmatched then
        q ← j;
        pred(q) ← i;
        found ← true;
        return
    if j is matched and unlabeled then
        pred(j) ← i;
        pred(mate(j)) ← j;
        add mate(j) to list
```

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

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

```
for all j ∈ Ā(i) do
    if j is even then contract(i, j) and return
    if j is unmatched then
        q ← j;
        pred(q) ← i;
        found ← true;
        return
    if j is matched and unlabeled then
        pred(j) ← i;
        pred(mate(j)) ← j;
        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$(i, j)$

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

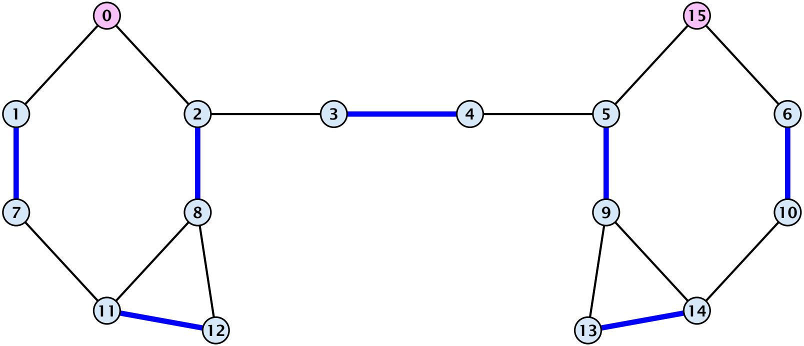
- A contraction operation can be performed in time $\mathcal{O}(m)$. Note, that any graph created will have at most $m$ edges.
- The time between two contraction-operation is basically a BFS/DFS on a graph. Hence takes time $\mathcal{O}(m)$.
- There are at most $n$ contractions as each contraction reduces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

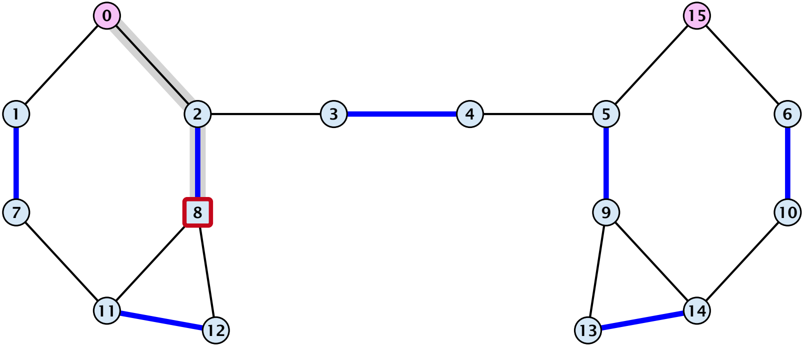
- A contraction operation can be performed in time $\mathcal{O}(m)$. Note, that any graph created will have at most $m$ edges.
- The time between two contraction-operation is basically a BFS/DFS on a graph. Hence takes time $\mathcal{O}(m)$.
- There are at most $n$ contractions as each contraction reduces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

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

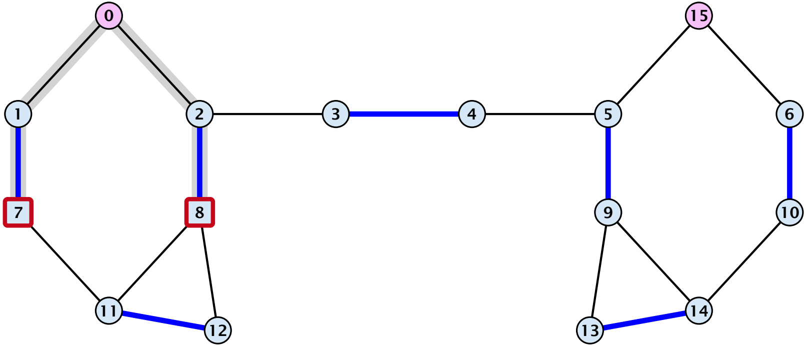
# Example: Blossom Algorithm



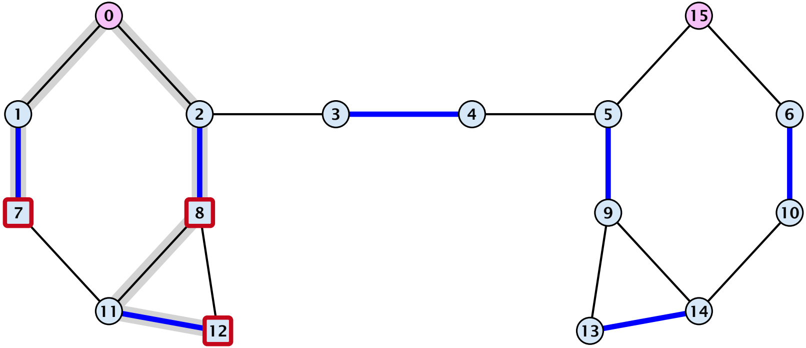
# Example: Blossom Algorithm



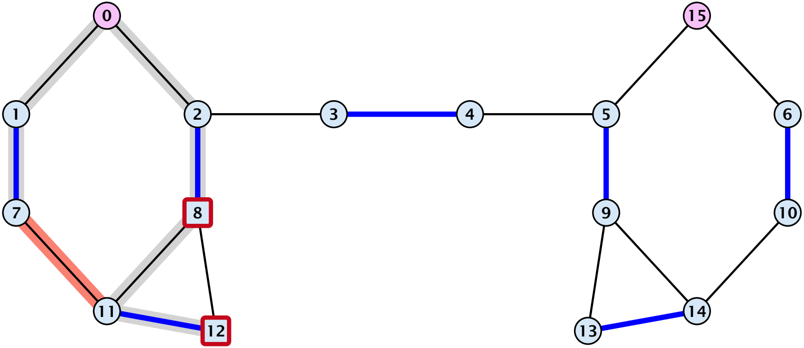
# Example: Blossom Algorithm



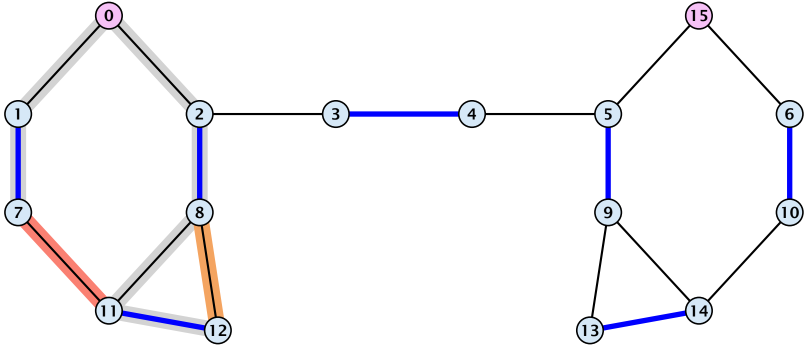
# Example: Blossom Algorithm



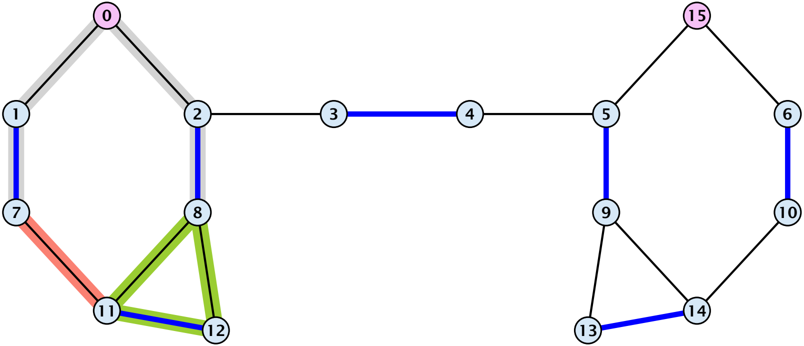
# Example: Blossom Algorithm



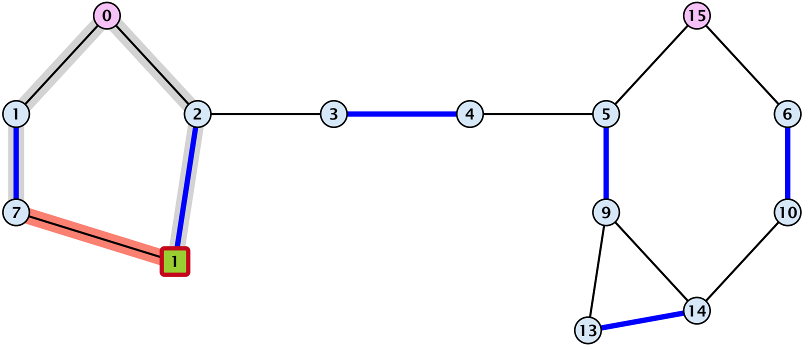
# Example: Blossom Algorithm



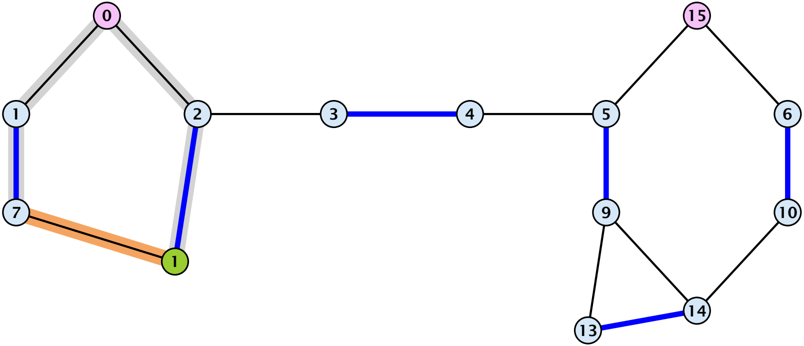
# Example: Blossom Algorithm



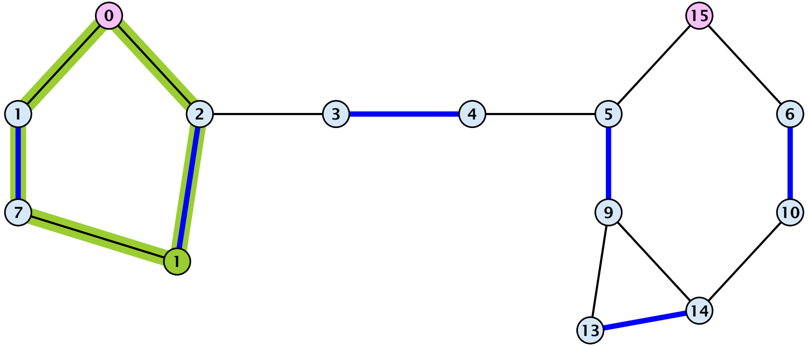
# Example: Blossom Algorithm



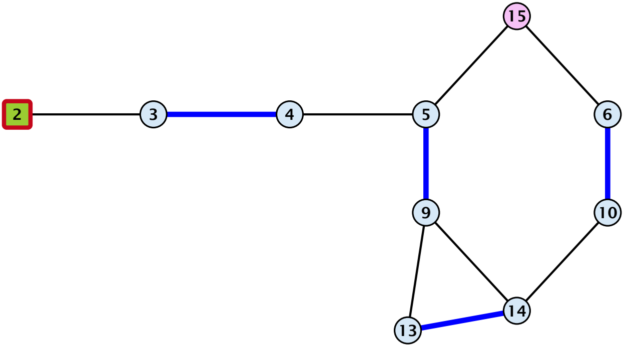
# Example: Blossom Algorithm



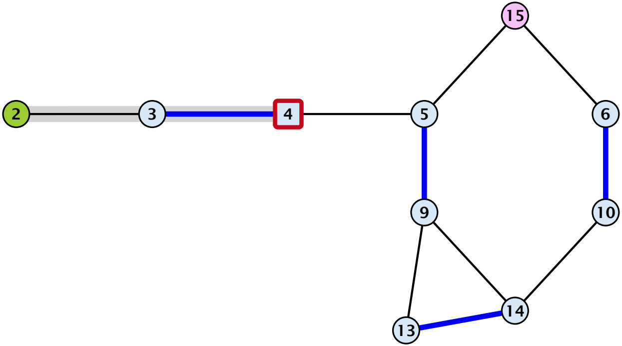
# Example: Blossom Algorithm



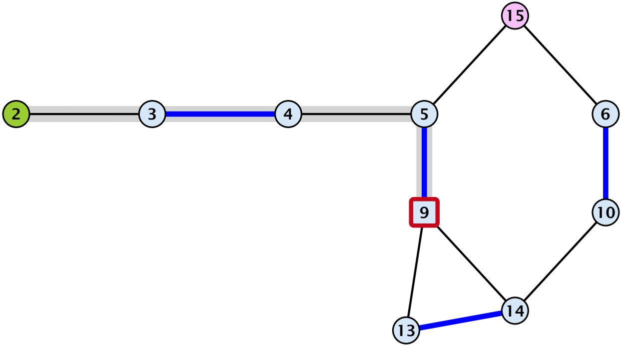
# Example: Blossom Algorithm



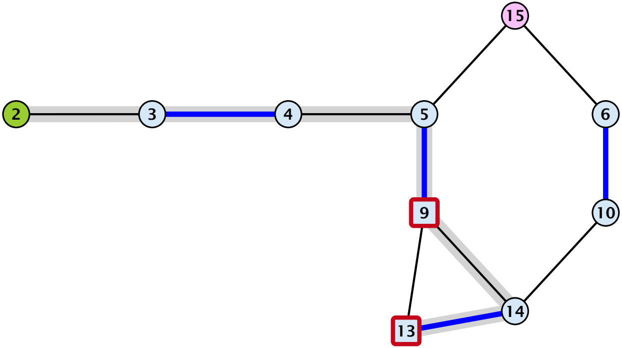
# Example: Blossom Algorithm



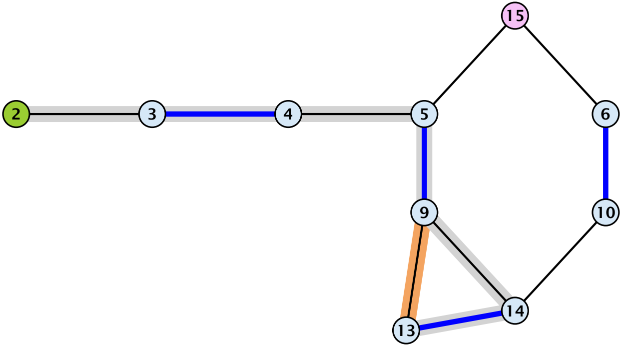
# Example: Blossom Algorithm



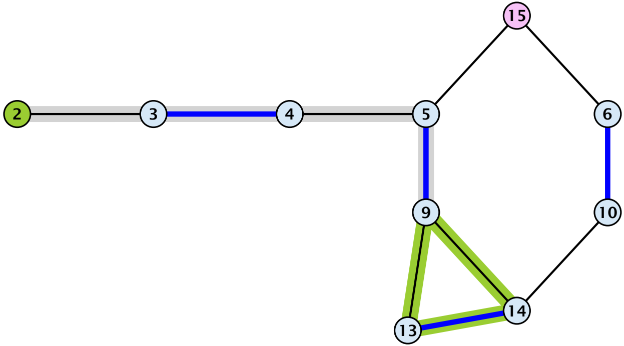
# Example: Blossom Algorithm



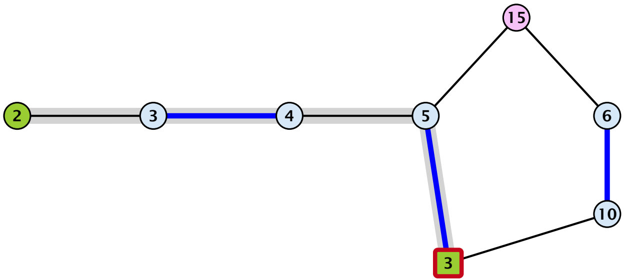
# Example: Blossom Algorithm



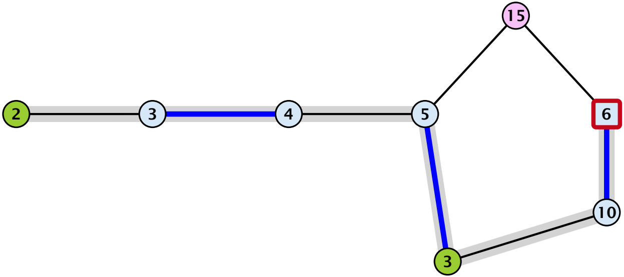
# Example: Blossom Algorithm



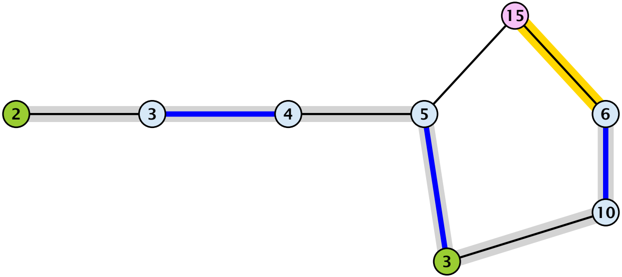
# Example: Blossom Algorithm



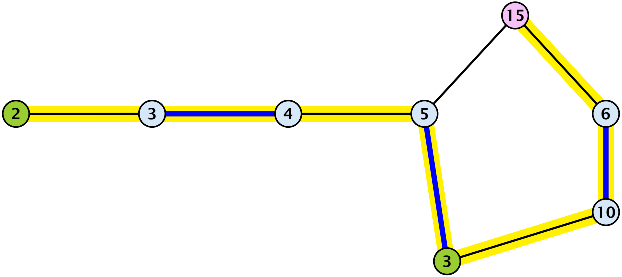
# Example: Blossom Algorithm



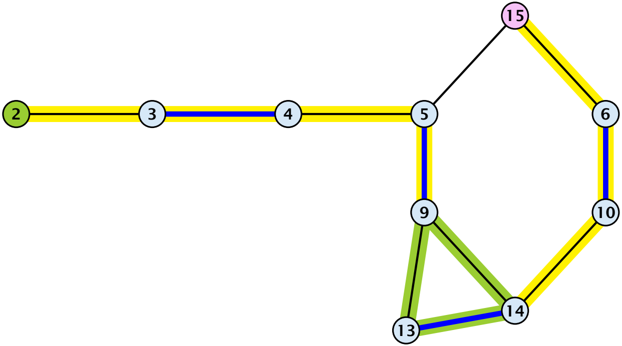
# Example: Blossom Algorithm



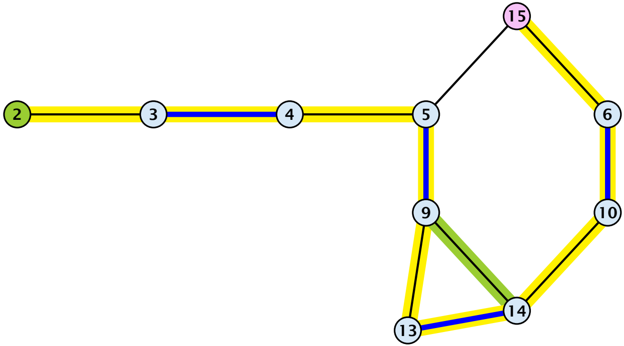
# Example: Blossom Algorithm



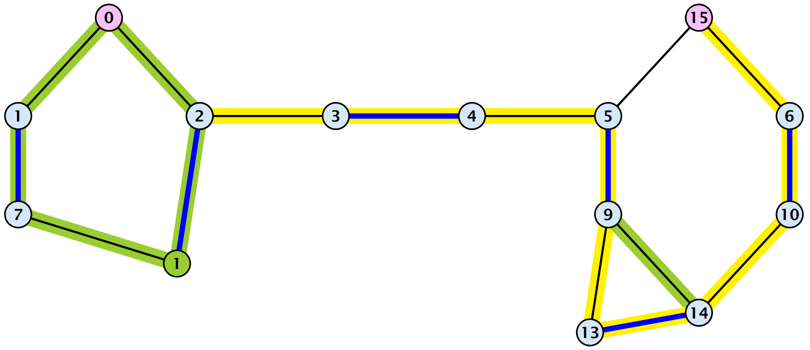
# Example: Blossom Algorithm



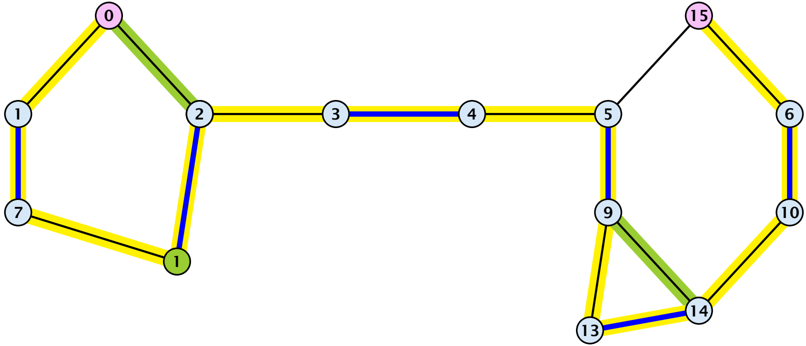
# Example: Blossom Algorithm



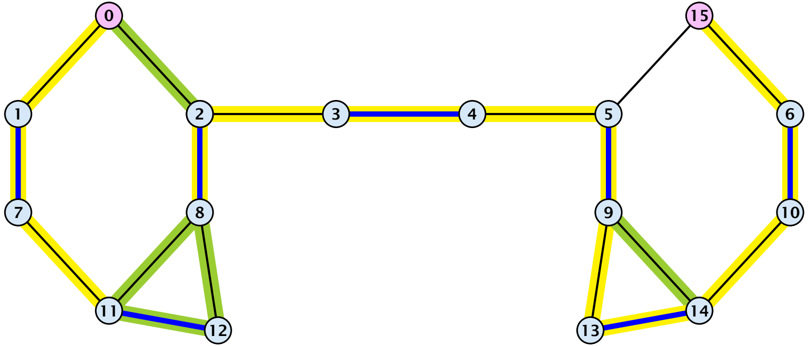
# Example: Blossom Algorithm



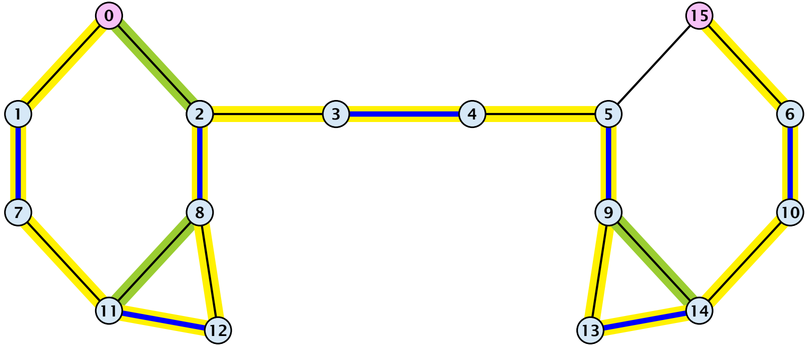
# Example: Blossom Algorithm



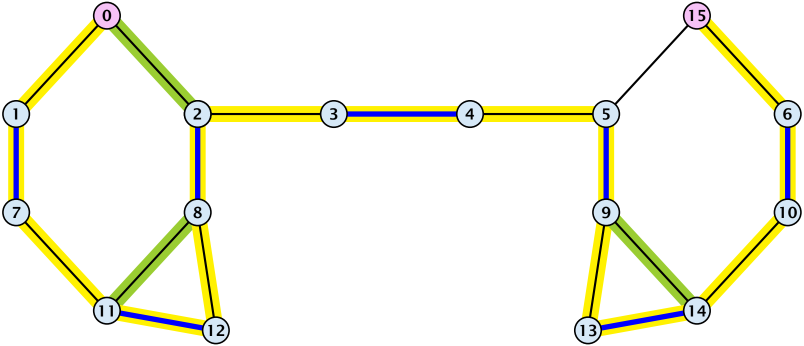
# Example: Blossom Algorithm



# Example: Blossom Algorithm



# Example: Blossom Algorithm



# A Fast Matching Algorithm

**Algorithm 53** Bimatch-Hopcroft-Karp($G$)

$M \leftarrow \emptyset$
**repeat**
    let $\mathcal{P} = \{P_1, \dots, P_k\}$ be maximal set of
    vertex-disjoint, shortest augmenting path w.r.t. $M$.
    $M \leftarrow M \oplus (P_1 \cup \cdots \cup P_k)$
**until** $\mathcal{P} = \emptyset$
**return** $M$

We call one iteration of the repeat-loop a phase of the algorithm.

# Analysis Hopcroft-Karp

**Lemma 12**

*Given a matching $M$ and a matching $M^*$ with $|M^*| - |M| \geq 0$. There exist $|M^*| - |M|$ vertex-disjoint augmenting path w.r.t. $M$.*

# Analysis Hopcroft-Karp

### Lemma 12

*Given a matching $M$ and a matching $M^*$ with $|M^*| - |M| \geq 0$. There exist $|M^*| - |M|$ vertex-disjoint augmenting path w.r.t. $M$.*

**Proof:**

- Similar to the proof that a matching is optimal iff it does not contain an augmenting path.

# Analysis Hopcroft-Karp

### Lemma 12

*Given a matching* $M$ *and a matching* $M^*$ *with* $|M^*| - |M| \geq 0$*.*
*There exist* $|M^*| - |M|$ *vertex-disjoint augmenting path w.r.t.* $M$*.*

**Proof:**

- Similar to the proof that a matching is optimal iff it does not contain an augmenting path.
- Consider the graph $G = (V, M \oplus M^*)$, and mark edges in this graph blue if they are in $M$ and red if they are in $M^*$.

# Analysis Hopcroft-Karp

### Lemma 12

*Given a matching* $M$ *and a matching* $M^*$ *with* $|M^*| - |M| \geq 0$*. There exist* $|M^*| - |M|$ *vertex-disjoint augmenting path w.r.t.* $M$*.*

**Proof:**

- Similar to the proof that a matching is optimal iff it does not contain an augmenting path.
- Consider the graph $G = (V, M \oplus M^*)$, and mark edges in this graph blue if they are in $M$ and red if they are in $M^*$.
- The connected components of $G$ are cycles and paths.

# Analysis Hopcroft-Karp

### Lemma 12

*Given a matching $M$ and a matching $M^*$ with $|M^*| - |M| \geq 0$. There exist $|M^*| - |M|$ vertex-disjoint augmenting path w.r.t. $M$.*

**Proof:**

- Similar to the proof that a matching is optimal iff it does not contain an augmenting path.
- Consider the graph $G = (V, M \oplus M^*)$, and mark edges in this graph blue if they are in $M$ and red if they are in $M^*$.
- The connected components of $G$ are cycles and paths.
- The graph contains $k \stackrel{\text{def}}{=} |M^*| - |M|$ more red edges than blue edges.

# Analysis Hopcroft-Karp

### Lemma 12

*Given a matching* $M$ *and a matching* $M^*$ *with* $|M^*| - |M| \geq 0$*. There exist* $|M^*| - |M|$ *vertex-disjoint augmenting path w.r.t.* $M$*.*

**Proof:**

- Similar to the proof that a matching is optimal iff it does not contain an augmenting path.
- Consider the graph $G = (V, M \oplus M^*)$, and mark edges in this graph blue if they are in $M$ and red if they are in $M^*$.
- The connected components of $G$ are cycles and paths.
- The graph contains $k \stackrel{\text{def}}{=} |M^*| - |M|$ more red edges than blue edges.
- Hence, there are at least $k$ components that form a path starting and ending with a red edge. These are augmenting paths w.r.t. $M$.

# Analysis Hopcroft-Karp

- Let $P_1, \dots, P_k$ be a maximal collection of vertex-disjoint, shortest augmenting paths w.r.t. $M$ (let $\ell = |P_i|$).

# Analysis Hopcroft-Karp

- Let $P_1, \dots, P_k$ be a maximal collection of vertex-disjoint, shortest augmenting paths w.r.t. $M$ (let $\ell = |P_i|$).
- $M' \stackrel{\text{def}}{=} M \oplus (P_1 \cup \cdots \cup P_k) = M \oplus P_1 \oplus \cdots \oplus P_k$.

# Analysis Hopcroft-Karp

- Let $P_1, \dots, P_k$ be a maximal collection of vertex-disjoint, shortest augmenting paths w.r.t. $M$ (let $\ell = |P_i|$).
- $M' \stackrel{\text{def}}{=} M \oplus (P_1 \cup \dots \cup P_k) = M \oplus P_1 \oplus \dots \oplus P_k$.
- Let $P$ be an augmenting path in $M'$.

# Analysis Hopcroft-Karp

- Let $P_1, \dots, P_k$ be a maximal collection of vertex-disjoint, shortest augmenting paths w.r.t. $M$ (let $\ell = |P_i|$).
- $M' \stackrel{\text{def}}{=} M \oplus (P_1 \cup \dots \cup P_k) = M \oplus P_1 \oplus \dots \oplus P_k$.
- Let $P$ be an augmenting path in $M'$.

**Lemma 13**

*The set* $A \stackrel{\text{def}}{=} M \oplus (M' \oplus P) = (P_1 \cup \dots \cup P_k) \oplus P$ *contains at least* $(k+1)\ell$ *edges.*

# Analysis Hopcroft-Karp

**Proof.**

- The set describes exactly the symmetric difference between matchings $M$ and $M' \oplus P$.

# Analysis Hopcroft-Karp

**Proof.**

- The set describes exactly the symmetric difference between matchings $M$ and $M' \oplus P$.
- Hence, the set contains at least $k+1$ vertex-disjoint augmenting paths w.r.t. $M$ as $|M'| = |M| + k + 1$.

# Analysis Hopcroft-Karp

**Proof.**

- The set describes exactly the symmetric difference between matchings $M$ and $M' \oplus P$.
- Hence, the set contains at least $k+1$ vertex-disjoint augmenting paths w.r.t. $M$ as $|M'| = |M| + k + 1$.
- Each of these paths is of length at least $\ell$.

# Analysis Hopcroft-Karp

**Lemma 14**

*$P$ is of length at least $\ell + 1$. This shows that the length of a shortest augmenting path increases between two phases of the Hopcroft-Karp algorithm.*

# Analysis Hopcroft-Karp

### Lemma 14

*$P$ is of length at least $\ell + 1$. This shows that the length of a shortest augmenting path increases between two phases of the Hopcroft-Karp algorithm.*

**Proof.**

- If $P$ does not intersect any of the $P_1, \dots, P_k$, this follows from the maximality of the set $\{P_1, \dots, P_k\}$.

# Analysis Hopcroft-Karp

### Lemma 14

*$P$ is of length at least $\ell + 1$. This shows that the length of a shortest augmenting path increases between two phases of the Hopcroft-Karp algorithm.*

**Proof.**

- If $P$ does not intersect any of the $P_1, \ldots, P_k$, this follows from the maximality of the set $\{P_1, \ldots, P_k\}$.
- Otherwise, at least one edge from $P$ coincides with an edge from paths $\{P_1, \ldots, P_k\}$.

# Analysis Hopcroft-Karp

### Lemma 14

*$P$ is of length at least $\ell + 1$. This shows that the length of a shortest augmenting path increases between two phases of the Hopcroft-Karp algorithm.*

**Proof.**

- If $P$ does not intersect any of the $P_1, \ldots, P_k$, this follows from the maximality of the set $\{P_1, \ldots, P_k\}$.
- Otherwise, at least one edge from $P$ coincides with an edge from paths $\{P_1, \ldots, P_k\}$.
- This edge is not contained in $A$.

# Analysis Hopcroft-Karp

### Lemma 14

*$P$ is of length at least $\ell + 1$. This shows that the length of a shortest augmenting path increases between two phases of the Hopcroft-Karp algorithm.*

**Proof.**

- If $P$ does not intersect any of the $P_1, \ldots, P_k$, this follows from the maximality of the set $\{P_1, \ldots, P_k\}$.
- Otherwise, at least one edge from $P$ coincides with an edge from paths $\{P_1, \ldots, P_k\}$.
- This edge is not contained in $A$.
- Hence, $|A| \leq k\ell + |P| - 1$.

# Analysis Hopcroft-Karp

### Lemma 14

*$P$ is of length at least $\ell + 1$. This shows that the length of a shortest augmenting path increases between two phases of the Hopcroft-Karp algorithm.*

**Proof.**

- If $P$ does not intersect any of the $P_1, \dots, P_k$, this follows from the maximality of the set $\{P_1, \dots, P_k\}$.
- Otherwise, at least one edge from $P$ coincides with an edge from paths $\{P_1, \dots, P_k\}$.
- This edge is not contained in $A$.
- Hence, $|A| \le k\ell + |P| - 1$.
- The lower bound on $|A|$ gives $(k+1)\ell \le |A| \le k\ell + |P| - 1$, and hence $|P| \ge \ell + 1$.

# Analysis Hopcroft-Karp

If the shortest augmenting path w.r.t. a matching $M$ has $\ell$ edges then the cardinality of the maximum matching is of size at most $|M| + \frac{|V|}{\ell+1}$.

# Analysis Hopcroft-Karp

If the shortest augmenting path w.r.t. a matching $M$ has $\ell$ edges then the cardinality of the maximum matching is of size at most $|M| + \frac{|V|}{\ell+1}$.

**Proof.**
The symmetric difference between $M$ and $M^*$ contains $|M^*| - |M|$ vertex-disjoint augmenting paths. Each of these paths contains at least $\ell + 1$ vertices. Hence, there can be at most $\frac{|V|}{\ell+1}$ of them.

# Analysis Hopcroft-Karp

**Lemma 15**
*The Hopcroft-Karp algorithm requires at most* $2\sqrt{|V|}$ *phases.*

# Analysis Hopcroft-Karp

### Lemma 15
*The Hopcroft-Karp algorithm requires at most* $2\sqrt{|V|}$ *phases.*

**Proof.**

- After iteration $\lfloor\sqrt{|V|}\rfloor$ the length of a shortest augmenting path must be at least $\lfloor\sqrt{|V|}\rfloor + 1 \geq \sqrt{|V|}$.
- Hence, there can be at most $|V|/(\sqrt{|V|} + 1) \leq \sqrt{|V|}$ additional augmentations.

# Analysis Hopcroft-Karp

**Lemma 16**
*One phase of the Hopcroft-Karp algorithm can be implemented in time $\mathcal{O}(m)$.*

construct a "level graph" $G'$:

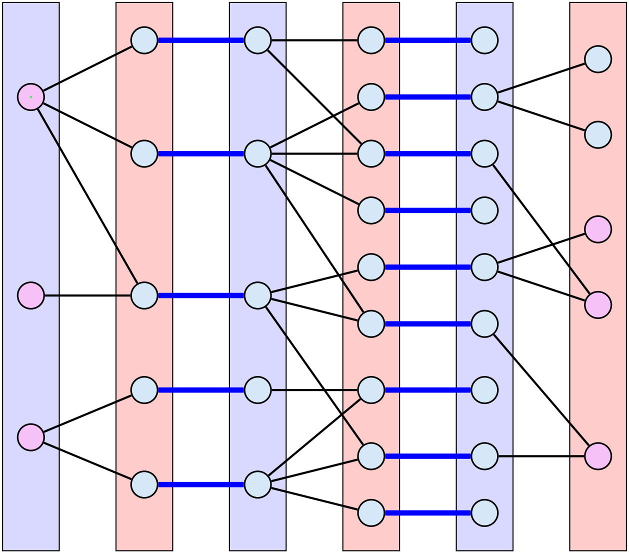
- construct Level 0 that includes all free vertices on left side $L$
- construct Level 1 containing all neighbors of Level 0
- construct Level 2 containing matching neighbors of Level 1
- construct Level 3 containing all neighbors of Level 2
- ...
- stop when a level (apart from Level 0) contains a free vertex

can be done in time $\mathcal{O}(m)$ by a modified BFS

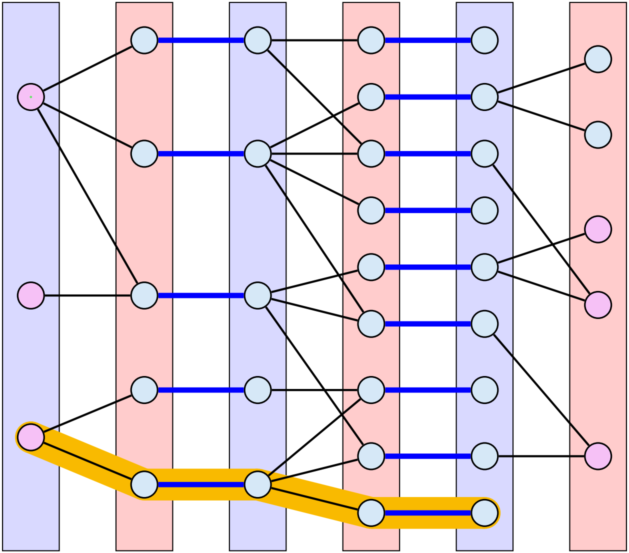
# Analysis Hopcroft-Karp

- a shortest augmenting path must go from Level 0 to the last layer constructed
- it can only use edges between layers
- construct a maxim**al** set of vertex disjoint augmenting path connecting the layers
- for this, go forward until you either reach a free vertex or you reach a “dead end” $v$
- if you reach a free vertex delete the augmenting path and all incident edges from the graph
- if you reach a dead end backtrack and delete $v$ together with its incident edges

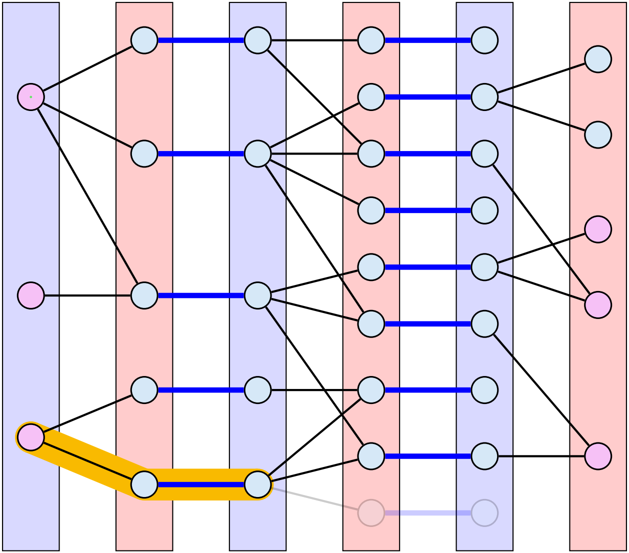
# Analysis Hopcroft-Karp



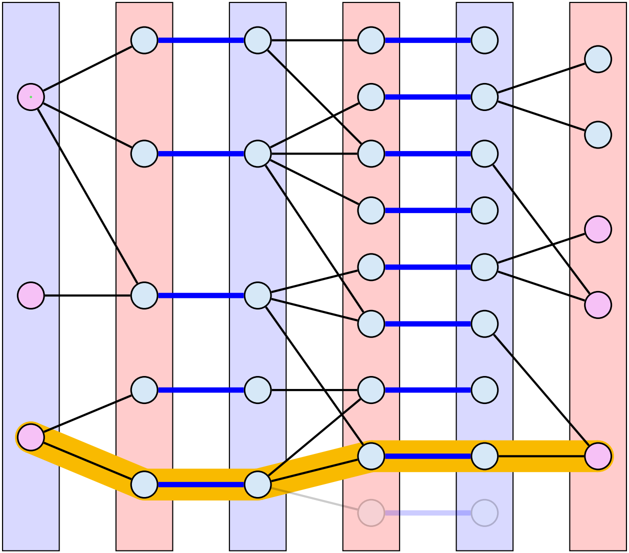
# Analysis Hopcroft-Karp



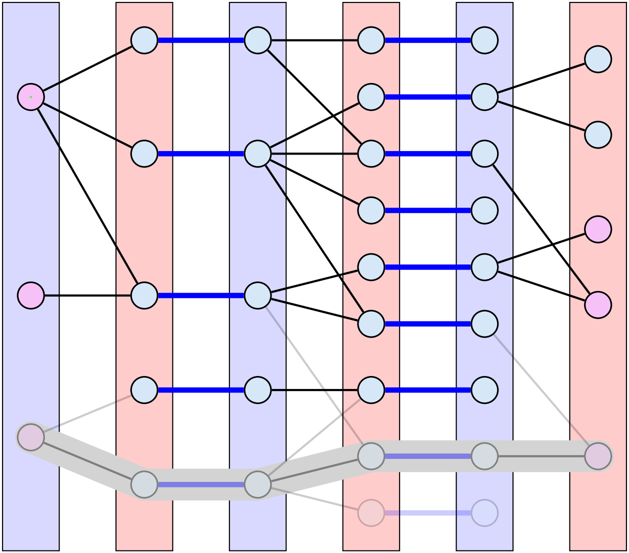
# Analysis Hopcroft-Karp



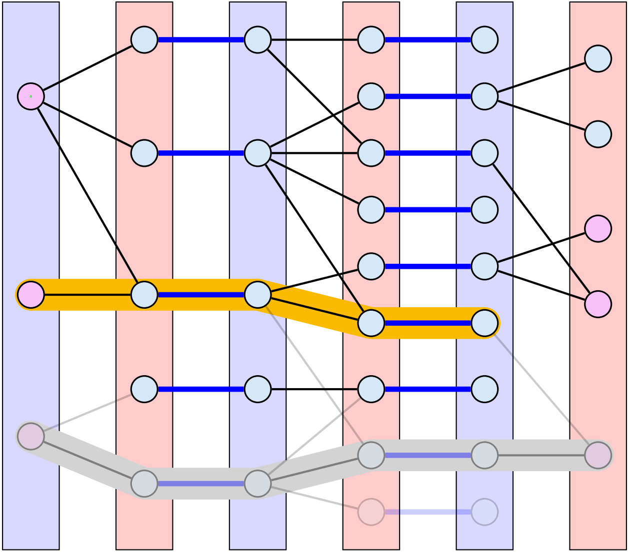
# Analysis Hopcroft-Karp



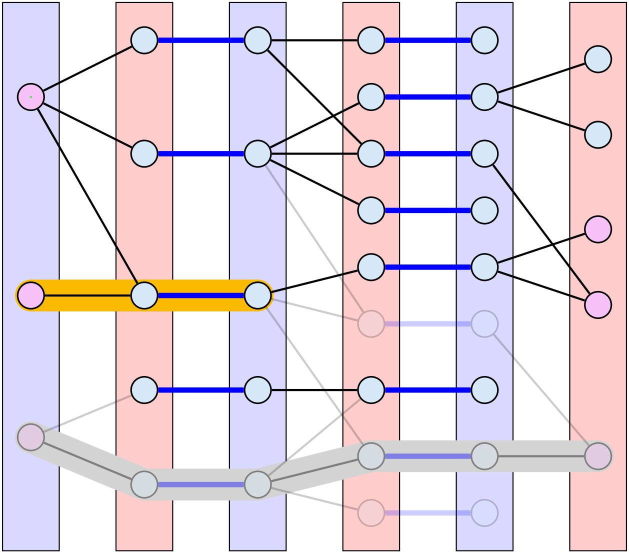
# Analysis Hopcroft-Karp



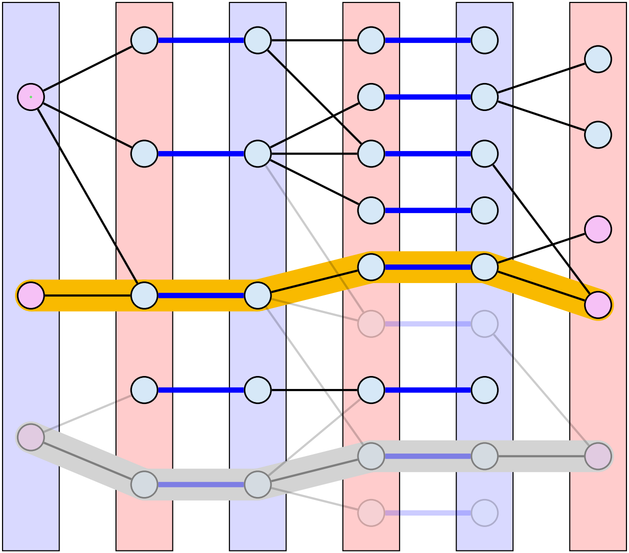
# Analysis Hopcroft-Karp



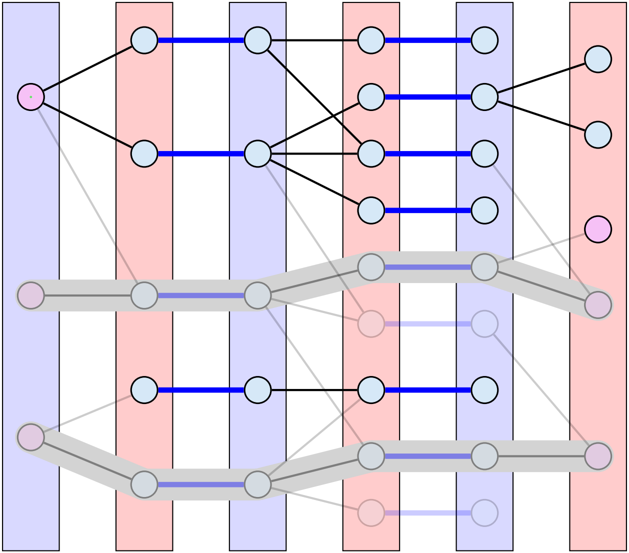
# Analysis Hopcroft-Karp



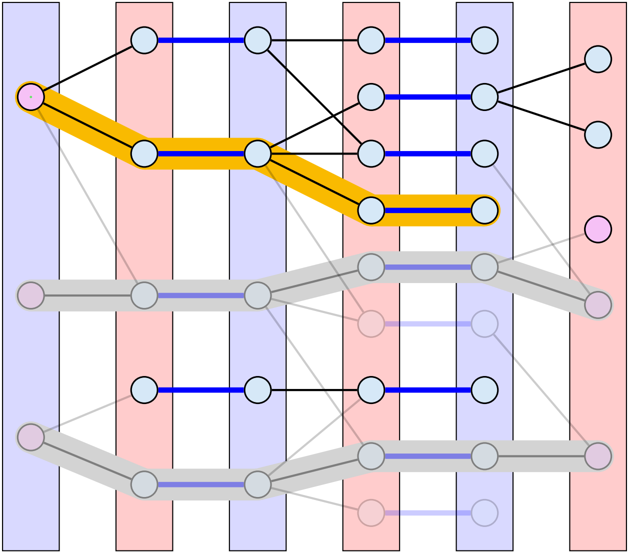
# Analysis Hopcroft-Karp



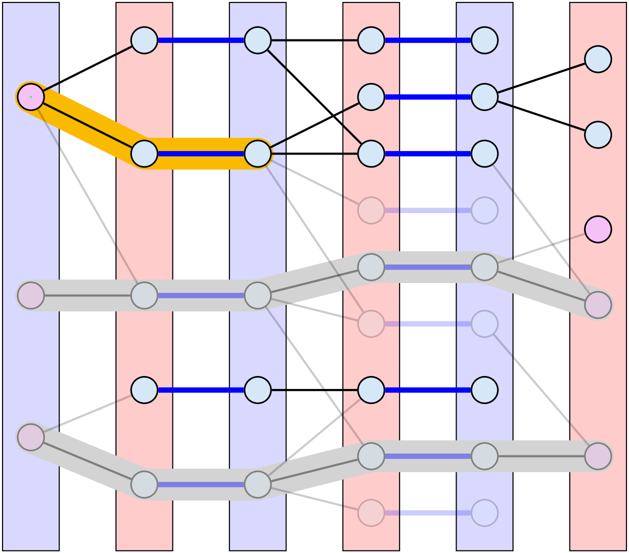
# Analysis Hopcroft-Karp



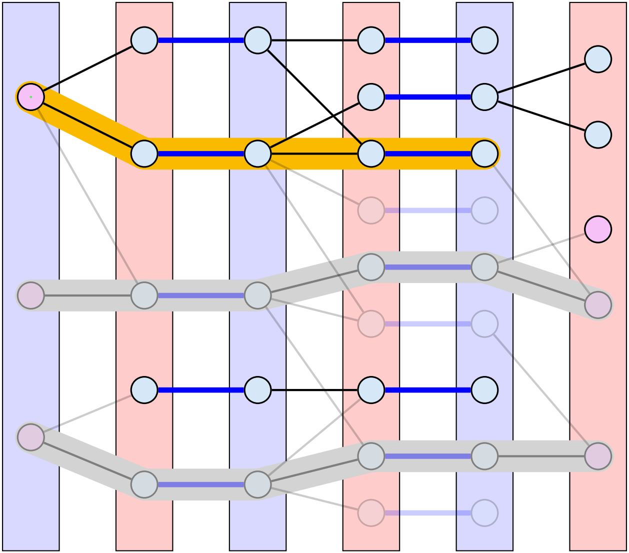
# Analysis Hopcroft-Karp



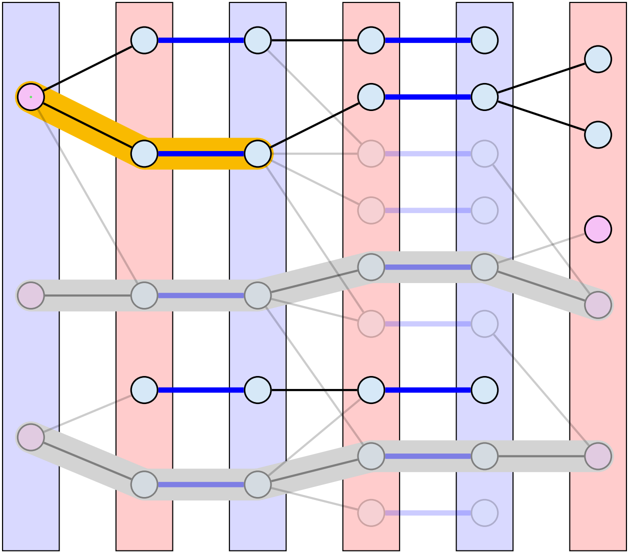
# Analysis Hopcroft-Karp



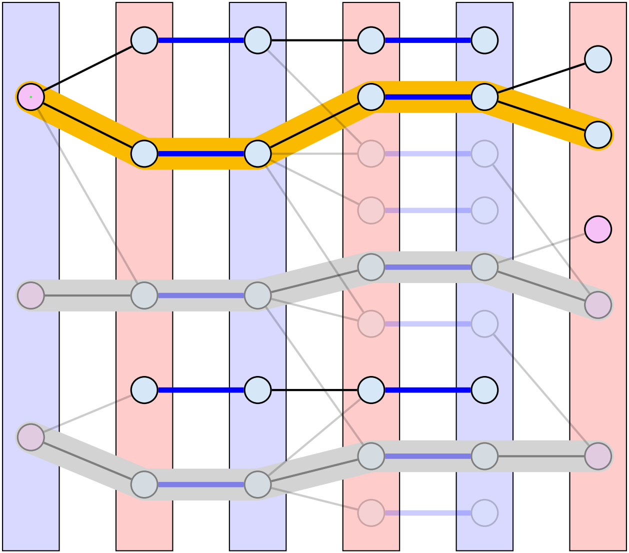
# Analysis Hopcroft-Karp



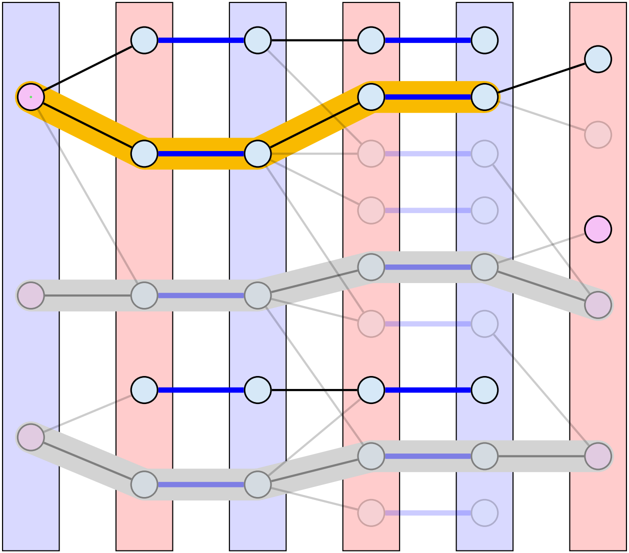
# Analysis Hopcroft-Karp



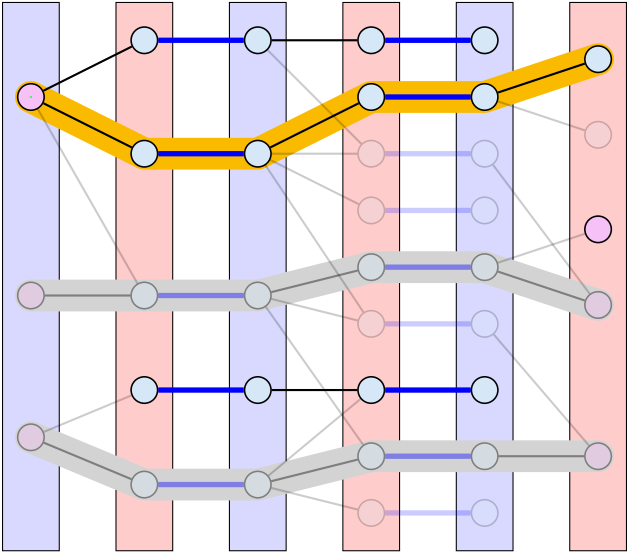
# Analysis Hopcroft-Karp



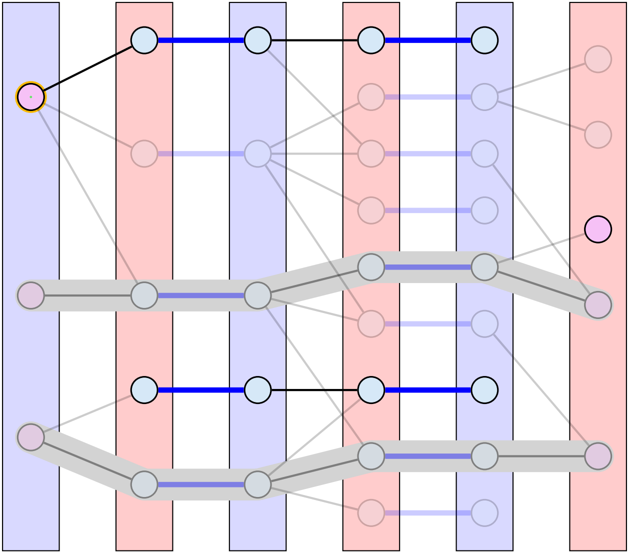
# Analysis Hopcroft-Karp



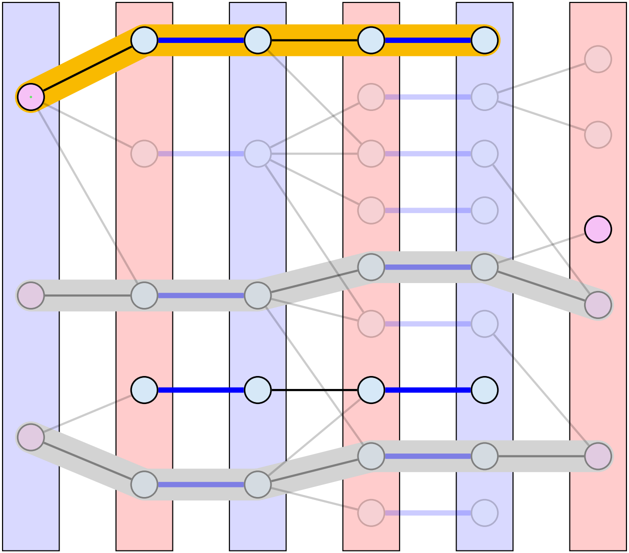
# Analysis Hopcroft-Karp



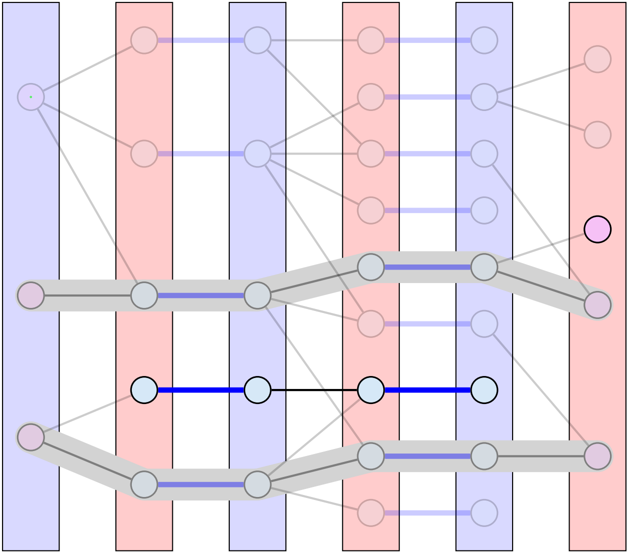
# Analysis Hopcroft-Karp



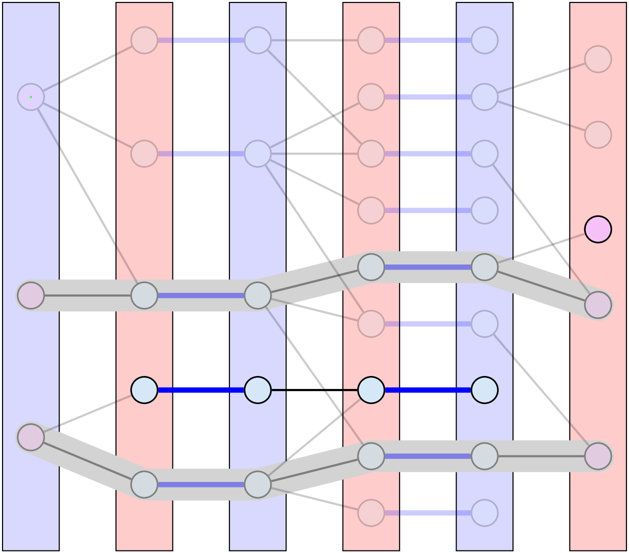
# Analysis Hopcroft-Karp



# Analysis Hopcroft-Karp



# Analysis Hopcroft-Karp



# Analysis: Shortest Augmenting Path for Flows

**cost for searches during a phase is $\mathcal{O}(mn)$**

- a search (successful or unsuccessful) takes time $\mathcal{O}(n)$
- a search deletes at least one edge from the level graph

**there are at most $n$ phases**

Time: $\mathcal{O}(mn^2)$.

# Analysis for Unit-capacity Simple Networks

**cost for searches during a phase is $\mathcal{O}(m)$**

- an edge/vertex is traversed at most twice

**need at most $\mathcal{O}(\sqrt{n})$ phases**

- after $\sqrt{n}$ phases there is a cut of size at most $\sqrt{n}$ in the residual graph
- hence at most $\sqrt{n}$ additional augmentations required

Time: $\mathcal{O}(m\sqrt{n})$.

# 21 Gomory Hu Trees

Given an undirected, weighted graph $G = (V, E, c)$ a cut-tree $T = (V, F, w)$ is a tree with edge-set $F$ and capacities $w$ that fulfills the following properties.

1. **Equivalent Flow Tree:** For any pair of vertices $s, t \in V$, $f(s, t)$ in $G$ is equal to $f_T(s, t)$.
2. **Cut Property:** A minimum $s$-$t$ cut in $T$ is also a minimum cut in $G$.

Here, $f(s, t)$ is the value of a maximum $s$-$t$ flow in $G$, and $f_T(s, t)$ is the corresponding value in $T$.

# Overview of the Algorithm

The algorithm maintains a partition of $V$, (sets $S_1, \ldots, S_t$), and a spanning tree $T$ on the vertex set $\{S_1, \ldots, S_t\}$.

# Overview of the Algorithm

The algorithm maintains a partition of $V$, (sets $S_1, \dots, S_t$), and a spanning tree $T$ on the vertex set $\{S_1, \dots, S_t\}$.

Initially, there exists only the set $S_1 = V$.

# Overview of the Algorithm

The algorithm maintains a partition of $V$, (sets $S_1, \dots, S_t$), and a spanning tree $T$ on the vertex set $\{S_1, \dots, S_t\}$.

Initially, there exists only the set $S_1 = V$.

Then the algorithm performs $n - 1$ split-operations:

# Overview of the Algorithm

The algorithm maintains a partition of $V$, (sets $S_1, \dots, S_t$), and a spanning tree $T$ on the vertex set $\{S_1, \dots, S_t\}$.

Initially, there exists only the set $S_1 = V$.

Then the algorithm performs $n - 1$ split-operations:

- In each such split-operation it chooses a set $S_i$ with $|S_i| \geq 2$ and splits this set into two non-empty parts $X$ and $Y$.

# Overview of the Algorithm

The algorithm maintains a partition of $V$, (sets $S_1, \ldots, S_t$), and a spanning tree $T$ on the vertex set $\{S_1, \ldots, S_t\}$.

Initially, there exists only the set $S_1 = V$.

Then the algorithm performs $n-1$ split-operations:

- In each such split-operation it chooses a set $S_i$ with $|S_i| \geq 2$ and splits this set into two non-empty parts $X$ and $Y$.
- $S_i$ is then removed from $T$ and replaced by $X$ and $Y$.

# Overview of the Algorithm

The algorithm maintains a partition of $V$, (sets $S_1, \dots, S_t$), and a spanning tree $T$ on the vertex set $\{S_1, \dots, S_t\}$.

Initially, there exists only the set $S_1 = V$.

Then the algorithm performs $n - 1$ split-operations:

- In each such split-operation it chooses a set $S_i$ with $|S_i| \geq 2$ and splits this set into two non-empty parts $X$ and $Y$.
- $S_i$ is then removed from $T$ and replaced by $X$ and $Y$.
- $X$ and $Y$ are connected by an edge, and the edges that before the split were incident to $S_i$ are attached to either $X$ or $Y$.

# Overview of the Algorithm

The algorithm maintains a partition of $V$, (sets $S_1, \ldots, S_t$), and a spanning tree $T$ on the vertex set $\{S_1, \ldots, S_t\}$.

Initially, there exists only the set $S_1 = V$.

Then the algorithm performs $n - 1$ split-operations:

- In each such split-operation it chooses a set $S_i$ with $|S_i| \geq 2$ and splits this set into two non-empty parts $X$ and $Y$.
- $S_i$ is then removed from $T$ and replaced by $X$ and $Y$.
- $X$ and $Y$ are connected by an edge, and the edges that before the split were incident to $S_i$ are attached to either $X$ or $Y$.

**In the end this gives a tree on the vertex set $V$.**

# Details of the Split-operation

- Select $S_i$ that contains at least two nodes $a$ and $b$.

# Details of the Split-operation

- Select $S_i$ that contains at least two nodes $a$ and $b$.
- Compute the connected components of the forest obtained from the current tree $T$ after deleting $S_i$. Each of these components corresponds to a set of vertices from $V$.

# Details of the Split-operation

- Select $S_i$ that contains at least two nodes $a$ and $b$.
- Compute the connected components of the forest obtained from the current tree $T$ after deleting $S_i$. Each of these components corresponds to a set of vertices from $V$.
- Consider the graph $H$ obtained from $G$ by contracting these connected components into single nodes.

# Details of the Split-operation

- Select $S_i$ that contains at least two nodes $a$ and $b$.
- Compute the connected components of the forest obtained from the current tree $T$ after deleting $S_i$. Each of these components corresponds to a set of vertices from $V$.
- Consider the graph $H$ obtained from $G$ by contracting these connected components into single nodes.
- Compute a minimum $a$-$b$ cut in $H$. Let $A$, and $B$ denote the two sides of this cut.

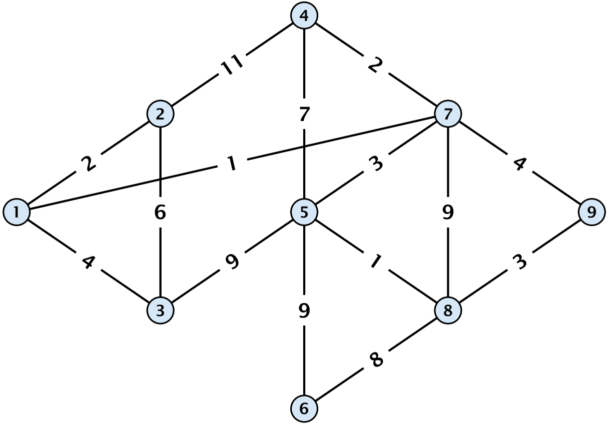
# Details of the Split-operation

- Select $S_i$ that contains at least two nodes $a$ and $b$.
- Compute the connected components of the forest obtained from the current tree $T$ after deleting $S_i$. Each of these components corresponds to a set of vertices from $V$.
- Consider the graph $H$ obtained from $G$ by contracting these connected components into single nodes.
- Compute a minimum $a$-$b$ cut in $H$. Let $A$, and $B$ denote the two sides of this cut.
- Split $S_i$ in $T$ into two sets/nodes $S_i^a := S_i \cap A$ and $S_i^b := S_i \cap B$ and add edge $\{S_i^a, S_i^b\}$ with capacity $f_H(a,b)$.

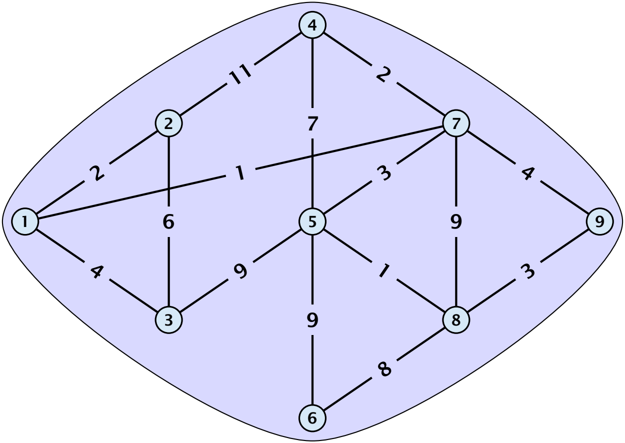
# Details of the Split-operation

- Select $S_i$ that contains at least two nodes $a$ and $b$.
- Compute the connected components of the forest obtained from the current tree $T$ after deleting $S_i$. Each of these components corresponds to a set of vertices from $V$.
- Consider the graph $H$ obtained from $G$ by contracting these connected components into single nodes.
- Compute a minimum $a$-$b$ cut in $H$. Let $A$, and $B$ denote the two sides of this cut.
- Split $S_i$ in $T$ into two sets/nodes $S_i^a \coloneqq S_i \cap A$ and $S_i^b \coloneqq S_i \cap B$ and add edge $\{S_i^a, S_i^b\}$ with capacity $f_H(a,b)$.
- Replace an edge $\{S_i, S_x\}$ by $\{S_i^a, S_x\}$ if $S_x \subset A$ and by $\{S_i^b, S_x\}$ if $S_x \subset B$.

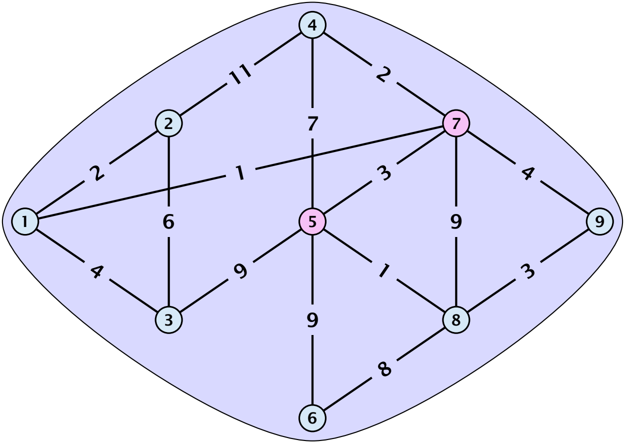
# Example: Gomory-Hu Construction



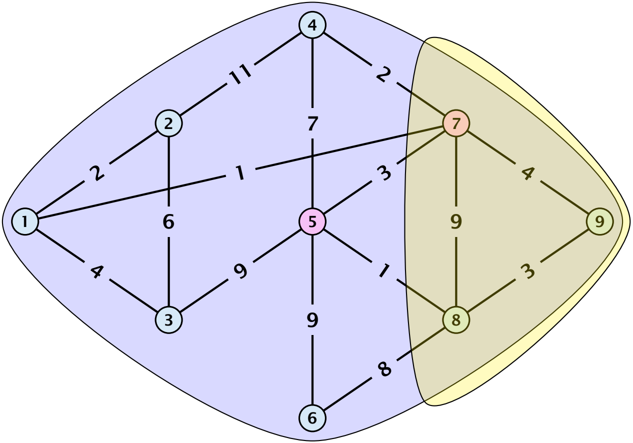
# Example: Gomory-Hu Construction



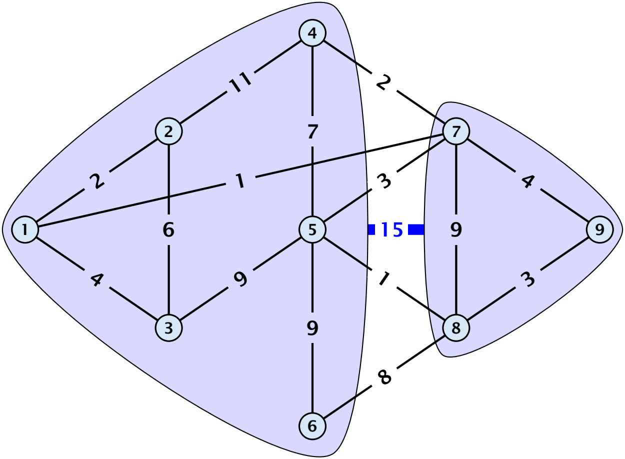
# Example: Gomory-Hu Construction



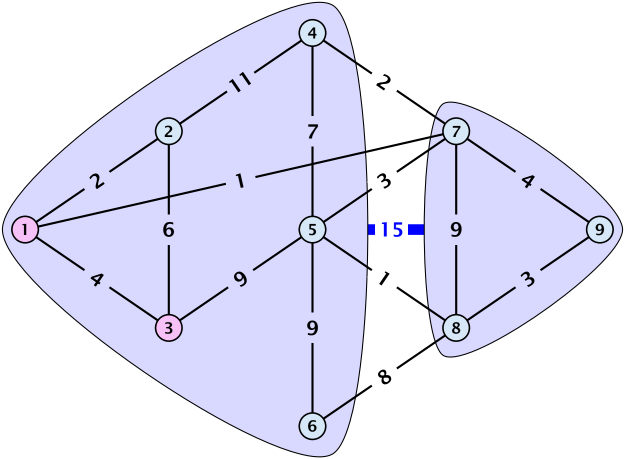
# Example: Gomory-Hu Construction



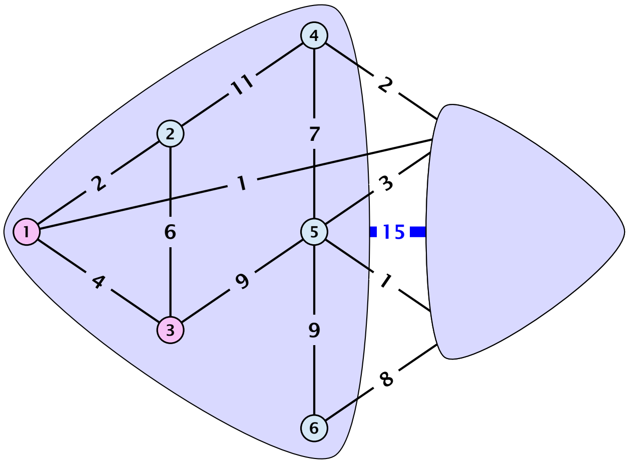
# Example: Gomory-Hu Construction



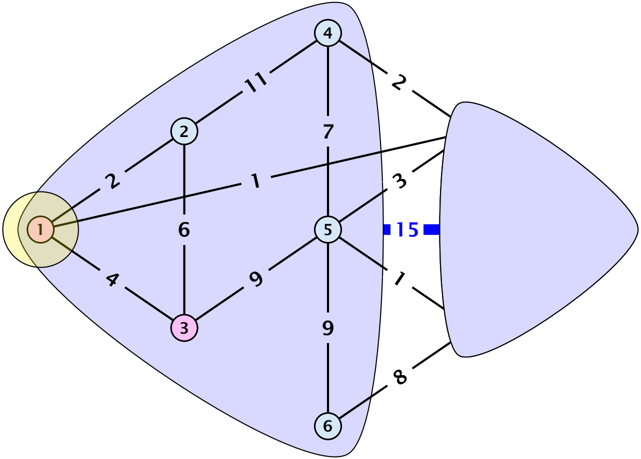
# Example: Gomory-Hu Construction



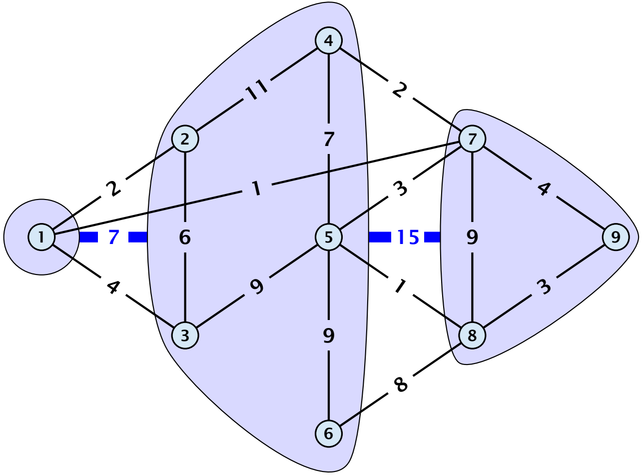
# Example: Gomory-Hu Construction



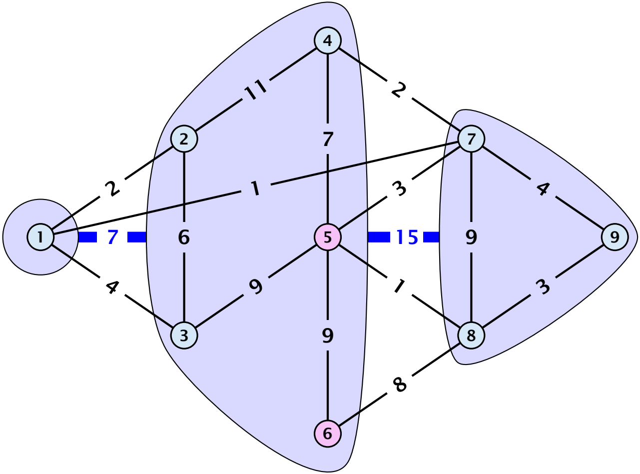
# Example: Gomory-Hu Construction



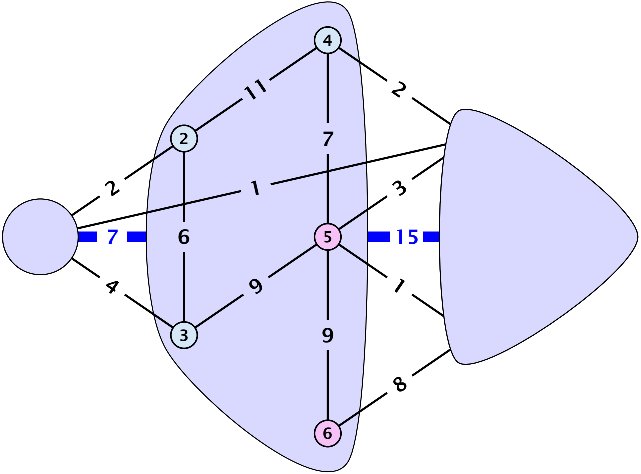
# Example: Gomory-Hu Construction



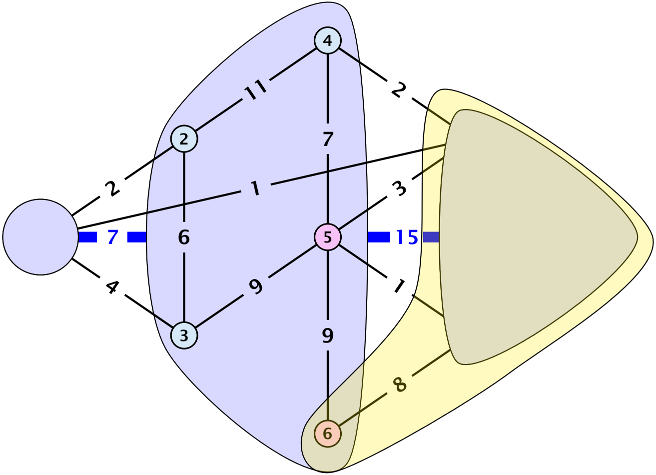
# Example: Gomory-Hu Construction



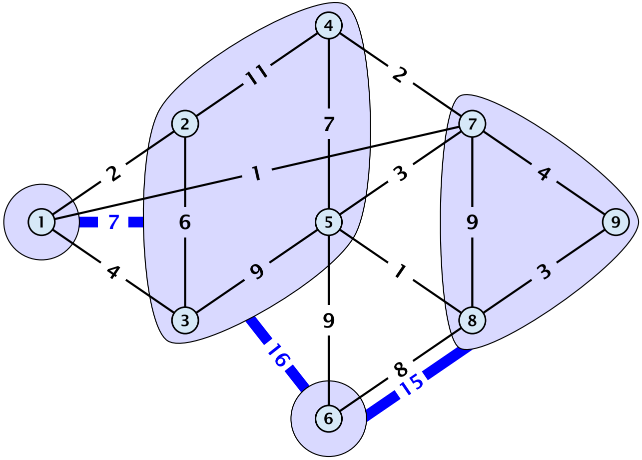
# Example: Gomory-Hu Construction



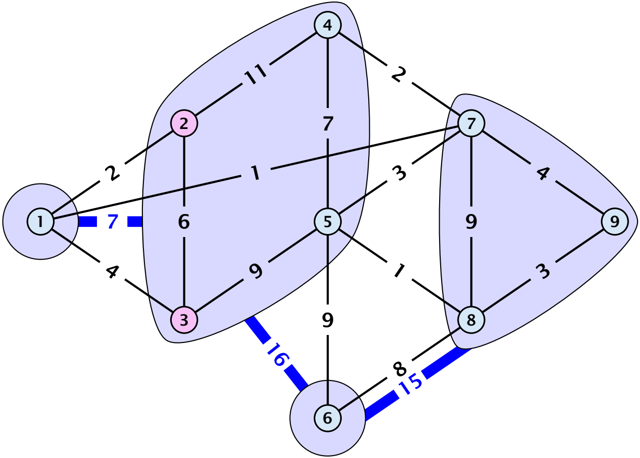
# Example: Gomory-Hu Construction



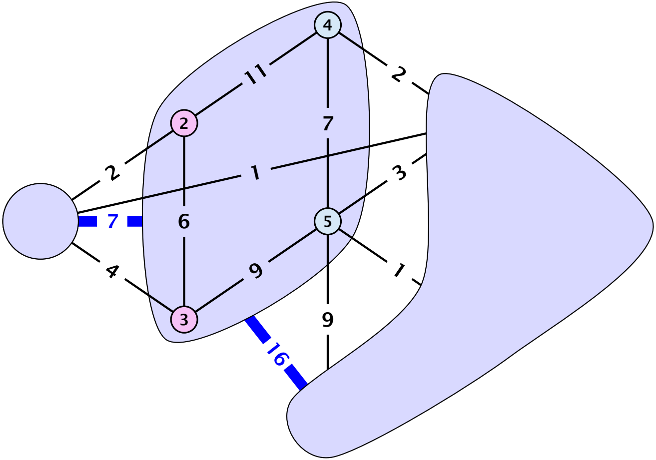
# Example: Gomory-Hu Construction



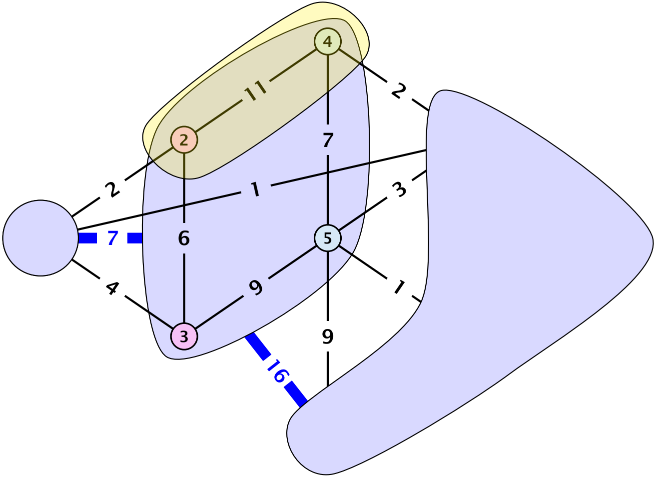
# Example: Gomory-Hu Construction



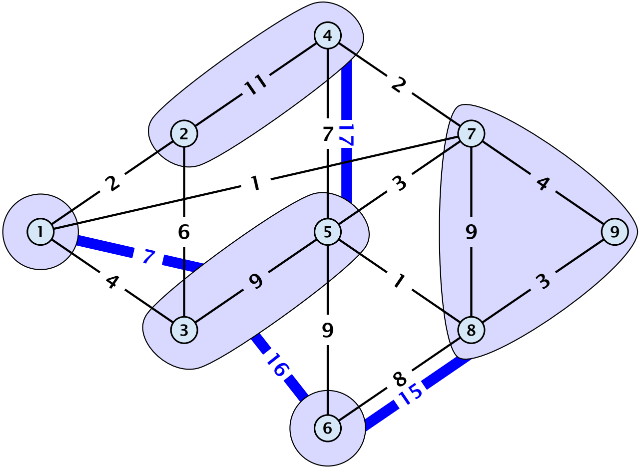
# Example: Gomory-Hu Construction



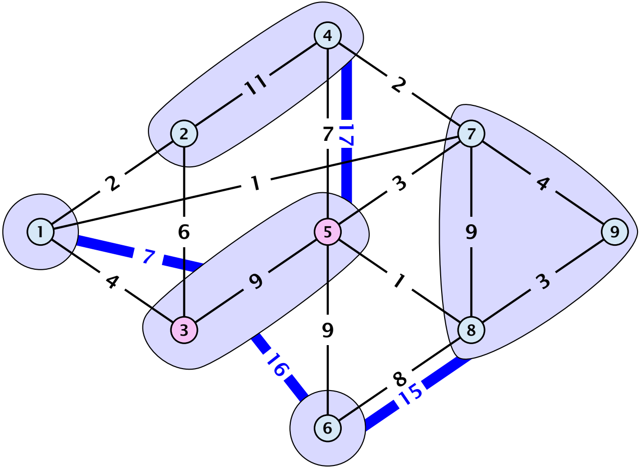
# Example: Gomory-Hu Construction



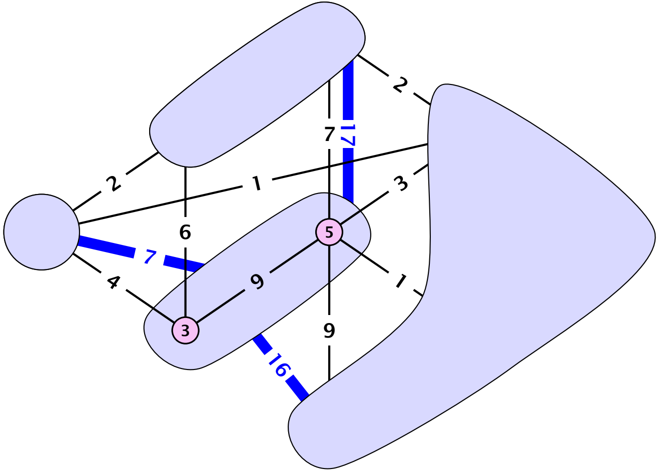
# Example: Gomory-Hu Construction



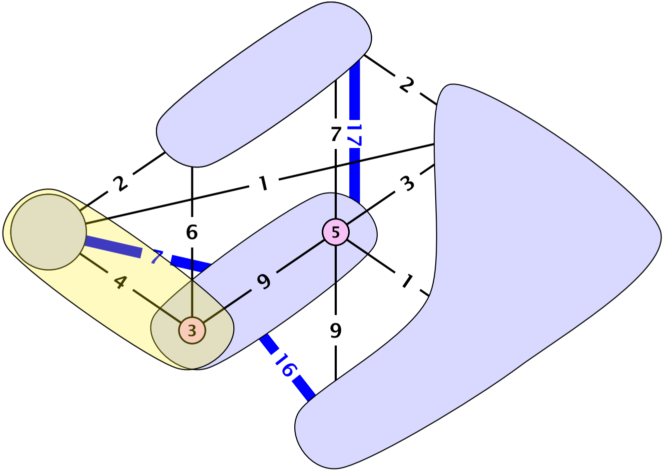
# Example: Gomory-Hu Construction



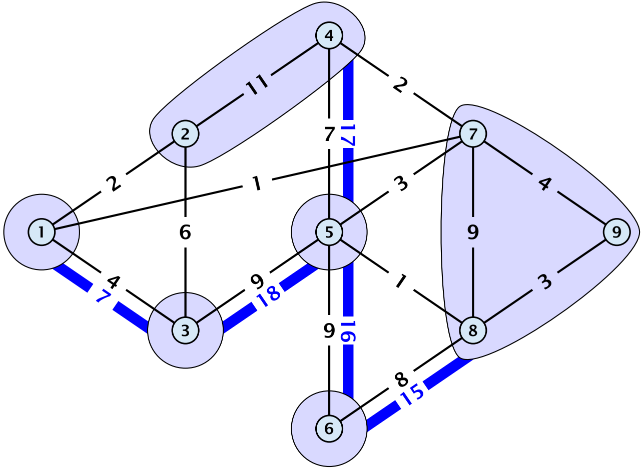
# Example: Gomory-Hu Construction



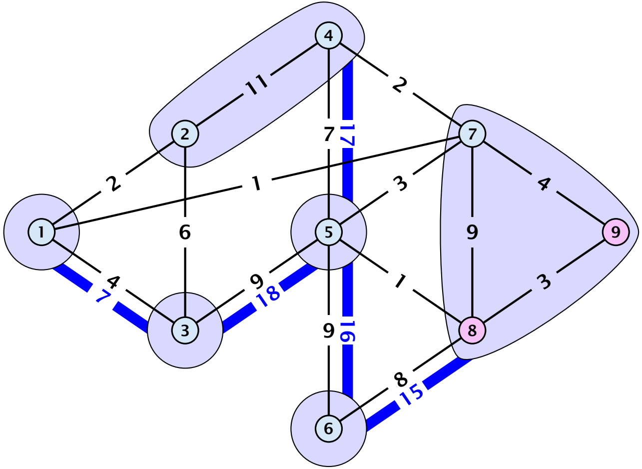
# Example: Gomory-Hu Construction



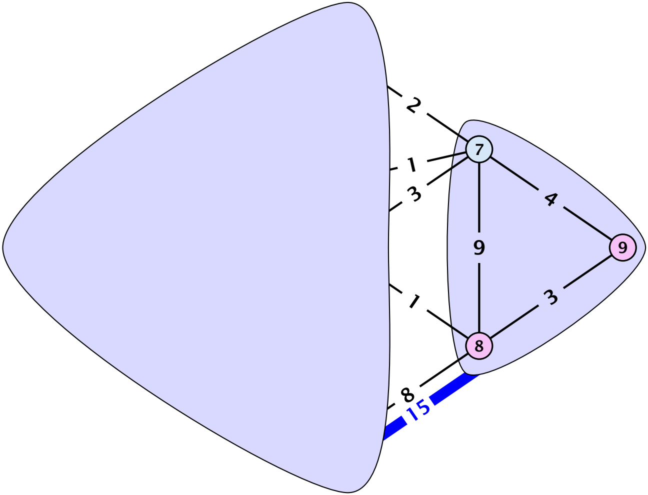
# Example: Gomory-Hu Construction



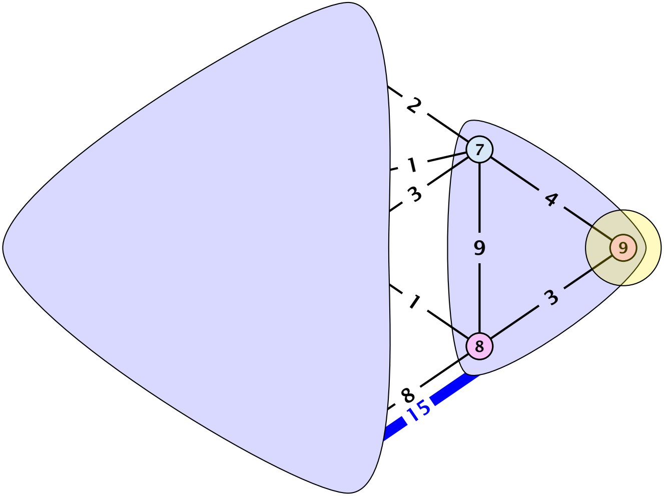
# Example: Gomory-Hu Construction



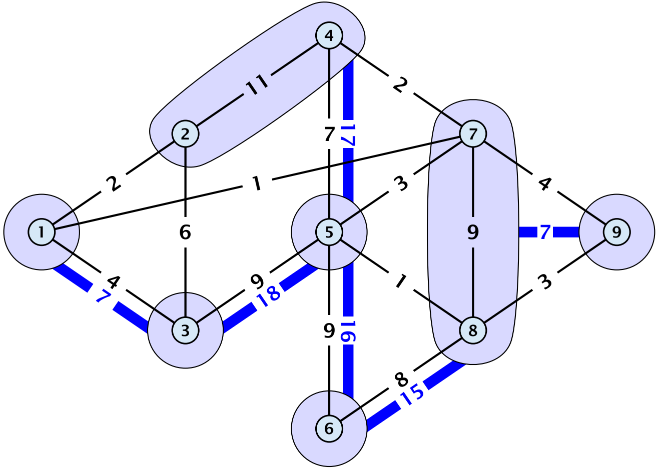
# Example: Gomory-Hu Construction



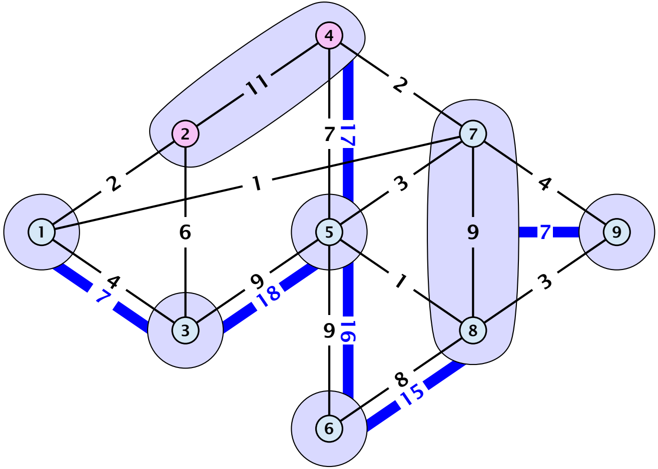
# Example: Gomory-Hu Construction



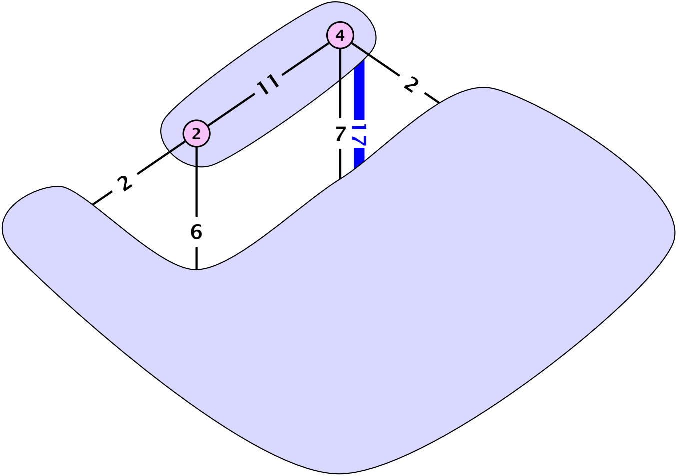
# Example: Gomory-Hu Construction



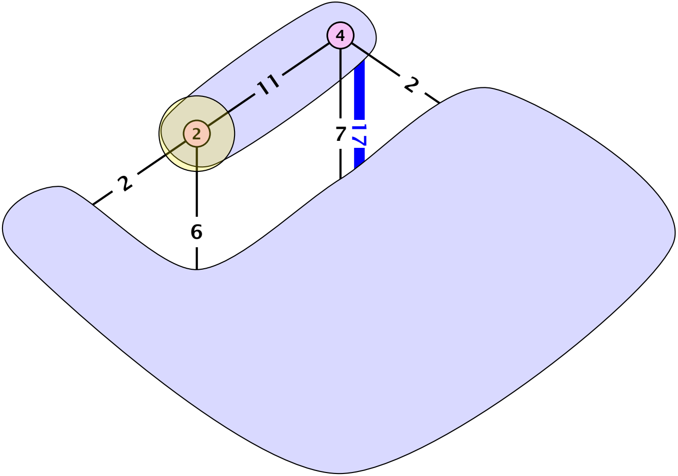
# Example: Gomory-Hu Construction



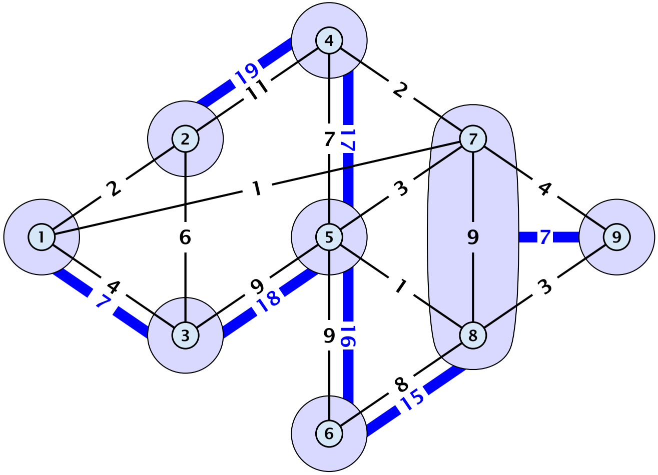
# Example: Gomory-Hu Construction



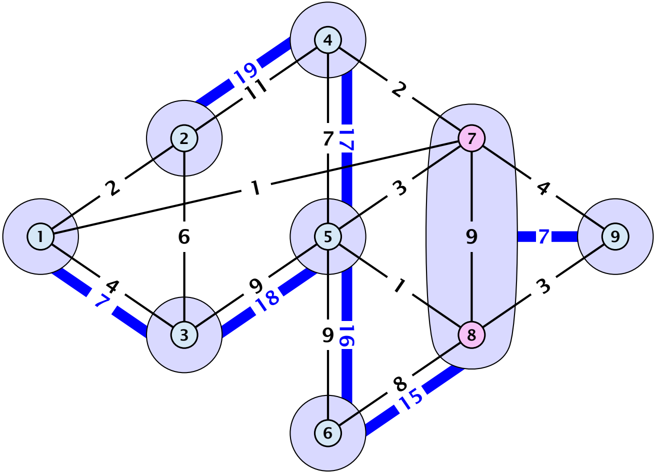
# Example: Gomory-Hu Construction



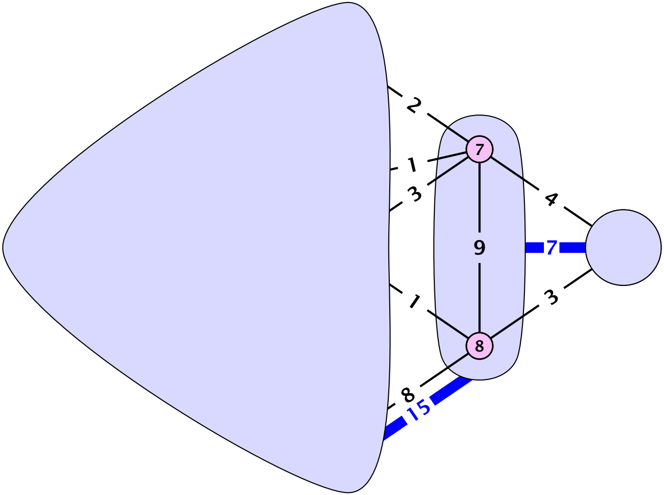
# Example: Gomory-Hu Construction



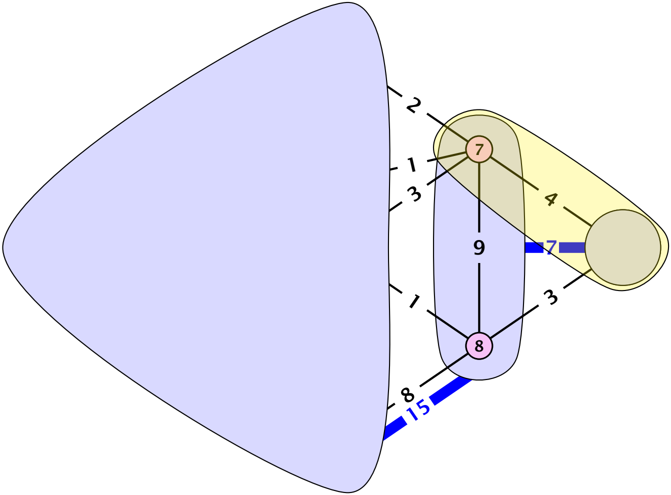
# Example: Gomory-Hu Construction



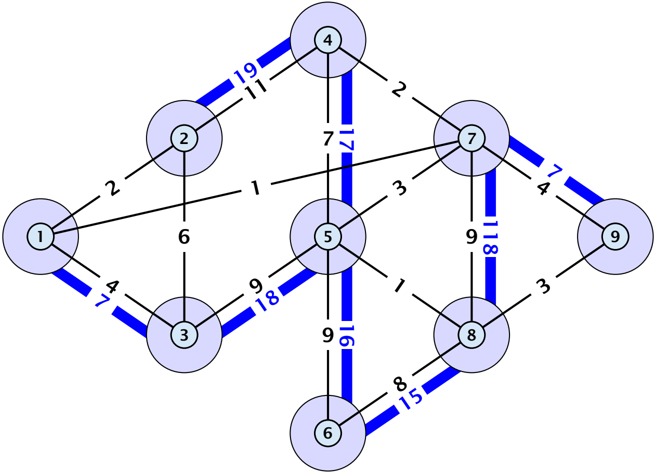
# Example: Gomory-Hu Construction



# Example: Gomory-Hu Construction



# Example: Gomory-Hu Construction



# Analysis

**Lemma 17**
*For nodes* $s, t, x \in V$ *we have* $f(s,t) \geq \min\{f(s,x), f(x,t)\}$

# Analysis

**Lemma 17**
*For nodes* $s, t, x \in V$ *we have* $f(s,t) \geq \min\{f(s,x), f(x,t)\}$

**Lemma 18**
*For nodes* $s, t, x_1, \ldots, x_k \in V$ *we have*
$f(s,t) \geq \min\{f(s,x_1), f(x_1,x_2), \ldots, f(x_{k-1},x_k), f(x_k,t)\}$

**Lemma 19**

*Let $S$ be some minimum $r$-$s$ cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum $v$-$w$-cut $T$ with $T \subset S$.*

**Lemma 19**

*Let $S$ be some minimum $r$-$s$ cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum $v$-$w$-cut $T$ with $T \subset S$.*

**Proof:** Let $X$ be a minimum $v$-$w$ cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$.

## Lemma 19

*Let $S$ be some minimum $r$-$s$ cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum $v$-$w$-cut $T$ with $T \subset S$.*

**Proof:** Let $X$ be a minimum $v$-$w$ cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are $v$-$w$ cuts inside $S$.

**Lemma 19**

*Let $S$ be some minimum $r$-$s$ cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum $v$-$w$-cut $T$ with $T \subset S$.*

**Proof:** Let $X$ be a minimum $v$-$w$ cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are $v$-$w$ cuts inside $S$.

We may assume w.l.o.g. $s \in X$.

## Lemma 19

*Let $S$ be some minimum $r$-$s$ cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum $v$-$w$-cut $T$ with $T \subset S$.*

**Proof:** Let $X$ be a minimum $v$-$w$ cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are $v$-$w$ cuts inside $S$.

We may assume w.l.o.g. $s \in X$.

**First case $r \in X$.**

### Lemma 19

*Let $S$ be some minimum $r$-$s$ cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum $v$-$w$-cut $T$ with $T \subset S$.*

**Proof:** Let $X$ be a minimum $v$-$w$ cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are $v$-$w$ cuts inside $S$.

We may assume w.l.o.g. $s \in X$.

**First case $r \in X$.**

- $\text{cap}(X \setminus S) + \text{cap}(S \setminus X) \leq \text{cap}(S) + \text{cap}(X)$.

## Lemma 19

*Let $S$ be some minimum $r$-$s$ cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum $v$-$w$-cut $T$ with $T \subset S$.*

**Proof:** Let $X$ be a minimum $v$-$w$ cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are $v$-$w$ cuts inside $S$.

We may assume w.l.o.g. $s \in X$.

**First case $r \in X$.**

- $\mathrm{cap}(X \setminus S) + \mathrm{cap}(S \setminus X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$.
- $\mathrm{cap}(X \setminus S) \geq \mathrm{cap}(S)$ because $X \setminus S$ is an $r$-$s$ cut.

**Lemma 19**

*Let $S$ be some minimum $r$-$s$ cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum $v$-$w$-cut $T$ with $T \subset S$.*

**Proof:** Let $X$ be a minimum $v$-$w$ cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are $v$-$w$ cuts inside $S$.

We may assume w.l.o.g. $s \in X$.

**First case $r \in X$.**

- $\mathrm{cap}(X \setminus S) + \mathrm{cap}(S \setminus X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$.
- $\mathrm{cap}(X \setminus S) \geq \mathrm{cap}(S)$ because $X \setminus S$ is an $r$-$s$ cut.
- This gives $\mathrm{cap}(S \setminus X) \leq \mathrm{cap}(X)$.

**Lemma 19**

*Let $S$ be some minimum $r$-$s$ cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum $v$-$w$-cut $T$ with $T \subset S$.*

**Proof:** Let $X$ be a minimum $v$-$w$ cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are $v$-$w$ cuts inside $S$.

We may assume w.l.o.g. $s \in X$.

**First case $r \in X$.**

- $\mathrm{cap}(X \setminus S) + \mathrm{cap}(S \setminus X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$.
- $\mathrm{cap}(X \setminus S) \geq \mathrm{cap}(S)$ because $X \setminus S$ is an $r$-$s$ cut.
- This gives $\mathrm{cap}(S \setminus X) \leq \mathrm{cap}(X)$.

**Second case $r \notin X$.**

### Lemma 19

*Let $S$ be some minimum $r$-$s$ cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum $v$-$w$-cut $T$ with $T \subset S$.*

**Proof:** Let $X$ be a minimum $v$-$w$ cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are $v$-$w$ cuts inside $S$.

We may assume w.l.o.g. $s \in X$.

**First case $r \in X$.**

- $\mathrm{cap}(X \setminus S) + \mathrm{cap}(S \setminus X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$.
- $\mathrm{cap}(X \setminus S) \geq \mathrm{cap}(S)$ because $X \setminus S$ is an $r$-$s$ cut.
- This gives $\mathrm{cap}(S \setminus X) \leq \mathrm{cap}(X)$.

**Second case $r \notin X$.**

- $\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$.

### Lemma 19

*Let $S$ be some minimum $r$-$s$ cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum $v$-$w$-cut $T$ with $T \subset S$.*

**Proof:** Let $X$ be a minimum $v$-$w$ cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are $v$-$w$ cuts inside $S$.

We may assume w.l.o.g. $s \in X$.

**First case $r \in X$.**

- $\text{cap}(X \setminus S) + \text{cap}(S \setminus X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \setminus S) \geq \text{cap}(S)$ because $X \setminus S$ is an $r$-$s$ cut.
- This gives $\text{cap}(S \setminus X) \leq \text{cap}(X)$.

**Second case $r \notin X$.**

- $\text{cap}(X \cup S) + \text{cap}(S \cap X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \cup S) \geq \text{cap}(S)$ because $X \cup S$ is an $r$-$s$ cut.

### Lemma 19

*Let $S$ be some minimum $r$-$s$ cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum $v$-$w$-cut $T$ with $T \subset S$.*

**Proof:** Let $X$ be a minimum $v$-$w$ cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are $v$-$w$ cuts inside $S$.

We may assume w.l.o.g. $s \in X$.

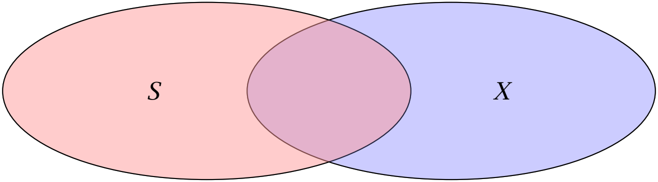
**First case $r \in X$.**

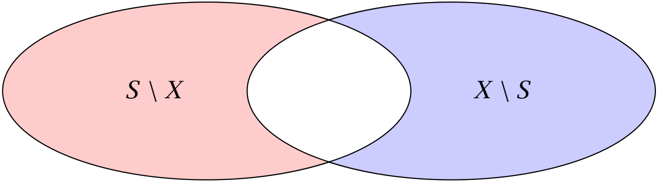
- $\text{cap}(X \setminus S) + \text{cap}(S \setminus X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \setminus S) \geq \text{cap}(S)$ because $X \setminus S$ is an $r$-$s$ cut.
- This gives $\text{cap}(S \setminus X) \leq \text{cap}(X)$.

**Second case $r \notin X$.**

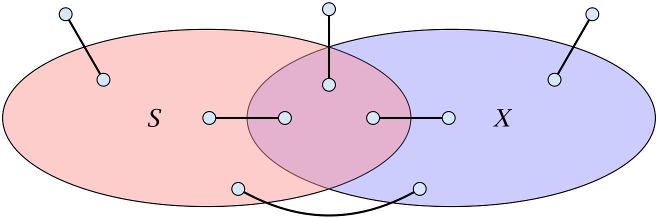
- $\text{cap}(X \cup S) + \text{cap}(S \cap X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \cup S) \geq \text{cap}(S)$ because $X \cup S$ is an $r$-$s$ cut.
- This gives $\text{cap}(S \cap X) \leq \text{cap}(X)$.

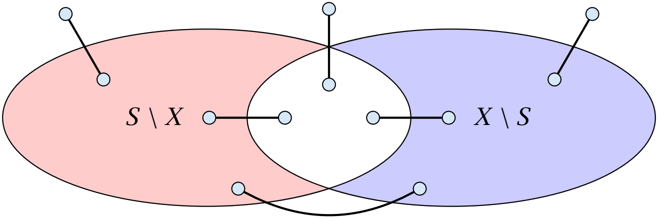
# $\mathbf{cap}(S \setminus X) + \mathbf{cap}(X \setminus S) \leq \mathbf{cap}(S) + \mathbf{cap}(X)$



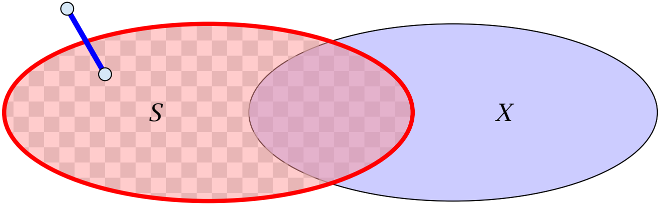



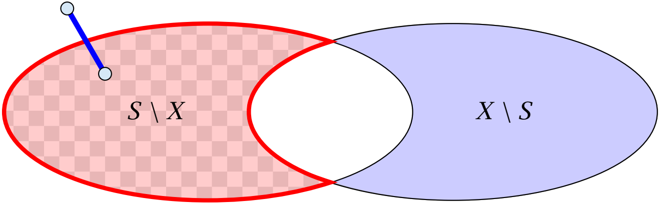
## $\text{cap}(S \setminus X) + \text{cap}(X \setminus S) \leq \text{cap}(S) + \text{cap}(X)$



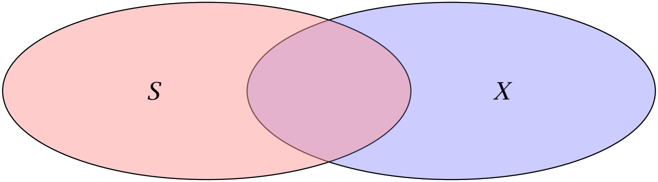



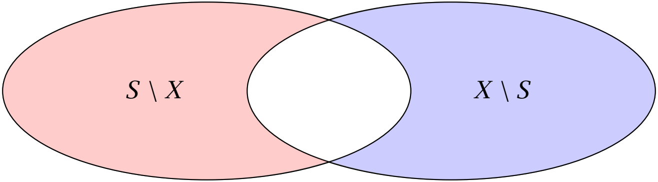
# $\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \le \mathrm{cap}(S) + \mathrm{cap}(X)$



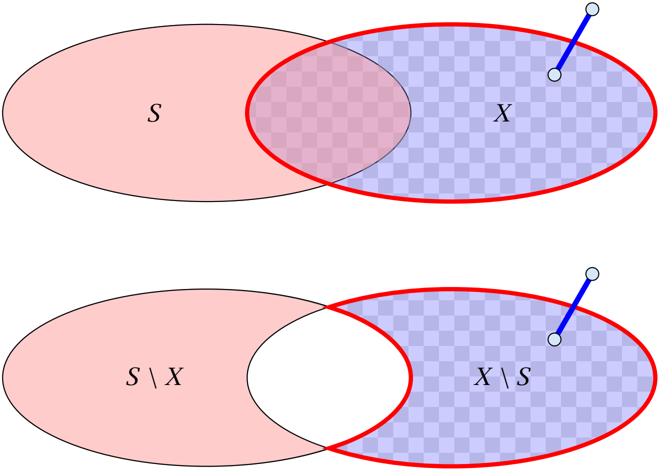



# $\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \le \mathrm{cap}(S) + \mathrm{cap}(X)$

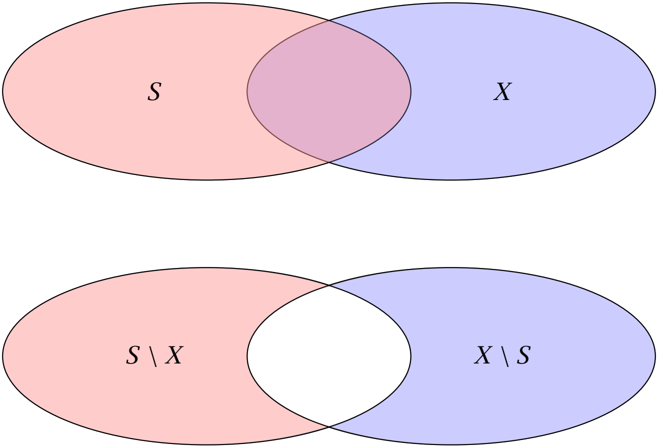


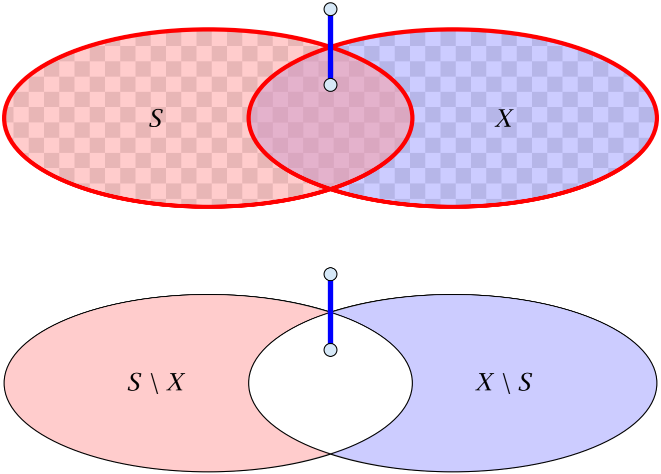
## $\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$



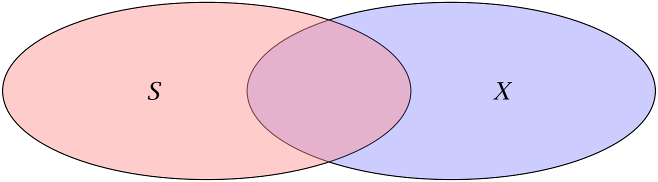
## $\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \le \mathrm{cap}(S) + \mathrm{cap}(X)$

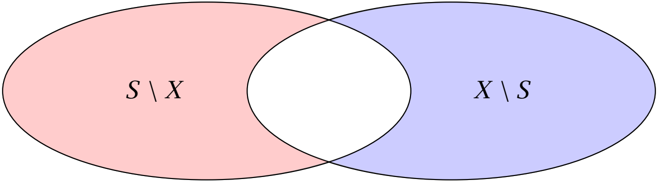


# $\mathbf{cap}(S \setminus X) + \mathbf{cap}(X \setminus S) \le \mathbf{cap}(S) + \mathbf{cap}(X)$

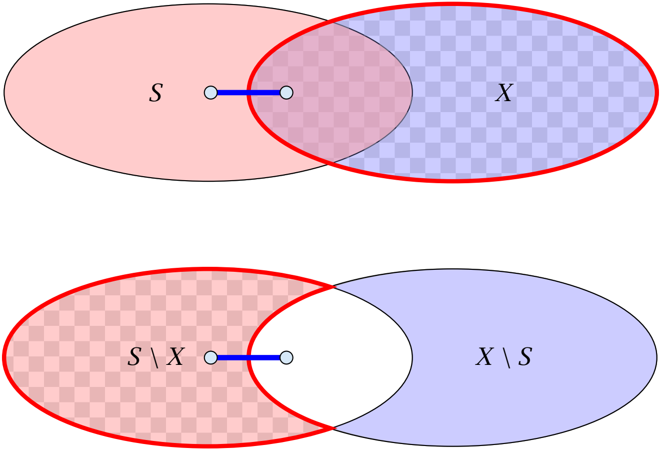


## $\mathbf{cap}(S \setminus X) + \mathbf{cap}(X \setminus S) \le \mathbf{cap}(S) + \mathbf{cap}(X)$

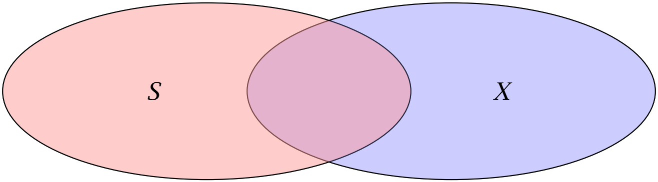


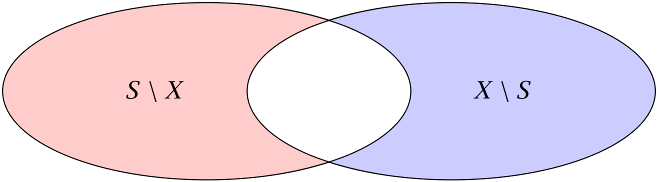



## $\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$

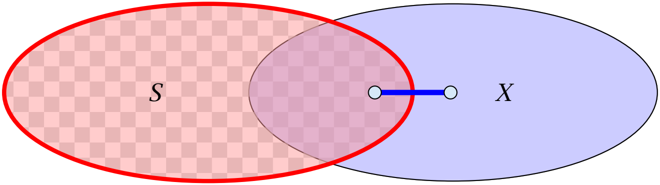


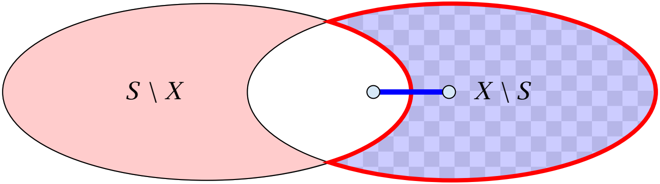
# $\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$



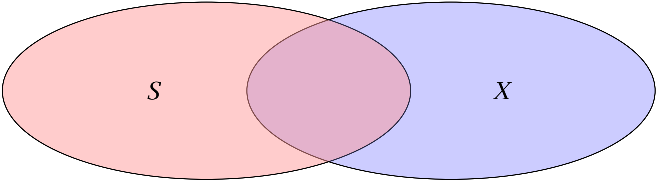



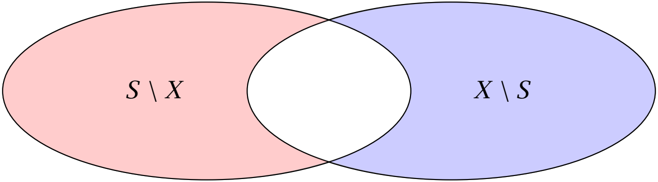
## $\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$



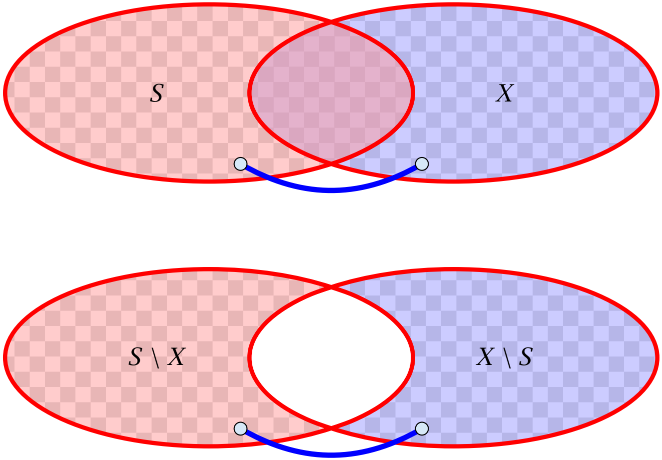



## $\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$

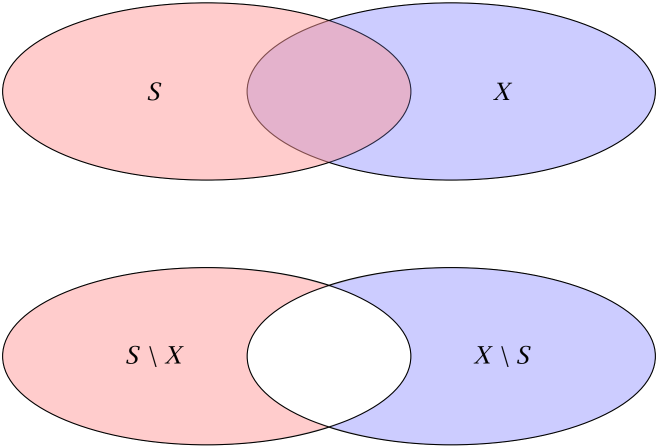


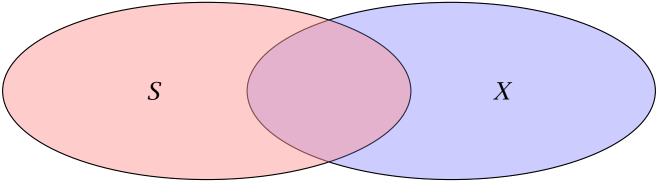
# $\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \le \mathrm{cap}(S) + \mathrm{cap}(X)$

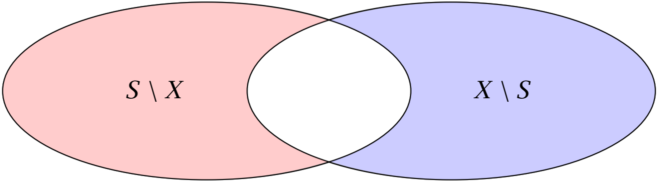


# $\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \le \mathrm{cap}(S) + \mathrm{cap}(X)$

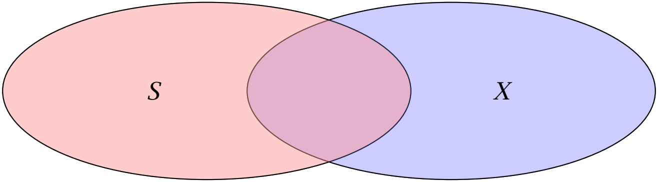


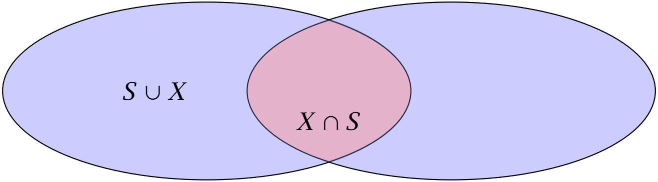
## $\mathbf{cap}(S \setminus X) + \mathbf{cap}(X \setminus S) \leq \mathbf{cap}(S) + \mathbf{cap}(X)$



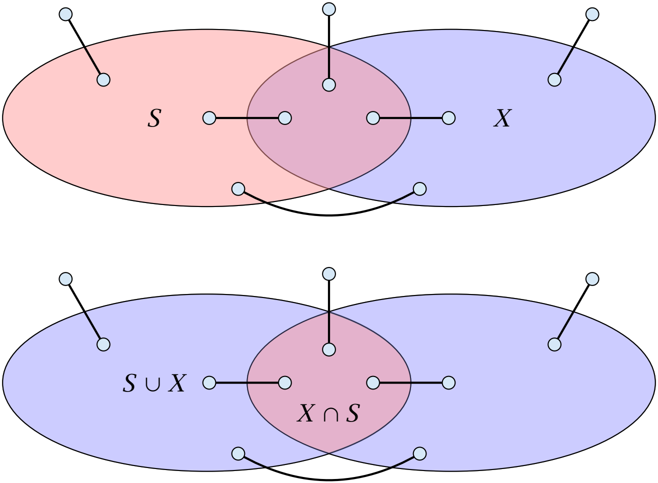



## $\text{cap}(X \cup S) + \text{cap}(S \cap X) \leq \text{cap}(S) + \text{cap}(X)$

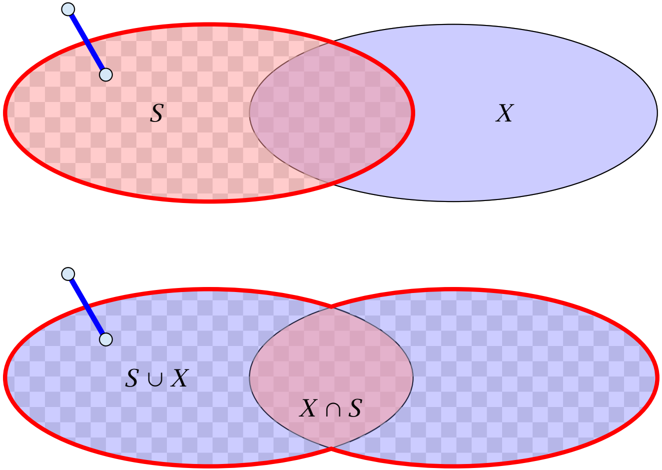


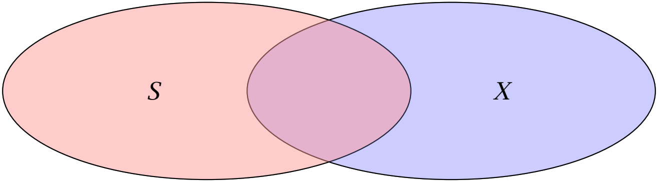
# $\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$

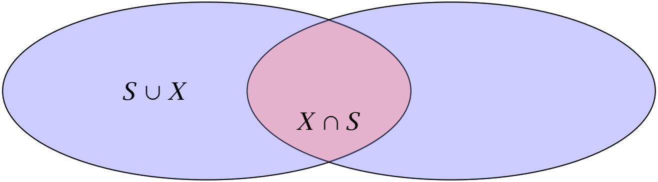


$$\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$$

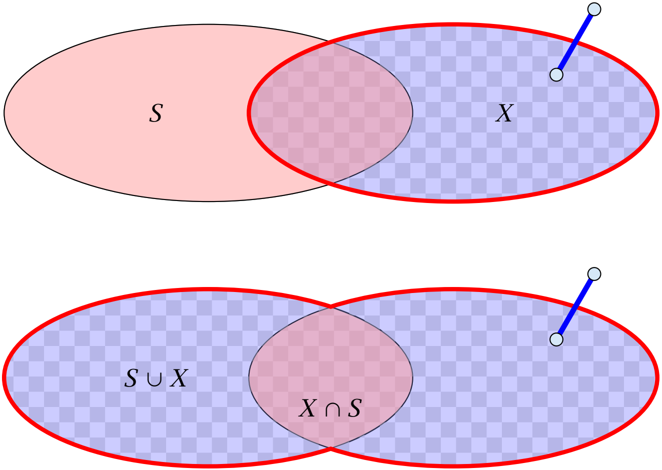


## $\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$

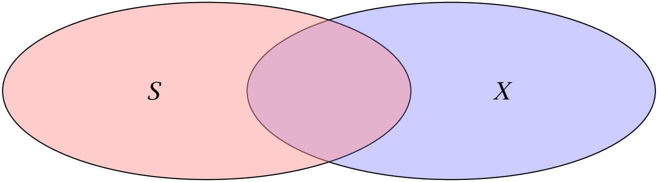


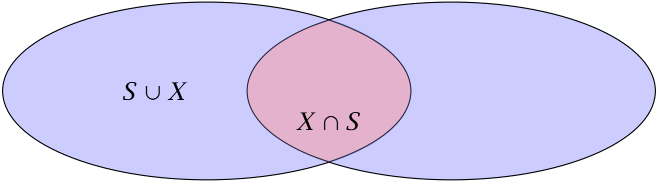



## $\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \le \mathrm{cap}(S) + \mathrm{cap}(X)$

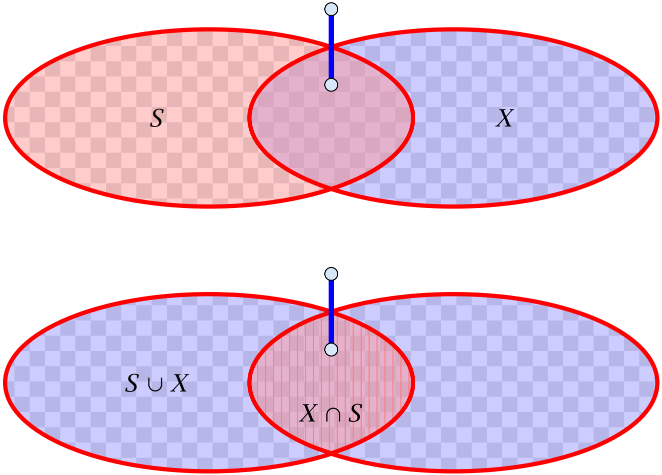


## $\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$

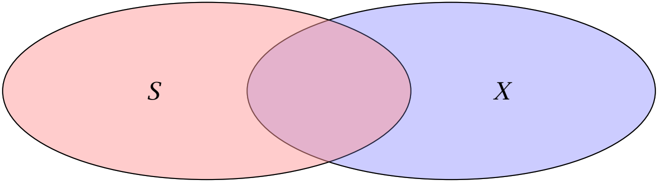


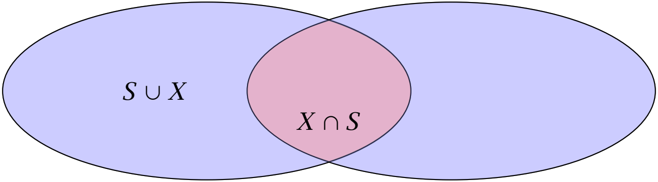



## $\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$

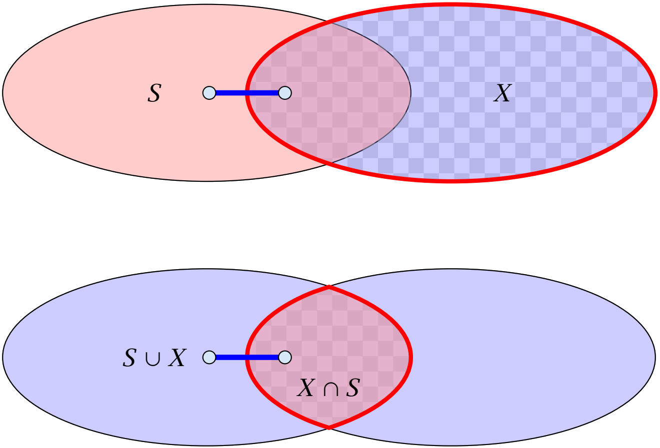


## $\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$

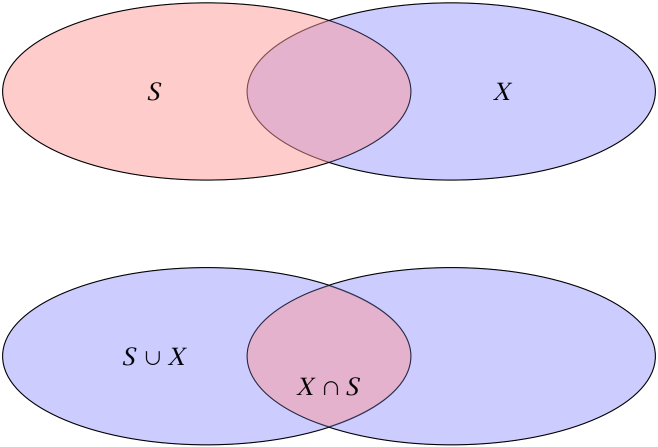


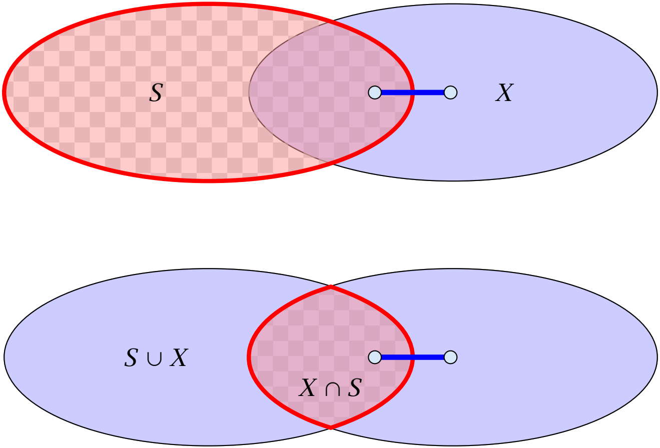
## $\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$



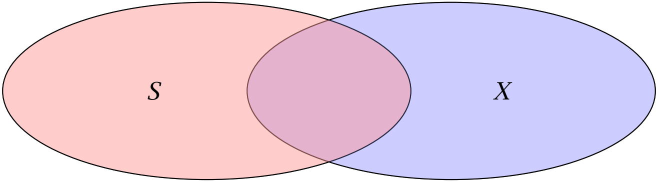
# $\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$

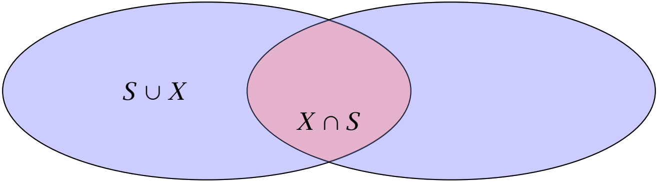


## $\mathbf{cap}(X \cup S) + \mathbf{cap}(S \cap X) \le \mathbf{cap}(S) + \mathbf{cap}(X)$

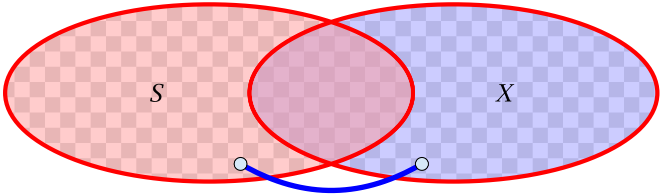


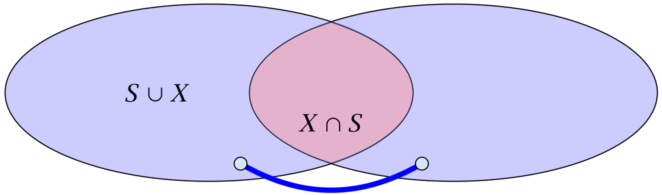
# $\text{cap}(X \cup S) + \text{cap}(S \cap X) \leq \text{cap}(S) + \text{cap}(X)$



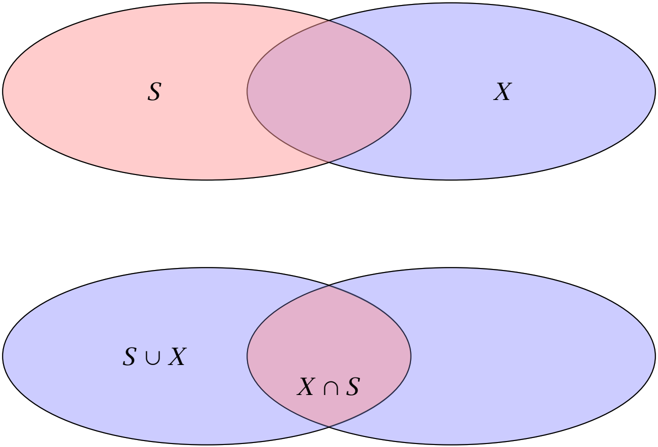



# $\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$

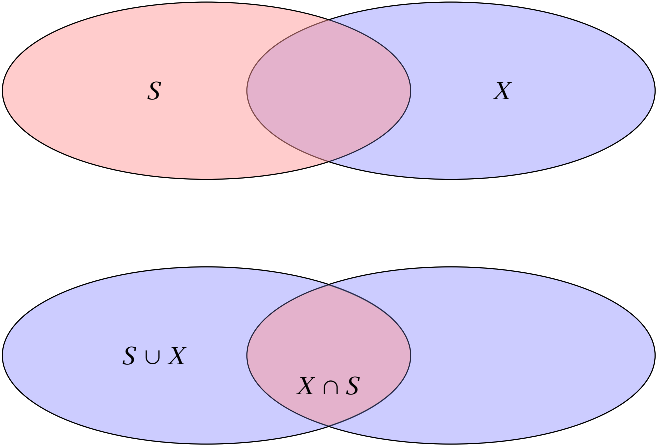


# $\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$



# $\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$



# Analysis

Lemma 19 tells us that if we have a graph $G = (V, E)$ and we contract a subset $X \subset V$ that corresponds to some mincut, then the value of $f(s,t)$ does not change for two nodes $s, t \notin X$.

We will show (later) that the connected components that we contract during a split-operation each correspond to some mincut and, hence, $f_H(s,t) = f(s,t)$, where $f_H(s,t)$ is the value of a minimum $s$-$t$ mincut in graph $H$.

# Analysis

**Invariant [existence of representatives]:**
For any edge $\{S_i, S_j\}$ in $T$, there are vertices $a \in S_i$ and $b \in S_j$ such that $w(S_i, S_j) = f(a, b)$ and the cut defined by edge $\{S_i, S_j\}$ is a minimum $a$-$b$ cut in $G$.

# Analysis

We first show that the invariant implies that at the end of the algorithm $T$ is indeed a cut-tree.

# Analysis

We first show that the invariant implies that at the end of the algorithm $T$ is indeed a cut-tree.

- Let $s = x_0, x_1, \ldots, x_{k-1}, x_k = t$ be the unique simple path from $s$ to $t$ in the final tree $T$. From the invariant we get that $f(x_i, x_{i+1}) = w(x_i, x_{i+1})$ for all $j$.

# Analysis

We first show that the invariant implies that at the end of the algorithm $T$ is indeed a cut-tree.

- Let $s = x_0, x_1, \dots, x_{k-1}, x_k = t$ be the unique simple path from $s$ to $t$ in the final tree $T$. From the invariant we get that $f(x_i, x_{i+1}) = w(x_i, x_{i+1})$ for all $j$.
- Then

$$f_T(s,t)$$

# Analysis

We first show that the invariant implies that at the end of the algorithm $T$ is indeed a cut-tree.

- Let $s = x_0, x_1, \dots, x_{k-1}, x_k = t$ be the unique simple path from $s$ to $t$ in the final tree $T$. From the invariant we get that $f(x_i, x_{i+1}) = w(x_i, x_{i+1})$ for all $j$.
- Then

$$f_T(s,t) = \min_{i \in \{0,\dots,k-1\}} \{w(x_i, x_{i+1})\}$$

# Analysis

We first show that the invariant implies that at the end of the algorithm $T$ is indeed a cut-tree.

- Let $s = x_0, x_1, \ldots, x_{k-1}, x_k = t$ be the unique simple path from $s$ to $t$ in the final tree $T$. From the invariant we get that $f(x_i, x_{i+1}) = w(x_i, x_{i+1})$ for all $j$.
- Then

$$
\begin{aligned}
f_T(s,t) &= \min_{i \in \{0,\ldots,k-1\}} \{w(x_i, x_{i+1})\} \\
&= \min_{i \in \{0,\ldots,k-1\}} \{f(x_i, x_{i+1})\}
\end{aligned}
$$

## Analysis

We first show that the invariant implies that at the end of the algorithm $T$ is indeed a cut-tree.

- Let $s = x_0, x_1, \dots, x_{k-1}, x_k = t$ be the unique simple path from $s$ to $t$ in the final tree $T$. From the invariant we get that $f(x_i, x_{i+1}) = w(x_i, x_{i+1})$ for all $j$.
- Then

$$
\begin{aligned}
f_T(s,t) &= \min_{i\in\{0,\dots,k-1\}} \{w(x_i, x_{i+1})\} \\
&= \min_{i\in\{0,\dots,k-1\}} \{f(x_i, x_{i+1})\} \le f(s,t) \enspace .
\end{aligned}
$$

# Analysis

We first show that the invariant implies that at the end of the algorithm $T$ is indeed a cut-tree.

- Let $s = x_0, x_1, \dots, x_{k-1}, x_k = t$ be the unique simple path from $s$ to $t$ in the final tree $T$. From the invariant we get that $f(x_i, x_{i+1}) = w(x_i, x_{i+1})$ for all $j$.
- Then

$$\begin{aligned} f_T(s,t) &= \min_{i \in \{0,\dots,k-1\}} \{w(x_i, x_{i+1})\} \\ &= \min_{i \in \{0,\dots,k-1\}} \{f(x_i, x_{i+1})\} \le f(s,t) \enspace . \end{aligned}$$

- Let $\{x_j, x_{j+1}\}$ be the edge with minimum weight on the path.

# Analysis

We first show that the invariant implies that at the end of the algorithm $T$ is indeed a cut-tree.

- Let $s = x_0, x_1, \dots, x_{k-1}, x_k = t$ be the unique simple path from $s$ to $t$ in the final tree $T$. From the invariant we get that $f(x_i, x_{i+1}) = w(x_i, x_{i+1})$ for all $j$.
- Then

$$
\begin{aligned}
f_T(s,t) &= \min_{i \in \{0,\dots,k-1\}} \{w(x_i, x_{i+1})\} \\
&= \min_{i \in \{0,\dots,k-1\}} \{f(x_i, x_{i+1})\} \le f(s,t) \enspace .
\end{aligned}
$$

- Let $\{x_j, x_{j+1}\}$ be the edge with minimum weight on the path.
- Since by the invariant this edge induces an $s$-$t$ cut with capacity $f(x_j, x_{j+1})$ we get $f(s,t) \le f(x_j, x_{j+1}) = f_T(s,t)$.

# Analysis

- Hence, $f_T(s,t) = f(s,t)$ (flow equivalence).

# Analysis

- Hence, $f_T(s,t) = f(s,t)$ (flow equivalence).
- The edge $\{x_j, x_{j+1}\}$ is a mincut between $s$ and $t$ in $T$.

# Analysis

- Hence, $f_T(s,t) = f(s,t)$ (flow equivalence).
- The edge $\{x_j, x_{j+1}\}$ is a mincut between $s$ and $t$ in $T$.
- By invariant, it forms a cut with capacity $f(x_j, x_{j+1})$ in $G$ (which separates $s$ and $t$).

# Analysis

- Hence, $f_T(s,t) = f(s,t)$ (flow equivalence).
- The edge $\{x_j, x_{j+1}\}$ is a mincut between $s$ and $t$ in $T$.
- By invariant, it forms a cut with capacity $f(x_j, x_{j+1})$ in $G$ (which separates $s$ and $t$).
- Since, we can send a flow of value $f(x_j, x_{j+1})$ btw. $s$ and $t$, this is an $s$-$t$ mincut (cut property).

# Proof of Invariant

# Proof of Invariant

The invariant obviously holds at the beginning of the algorithm.

# Proof of Invariant

The invariant obviously holds at the beginning of the algorithm.

Now, we show that it holds after a split-operation provided that it was true before the operation.

# Proof of Invariant

The invariant obviously holds at the beginning of the algorithm.

Now, we show that it holds after a split-operation provided that it was true before the operation.

Let $S_i$ denote our selected cluster with nodes $a$ and $b$. Because of the invariant all edges leaving $\{S_i\}$ in $T$ correspond to some mincuts.

# Proof of Invariant

The invariant obviously holds at the beginning of the algorithm.

Now, we show that it holds after a split-operation provided that it was true before the operation.

Let $S_i$ denote our selected cluster with nodes $a$ and $b$. Because of the invariant all edges leaving $\{S_i\}$ in $T$ correspond to some mincuts.

Therefore, contracting the connected components does not change the mincut btw. $a$ and $b$ due to Lemma 19.

# Proof of Invariant

The invariant obviously holds at the beginning of the algorithm.

Now, we show that it holds after a split-operation provided that it was true before the operation.

Let $S_i$ denote our selected cluster with nodes $a$ and $b$. Because of the invariant all edges leaving $\{S_i\}$ in $T$ correspond to some mincuts.

Therefore, contracting the connected components does not change the mincut btw. $a$ and $b$ due to Lemma 19.

After the split we have to choose representatives for all edges. For the new edge $\{S_i^a, S_i^b\}$ with capacity $w(S_i^a, S_i^b) = f_H(a, b)$ we can simply choose $a$ and $b$ as representatives.

# Proof of Invariant

# Proof of Invariant

For edges that are not incident to $S_i$ we do not need to change representatives as the neighbouring sets do not change.

# Proof of Invariant

For edges that are not incident to $S_i$ we do not need to change representatives as the neighbouring sets do not change.

Consider an edge $\{X, S_i\}$, and suppose that before the split it used representatives $x \in X$, and $s \in S_i$. Assume that this edge is replaced by $\{X, S_i^a\}$ in the new tree (the case when it is replaced by $\{X, S_i^b\}$ is analogous).

# Proof of Invariant

For edges that are not incident to $S_i$ we do not need to change representatives as the neighbouring sets do not change.

Consider an edge $\{X, S_i\}$, and suppose that before the split it used representatives $x \in X$, and $s \in S_i$. Assume that this edge is replaced by $\{X, S_i^a\}$ in the new tree (the case when it is replaced by $\{X, S_i^b\}$ is analogous).

If $s \in S_i^a$ we can keep $x$ and $s$ as representatives.

# Proof of Invariant

For edges that are not incident to $S_i$ we do not need to change representatives as the neighbouring sets do not change.

Consider an edge $\{X, S_i\}$, and suppose that before the split it used representatives $x \in X$, and $s \in S_i$. Assume that this edge is replaced by $\{X, S_i^a\}$ in the new tree (the case when it is replaced by $\{X, S_i^b\}$ is analogous).

If $s \in S_i^a$ we can keep $x$ and $s$ as representatives.

Otherwise, we choose $x$ and $a$ as representatives. We need to show that $f(x, a) = f(x, s)$.

# Proof of Invariant

# Proof of Invariant

Because the invariant was true before the split we know that the edge $\{X, S_i\}$ induces a cut in $G$ of capacity $f(x,s)$. Since, $x$ and $a$ are on opposite sides of this cut, we know that $f(x,a) \le f(x,s)$.

# Proof of Invariant

Because the invariant was true before the split we know that the edge $\{X, S_i\}$ induces a cut in $G$ of capacity $f(x,s)$. Since, $x$ and $a$ are on opposite sides of this cut, we know that $f(x,a) \le f(x,s)$.

The set $B$ forms a mincut separating $a$ from $b$. Contracting all nodes in this set gives a new graph $G'$ where the set $B$ is represented by node $v_B$. Because of Lemma 19 we know that $f'(x,a) = f(x,a)$ as $x, a \notin B$.

# Proof of Invariant

Because the invariant was true before the split we know that the edge $\{X, S_i\}$ induces a cut in $G$ of capacity $f(x,s)$. Since, $x$ and $a$ are on opposite sides of this cut, we know that $f(x,a) \leq f(x,s)$.

The set $B$ forms a mincut separating $a$ from $b$. Contracting all nodes in this set gives a new graph $G'$ where the set $B$ is represented by node $v_B$. Because of Lemma 19 we know that $f'(x,a) = f(x,a)$ as $x, a \notin B$.

We further have $f'(x,a) \geq \min\{f'(x,v_B), f'(v_B,a)\}$.

# Proof of Invariant

Because the invariant was true before the split we know that the edge $\{X, S_i\}$ induces a cut in $G$ of capacity $f(x,s)$. Since, $x$ and $a$ are on opposite sides of this cut, we know that $f(x,a) \leq f(x,s)$.

The set $B$ forms a mincut separating $a$ from $b$. Contracting all nodes in this set gives a new graph $G'$ where the set $B$ is represented by node $v_B$. Because of Lemma 19 we know that $f'(x,a) = f(x,a)$ as $x, a \notin B$.

We further have $f'(x,a) \geq \min\{f'(x,v_B), f'(v_B,a)\}$.

Since $s \in B$ we have $f'(v_B,x) \geq f(s,x)$.

# Proof of Invariant

Because the invariant was true before the split we know that the edge $\{X, S_i\}$ induces a cut in $G$ of capacity $f(x,s)$. Since, $x$ and $a$ are on opposite sides of this cut, we know that $f(x,a) \leq f(x,s)$.

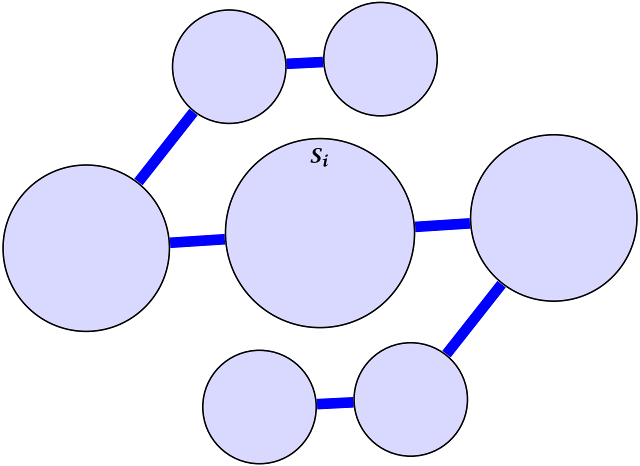
The set $B$ forms a mincut separating $a$ from $b$. Contracting all nodes in this set gives a new graph $G'$ where the set $B$ is represented by node $v_B$. Because of Lemma 19 we know that $f'(x,a) = f(x,a)$ as $x, a \notin B$.

We further have $f'(x,a) \geq \min\{f'(x,v_B), f'(v_B,a)\}$.

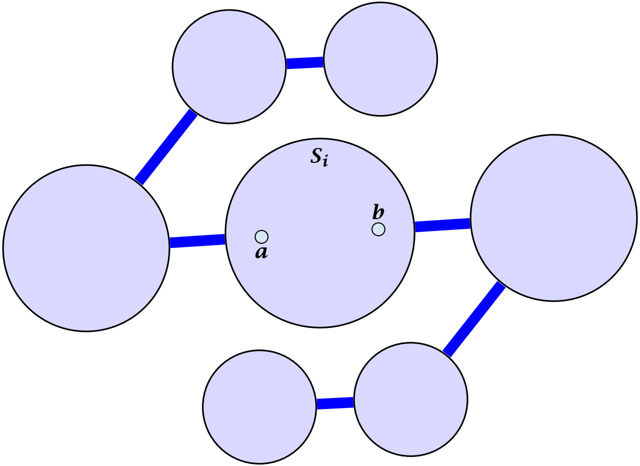
Since $s \in B$ we have $f'(v_B,x) \geq f(s,x)$.

Also, $f'(a,v_B) \geq f(a,b) \geq f(x,s)$ since the $a$-$b$ cut that splits $S_i$ into $S_i^a$ and $S_i^b$ also separates $s$ and $x$.

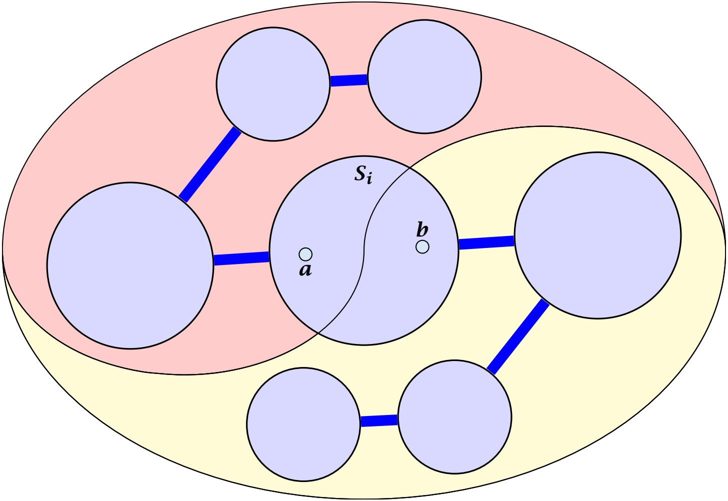
# Analysis



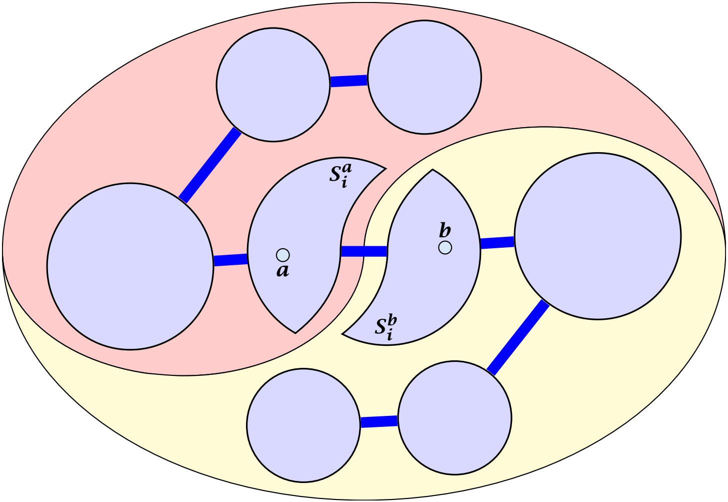
# Analysis



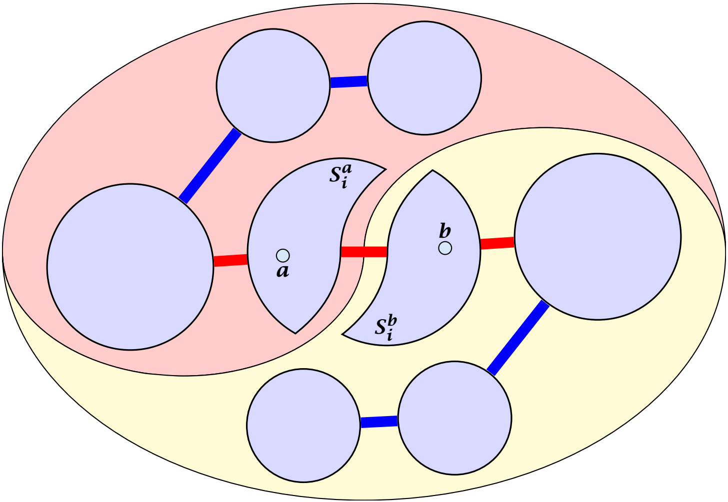
# Analysis



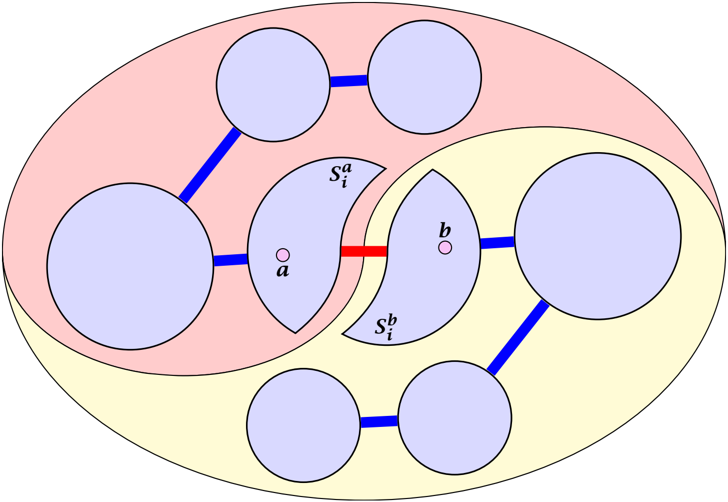
# Analysis



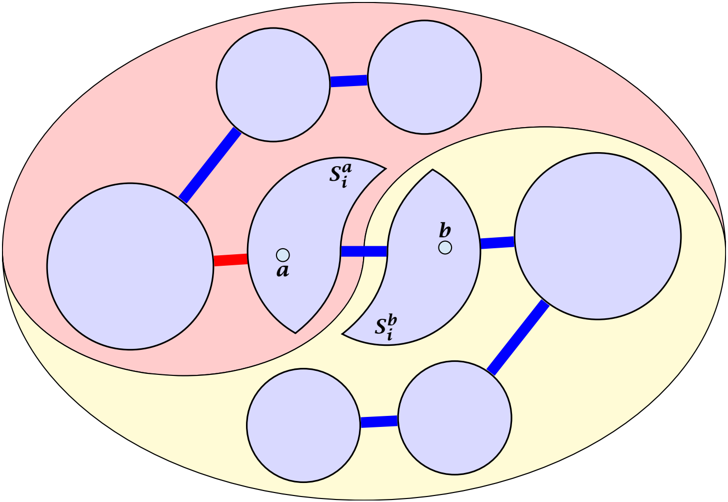
# Analysis



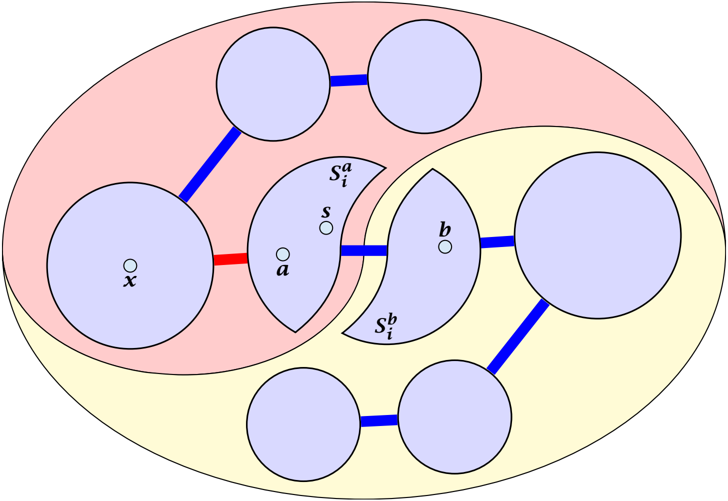
# Analysis



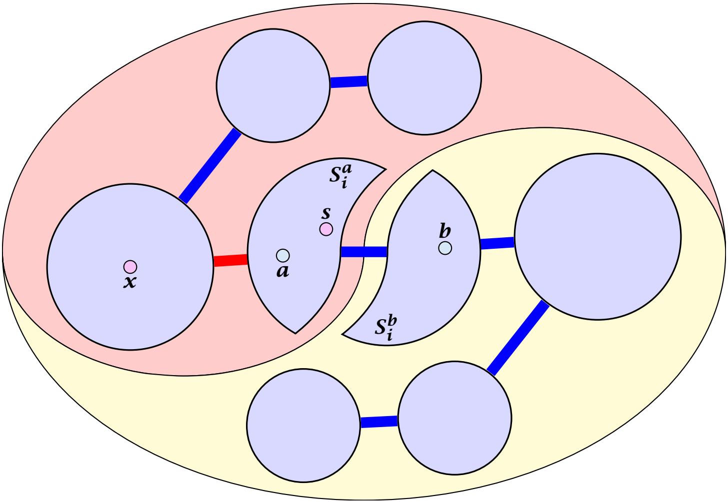
# Analysis



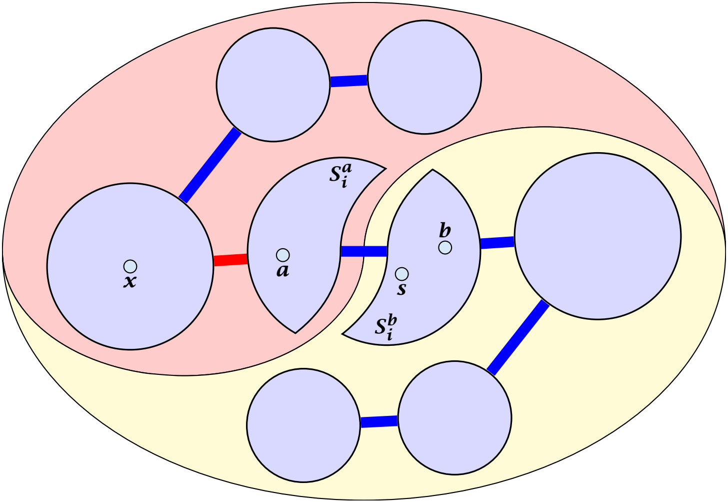
# Analysis



# Analysis



# Analysis



# Analysis