Stefan Altherr (334446)
altherr@rhrk.uni-kl.de

Seminar Report

„Algorithms for Two Bottleneck Optimization Problems“

Harold N. Gabow and Robert E. Tarjan ,1987

Content:

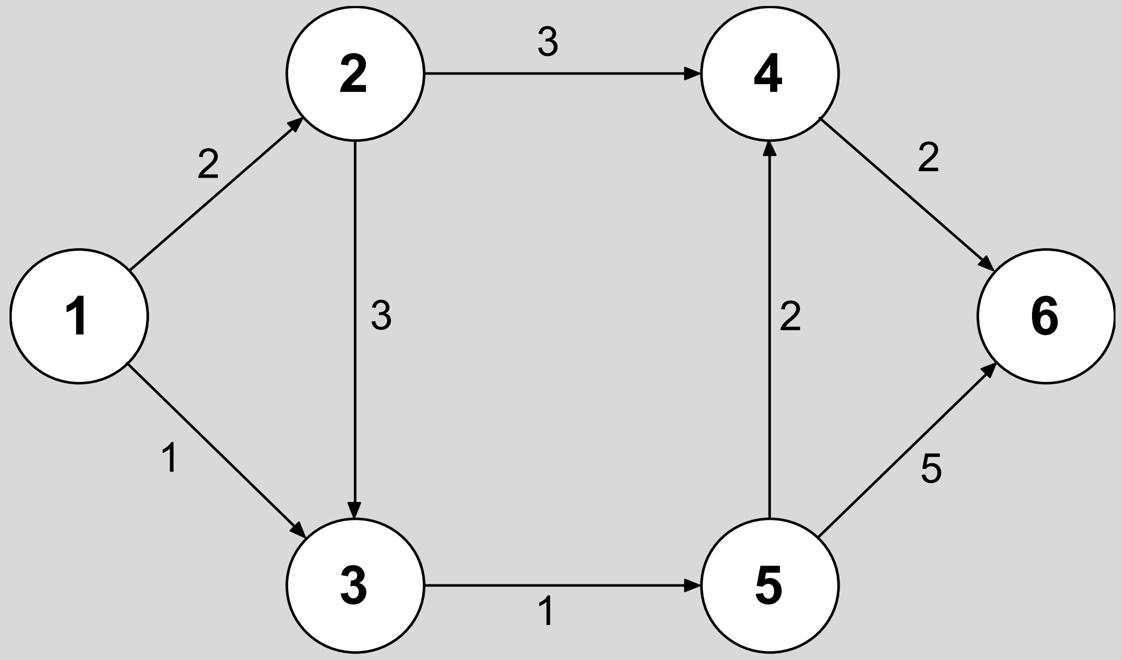
0. Introduction

Bottleneck Optimization Problem:
Find a subgraph in a graph with edge costs, that minimizes the maximum edge cost in the subgraph.

Definition:
A graph G = (V, E) is a finite, nonempty set of vertices $V = \{v_1, \ldots, v_n\}$ and edges $E = \{e_1, \ldots, e_m\}$. Each edge is defined by two endpoints and denoted by $e = [v_i, v_j]$ or $e = [i, j]$.

Definition:
The edge costs $c(i, j) = c_{ij}$ define the "length" of an edge. If two vertices i, j are not directly connected by an edge, the edge cost is set to infinity, i.e. $e_{ij} = \infty$.

Example:
Directed graph with non-negative edge costs

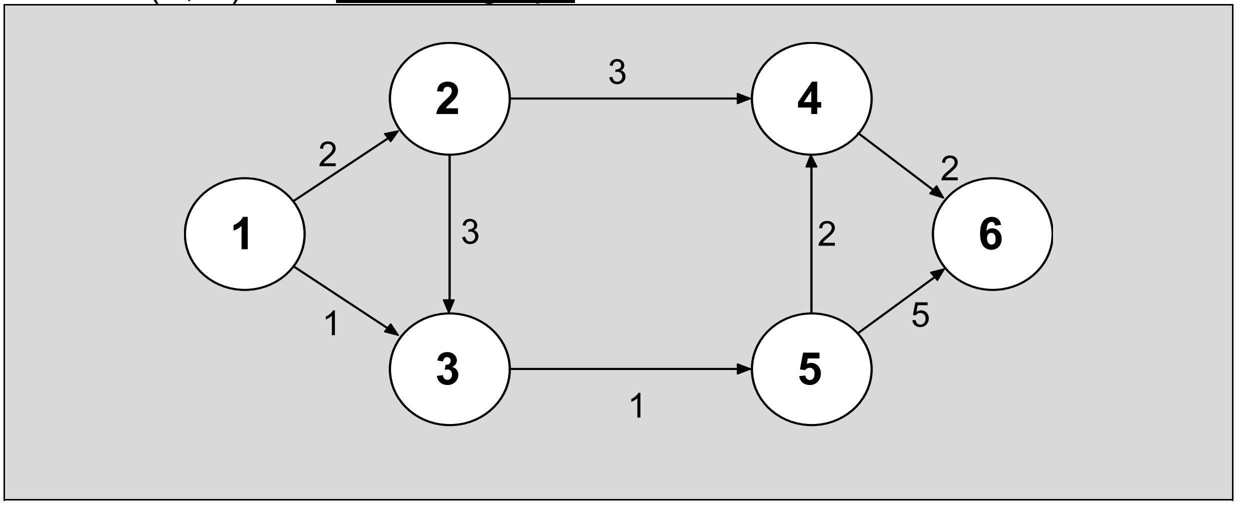


The paper proposes two algorithms for finding a bottleneck spanning tree in a directed graph and the bottleneck maximum cardinality matching problem.

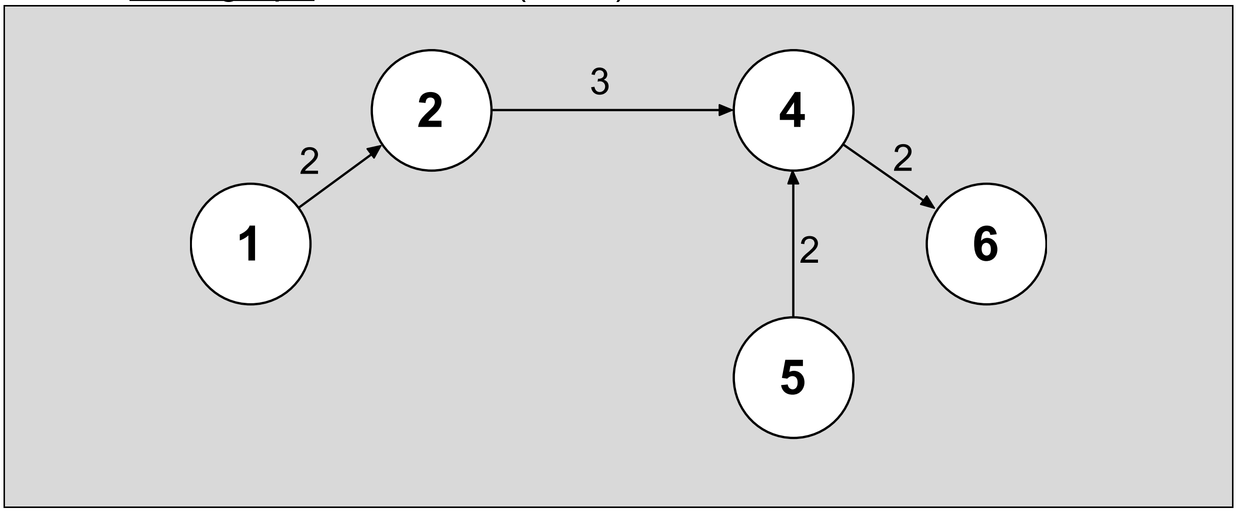
1. Bottleneck Spanning Trees in Directed Graphs

1.1 Basics

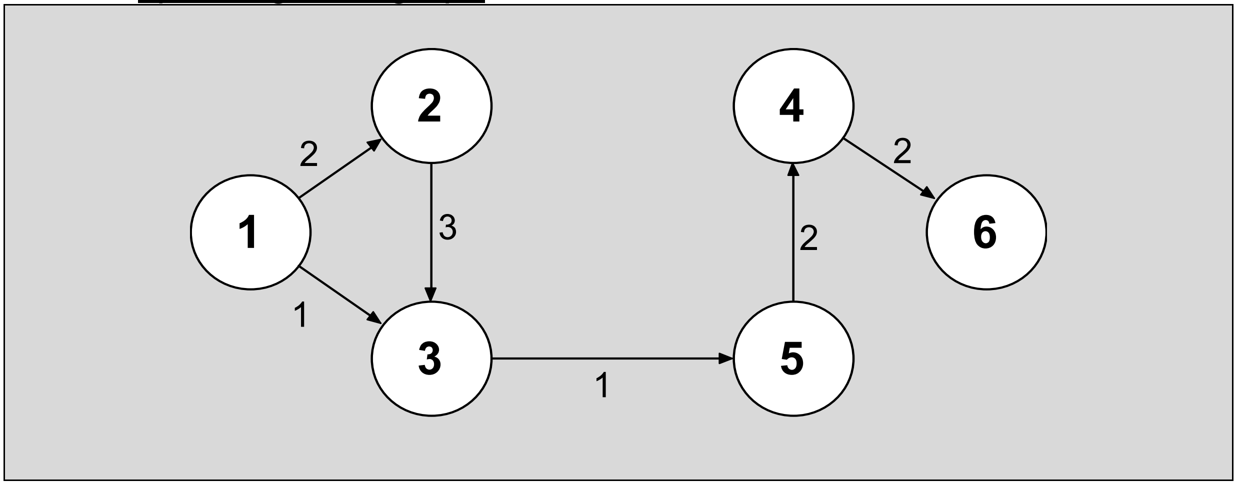
Let G = (V, E) be a directed graph.



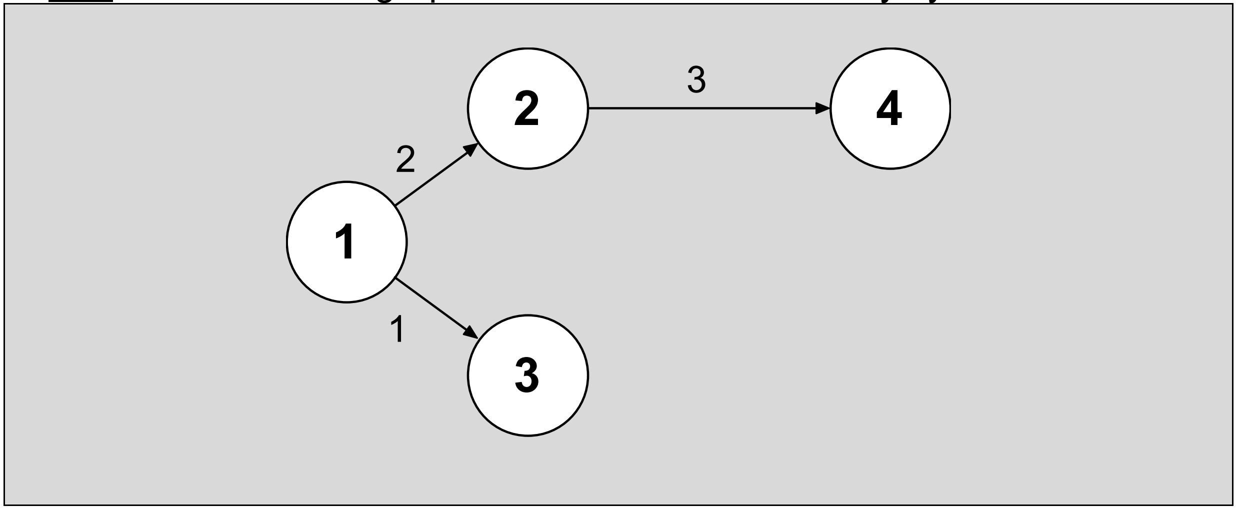
G' is a subdigraph of G if G' = (V', E') with $V' \subset V$ and $E' \subset E$.



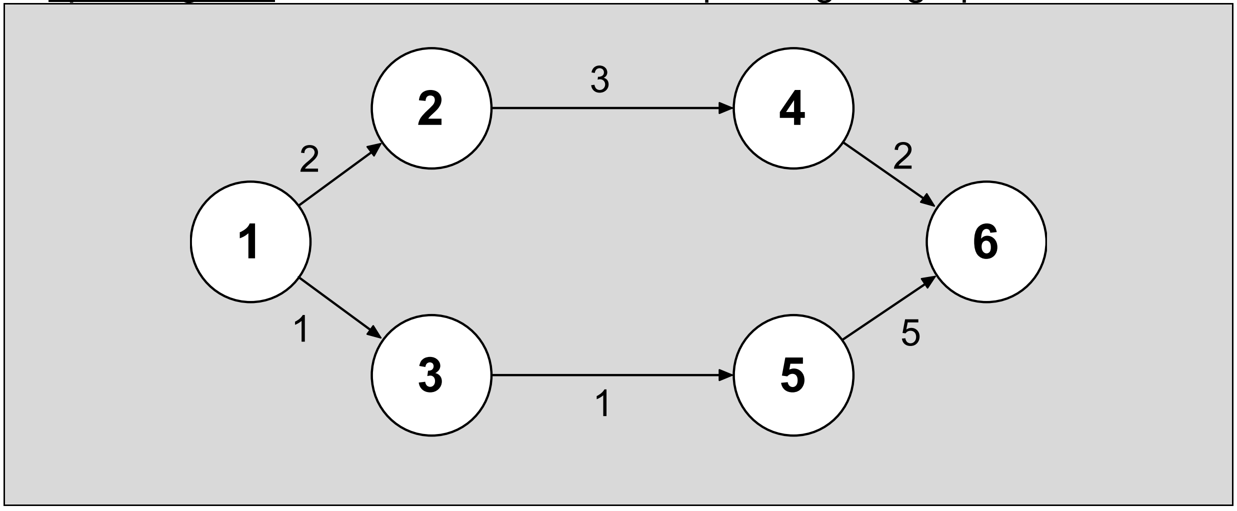
G'' is a spanning subdigraph of G if V'=V.



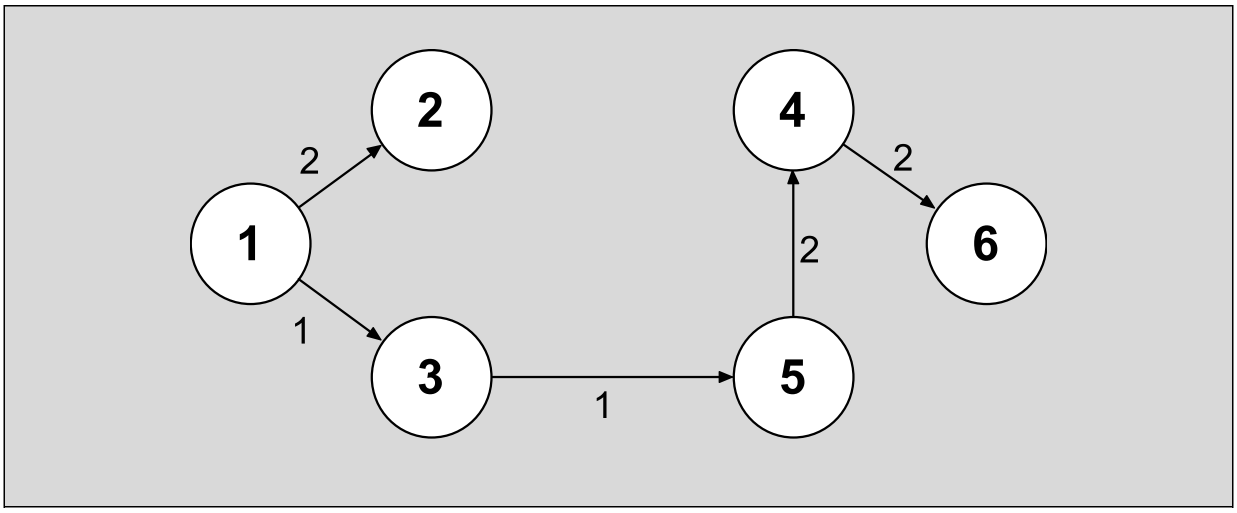
A tree is a connected graph that does not contain any cycles.



A spanning tree T of G is a tree that is a spanning subgraph of G.



A bottleneck spanning tree is a spanning tree whose maximum edge cost is minimum.



Our goal is to find such a bottleneck spanning tree in a directed graph.
We consider the problem of finding a spanning tree rooted at s (containing paths from s to all other vertices) whose maximum edge cost is minimum.

For this let G = (V, E) be a directed graph with n vertices and m edges. (wlog $m \geq n \geq 2$).
Let s be a root vertex of G and for each edge (v, w) let c_{vw} be a real valued cost.

1.2 Dijkstra's Algorithm

To find a solution for our bottleneck problem we start with Dijkstra's algorithm. This algorithm solves the shortest directed graph dipath problem with respect to non-negative costs c(e) with a complexity of $O(n^2)$.
The algorithm works as follows:

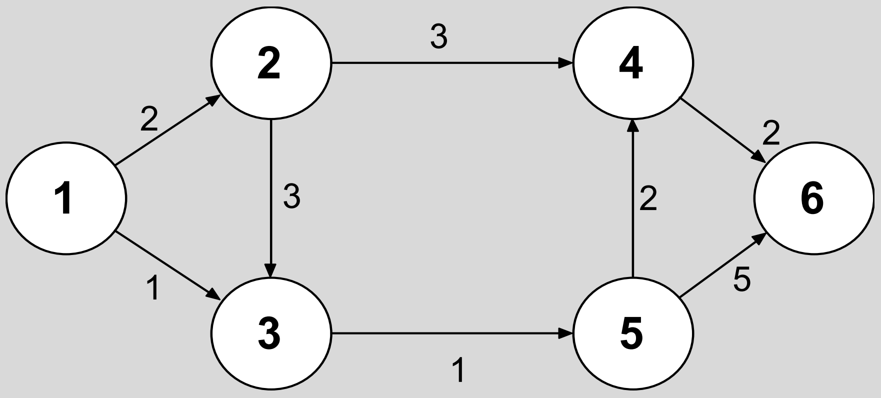
```
DIJKSTRA'S ALGORITHM

(INPUT) $G = (V, E)$ digraph with costs $c(e) \geq 0$ $(\forall e \in E)$

(1)Set $d_i := \infty$ $(\forall i = 1, \ldots, n)$, $d_s := 0$, pred(s) := 0, p(s) := 1.
    a. Set $X_p := \{s\}$
    b. $p_i := c_{si}$        $(\forall i \in E \setminus X_p)$
    c. pred(i) = s    $(\forall i \in E \setminus X_p)$
(2)Determine $I_{p+1}$ with $p_{i(p+1)} = \min_{i \in E/X_p} p_i$ and set $d_{i(p+1)} := p_{i(p+1)}$.
    If $p_{i(p+1)} = \infty$ (STOP), dipath from s to i, $\forall i \in X_p$ do not exist.
(3)Set $X_{p+1} := X_p \cup \{i_{p+1}\}$
    p := p+1
    If p = n (STOP), all shortest dipaths $P_{si}$ are known.
(4)For all $i \in X_p$ do
    If $p_i > d_{i(p)} + c_{i(p),i}$ set $p_i := d_{i(p)} + c_{i(p),i}$
    pred(i) := $i_p$.
    Goto Step (2)
```

When the algorithm terminates, the subgraph $T = (V', E')$ with $V' := \{i \mid d_i < \infty\}$ and $E' := \{(pred(j), j \mid j \in V')\}$ contains no cycle. The tree T has the property that there exists a uniquely defined path from the source-node s to i for every $i \in V'$. These paths are the shortest dipath from s to i. T is called the shortest dipath tree with root s. T is a spanning tree if $d_i < \infty$ $\forall i \in V$.

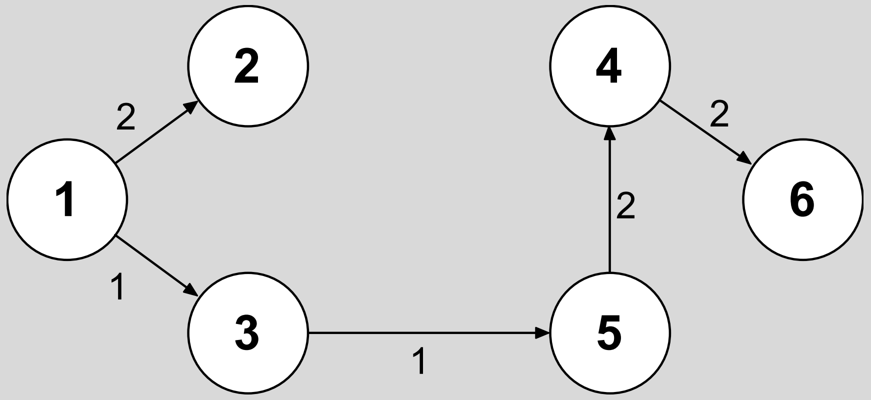
Example:



Determine shortest dipath from s = 1 to $v_{i,}$ i = 2, …, 6.

P		v2	v3	v4	v5	v6	X(p+1)
1	p(j)	2	1	inf	inf	inf	
	pred(j)	1	1	1	1	1	{v1, v3}
2	p(j)	2	-	inf	2	inf	
	pred(j)	1	-	1	3	-	{v1, v3, v2}
3	p(j)	-	-	5	2	inf	
	pred(j)	-	-	2	3	-	{v1, v3, v2, v5}
4	p(j)	-	-	4	-	7	
	pred(j)	-	-	5	-	5	{v1, v3, v2, v5, v4}
5	p(j)	-	-	-	-	6	
	pred(j)	-	-	-	-	4	{v1, v3, v2, v5, v4, v6}

T = (V', E') with
V' := {i | $d_i < \infty$} = {v_1, …, v_6} and
E' := {(pred(j), j) | j $\in$ V'} = {(1, 3), (1, 2), (3, 5), 5, 4), (4, 6)}
➔ Shortest dipath tree with root v_1



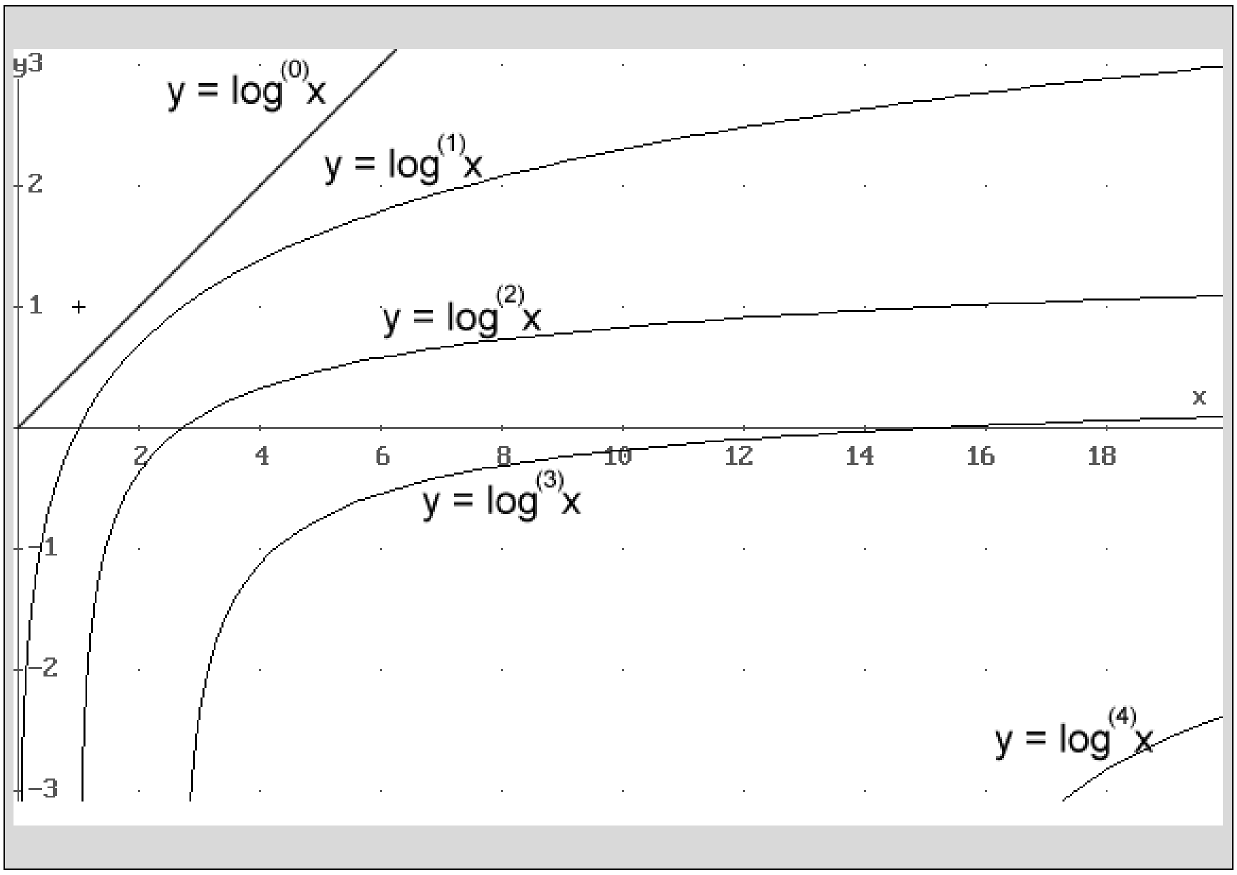
1.3 Revised Algorithm by Gabow and Tarjan

While Dijkstra's algorithm runs in O(n log n + m) time, for sparse graphs Gabow and Tarjan build an algorithm with the better bound of O(m log*n). log* is the iterated logarithm defined by

$\log^{(0)}x = x$

$\log^{(i+1)}x = \log \log^{(i)}x$

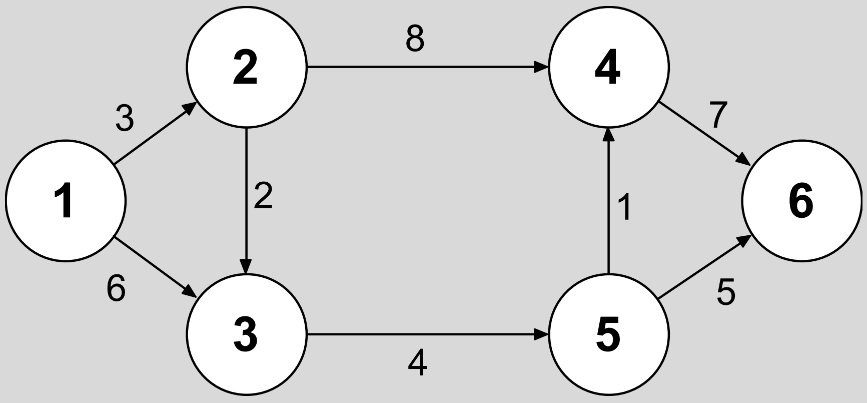
$\log^*x = \min[i \mid \log^{(i)}x \leq 1\}$.



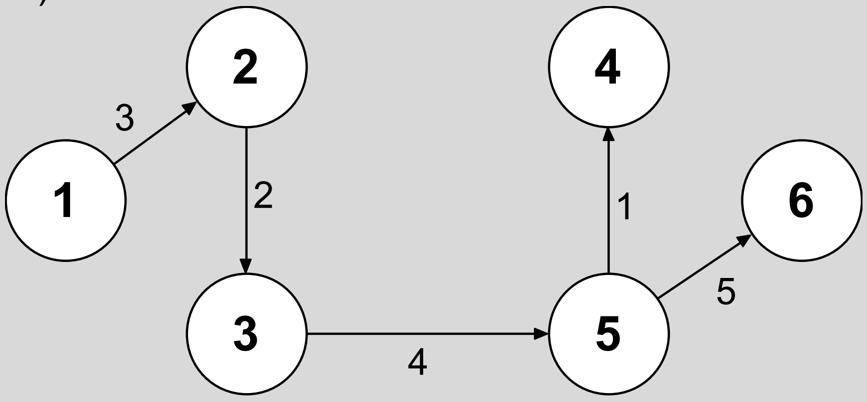
For any real number λ, let $G(\lambda) = (V, \{(v, w) \in E \mid c(v, w) = c_{vw} \leq \lambda\})$, which means, that all edges with cost larger than λ are cut away.
Further $\lambda^* = \min\{\lambda \mid \forall v \in V$ there is a path in $G(\lambda)$ from s to v$\}$.

Example:

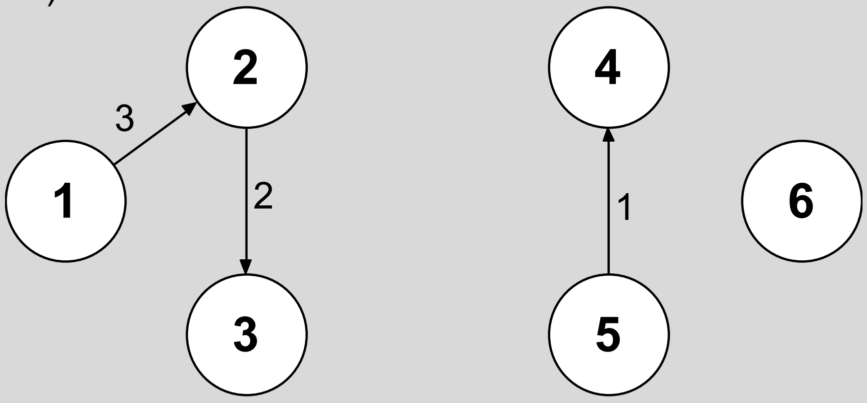
G = (V, E):



G(5) = (V, E'):



G(3) = (V, E''):



To find a bottleneck spanning tree, it suffices to compute λ^*, since any spanning tree in $G(\lambda^*)$ is a bottleneck spanning tree for G and it can be found in O(m) time.

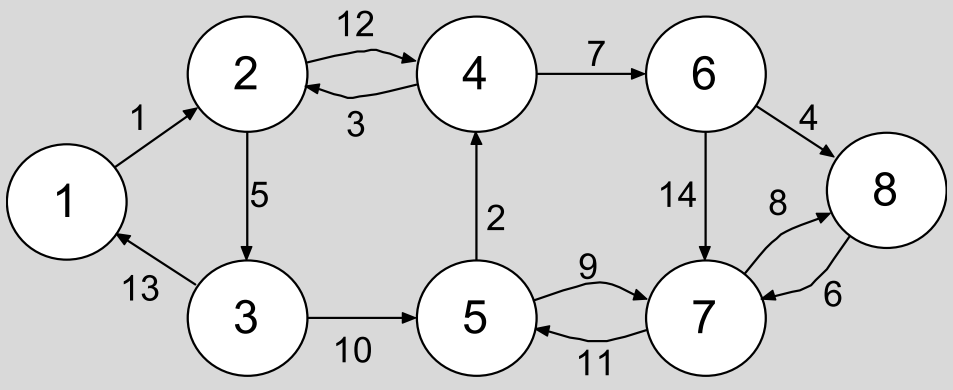
To compute λ^*, we split repeatedly to narrow the interval of possible values. The number of intervals k(i) is a function of the number of splits that have taken place. The algorithm computes values λ_1, λ_2 such that $\lambda_1 \leq \lambda^* \leq \lambda_2$.
When we start the algorithm λ_1 respectively λ_2 are the minimum and maximum edge costs, and i counts the iterations.

GABOW - TARJAN SPLITTING ALGORITHM

(1) Set i := i+1. Let $S_0 = \{(v, w) \in E \mid c_{vw} \leq \lambda_1\}$ and $E_1 = \{(v, w) \in E \mid \lambda_1 < c_{vw} \leq \lambda_2\}$.
(2) Partition E_1 into k(i) subsets $S_1, S_2, \ldots, S_{k(i)}$, such that if $(v, w) \in S_i$ and $(x, y) \in S_{i+1} \Rightarrow c_{vw} \leq c_{xy}$.
(3) Find the minimum j, such that in $G_j = (V, S_0 \cup S_1 \cup \ldots \cup S_j)$ all vertices are reachable from s.
(4) If $j = 0 \Rightarrow \lambda^* = \lambda_1$ (STOP)
Else replace λ_1, λ_2 respectively by the minimum cost and the maximum cost of an edge in S_j. (GOTO (1)).

Example:

Given G = (V, E):



(INPUT) Set i = 0; λ_1 = 1; λ_2 = 14, we chose $k(i) = 2^i$, source $s = v_1$

(1) i = 0+1 = 1
S_0 = {(1, 2)}
E_1 = {(5, 4), (4, 2), (6, 8), (2, 3), (8, 7), (4, 6), (7, 8), (5, 7), (3, 5), (7, 5), (2, 4), (3, 1), (6, 7)}

(2) $k(i) = 2^1 = 2$
S_1 = {(5, 4), (4, 2), (6, 8), (2, 3), (8, 7), (4, 6), (7, 8)}
S_2 = {(5, 7), (3, 5), (7, 5), (2, 4), (3, 1), (6, 7)}
(Only vertex 2 can be reached from s)

(3) $G_j = (V, S_0 \cup S_1 \cup S_2)$ ➔ j = 2

(4) λ_1 = 9, λ_2 = 14

(1) i = 1+1 = 2
S_0 = {(1, 2), (5, 4), (4, 2), (6, 8), (2, 3), (8, 7), (4, 6), (7, 8)}
E_1 = {(7, 5), (2, 4), (3, 1), (6, 7)}

(2) $k(i) = 2^2 = 4$
S_1 = {(3, 5), (7, 5)}
S_2 = {(2, 4)}
S_3 = {(3, 1)}
S_4 = {(6, 7)}
(Only vertices 2 and 3 can be reached from s)

(3) $G_j = (V, S_0 \cup S_1)$ ➔ j = 1

(4) λ_1 = 10, λ_2 = 11

(1) i = 2+1 = 3
S_0 = {(1, 2), (5, 4), (4, 2), (6, 8), (2, 3), (8, 7), (4, 6), (7, 8), (3, 5), (3, 5)}
E_1 = {(7, 5})

(2) $k(i) = 2^3 = 8$
S_1 = {7, 5)}
(All vertices can be reached from s)

(3) $G_j = (V, S_0)$ ➔ j = 0

$\lambda^* = \lambda_1$ = 10 ➔ (STOP)

1.4 Computing Times

- Steps (1) and (4) each take O(m) time per iteration.

- Step (3) takes O(m) time using an incremental search. Start at s and advance only along edges in S_0. If the search stops before all vertices are reached, edges in S_1 become eligible for searching and so on.
  This search algorithm takes O(m) time. We have a closer look at it in 1.5

- Gabow and Tarjan chose k(i) different than we did in the example. They define:
  $k(1) = 2; k(i) = 2^{k(i-1)}$

  $\boxed{k(1) = 2, k(2) = 2^2 = 4, k(3) = 2^4 = 16, k(4) = 2^{16} = 65536, \ldots}$

  This choice guarantees that in the i-th iteration $|E_1| = O(\frac{m}{k(i-1)})$,
  which implies that step (2) in the i-th iteration takes $O(\frac{m}{k(i-1)} \log k(i)) = O(m)$ time.
  Each iteration takes O(m) time and we have to perform O(log*n) iterations ➔ the algorithm computes in O(m log*n) time.

1.5 Further Information

- In Step (2) we partition E_1 into k(i) subsets. Dividing $|E_1|$ by k(i) generally does not deliver integers. So the sizes of $S_1, \ldots, S_{k(i)}$ can be chosen $\lceil |E_1| / k(i) \rceil$ or $\lfloor |E_1| / k(i) \rfloor$ without inflicting with the algorithm's result or performance.
- Implementation of the incremental search in step (3) requires an adjacency list A(v) for each vertex v. Each vertex is in one of three states: unlabeled, labeled or scanned.
- The search can be performed using the following algorithm:

```
SEARCH ALGORITHM (STEP (3) OF GABOW – TARJAN
SPLITTING ALGORITHM)

(3.0) Set s labeled and all other vertices unlabeled.
      Set j := 0, a(v) = {w | (v, w) ∈ S_0}.
(3.1) If some vertex is labeled, select a labeled vertex v and
      mark it scanned ➔ (GOTO (3.2))
      Else if no vertex is unlabeled ➔ (STOP)
      Else ➔ (GOTO (3.3))
(3.2) For every vertex w ∈ A(v), if w is unlabeled, mark it
      labelled ➔ (GOTO (3.1))
(3.3) Set j := j+1. For each edge (v, w)  S_j, if v is unlabeled,
      add w to A(v).
      Else if w is unlabeled, mark w labeled ➔ (GOTO (3.1))
```

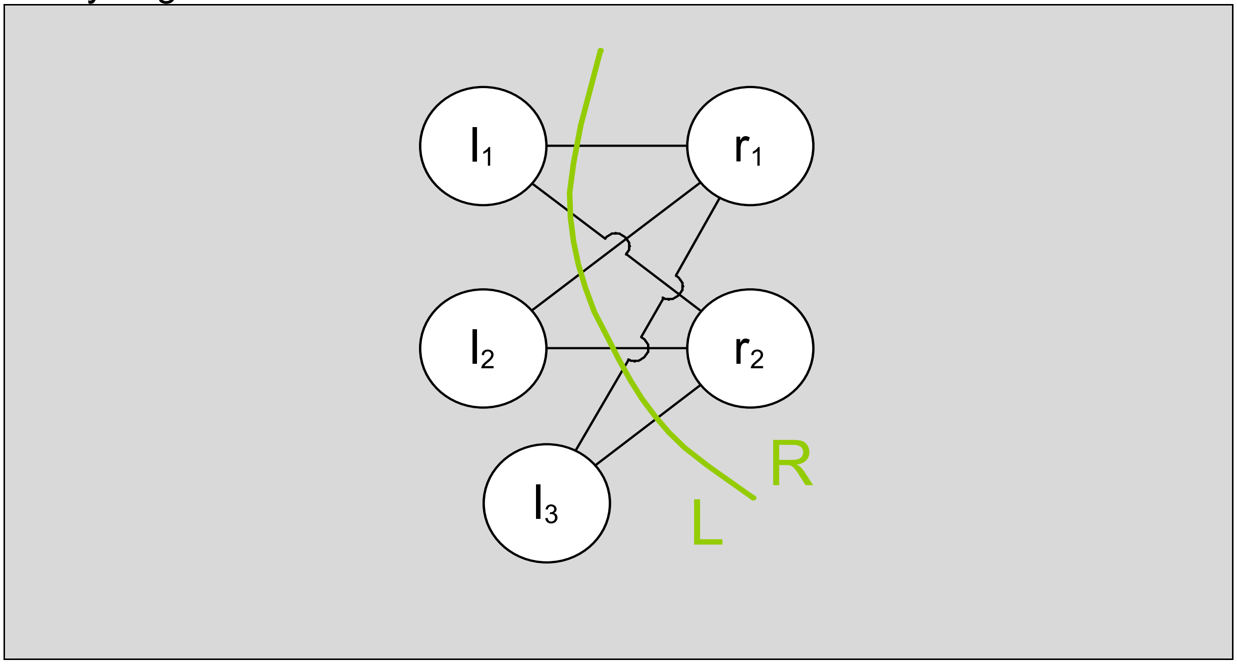
This search algorithm is a formal implementation of what we did intuitively in the example of the splitting algorithm.

- With the same splitting algorithm we can solve the bottleneck shortest path problem.
  Two vertices s, t are given in a directed graph and the task is to find a path from s to t that minimizes the maximum cost of an edge on the path.
  All we have to do is to stop when vertex t is reached in step (3) instead of checking for all vertices. The algorithm performs in $O(\min(n \log n + m, m \log^* n))$ time.

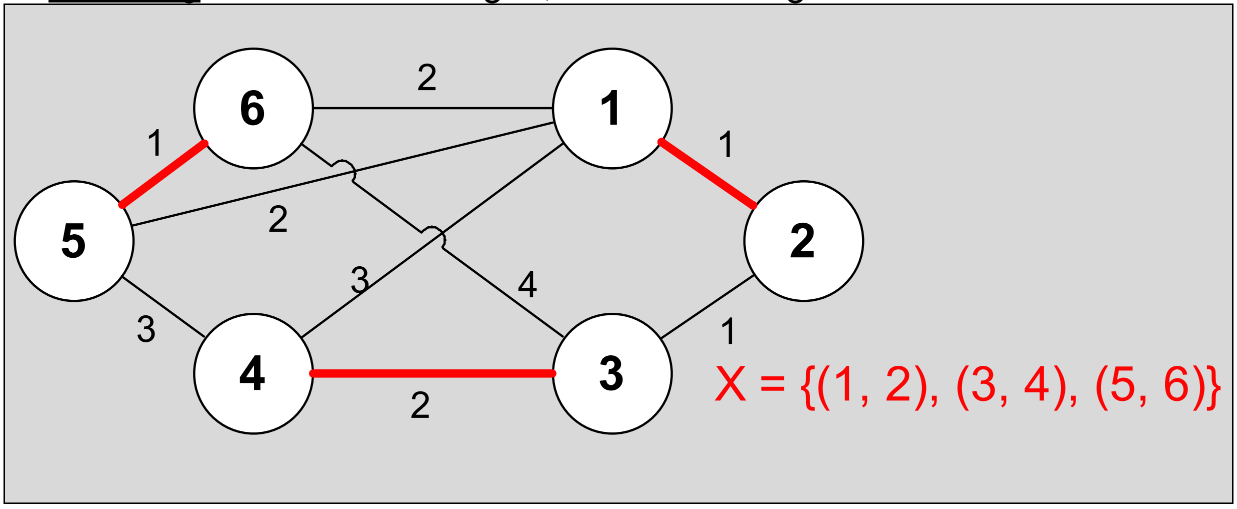
2. Bottleneck Maximum Cardinality Matchings

2.1 Basics

Let G = (V, E) be an undirected graph.
G is bipartite if V can be partitioned into two sets L and R, such that every edge has one vertex in L and one vertex in R.



A matching is a subset of edges, no two sharing a common vertex.



A maximum cardinality matching is a matching containing as many edges as possible (as shown above).

In this chapter we consider the problem of finding a maximum cardinality matching whose maximum edge cost is minimal. We call this a bottleneck matching problem.

To find a bottleneck matching K.V.S. Baht used a maximum cardinality matching algorithm in combination with binary search which performed in $O(\sqrt{n}\ \ m \log n)$ time.
This method is described below and then modified to work in $O(\sqrt{n \log n}\ m)$ time.

2.2 Maximum Cardinality Matchings

In this chapter we want to show an algorithm that completes a maximum cardinality matching in bipartite graphs in $O(\sqrt{n}\ m)$ time from Hopcroft and Karp (1973).
An algorithm which is also bound by $O(\sqrt{n}\ m)$ for general graphs was developed by Micali and Vazirani seven years later but can not be shown here due to time constraints.
The former algorithm is based on two theorems. Those we want to show now.

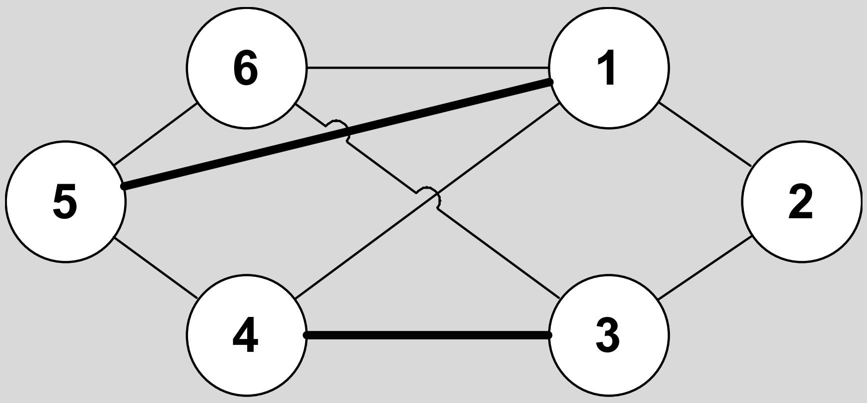
2.2.1 Theorem about Matching Augmenting Path

Definition:
Vertices that are *not* incident to matching edges v ∈ X are called unmatched or exposed vertices.

Definition:
A path P = (i_1, …, i_q, i_{q+1}, …, i_e) is called an alternating path wrt. X, if l ≥ 2 and if the edges are alternating free and matched, where the starting edge is free.

Definition:
A matching augmenting path (ma-path) is an alternating path P for which both endpoints are exposed.

Example:



X = {(1, 5), (3, 4)}
Exposed vertices: 2, 6
P_1 = (5, 4, 3) is an alternating path but not an ma-path.
P_2 = (6, 1, 5, 4, 3, 2,) and P_3 = (2, 1, 5, 6) are ma-paths.

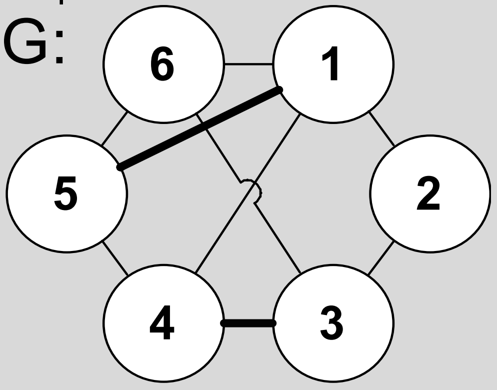
Theorem:
X is a maximal matching in G = (v, E) if and only if there does not exist an ma-path wrt. X.

Proof:
We will show: X is not a maximal matching iff there exists an ma-path P = (i_1, ..., i_e).

"←" We will show, that the symmetric difference
$X' := X \Delta P = (X \setminus P) \dot{\cup} (P \setminus X)$ is a matching with $|X'| = |X| + 1$.

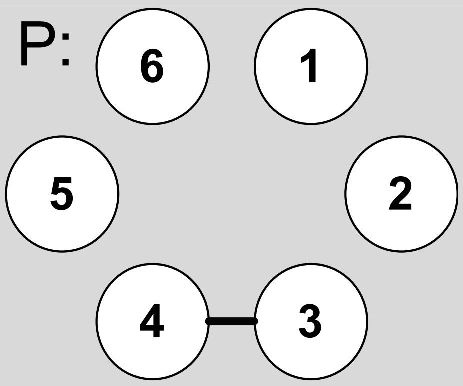
Example:

G:

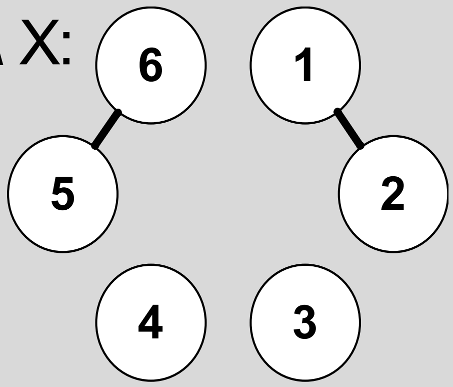


$X = \{(1, 5), (3, 4)\}$
$|X| = 2$
$P = (2, 1, 5, 6)$

X \ P:

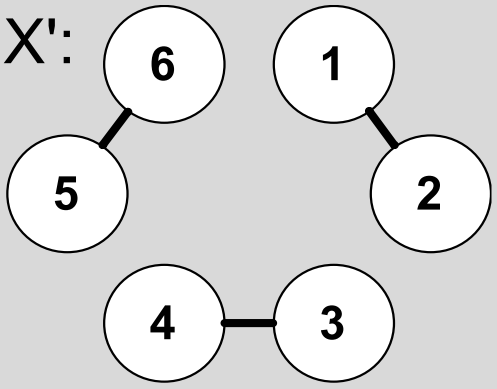


P \ X:



$X' = (X \setminus P) \dot{\cup} (P \setminus X)$
$= \{(1, 2), (3, 4), (5, 6)\}$
$|X'| = 3$

X':



Since both endpoints of P are exposed, we know that the number l of vertices in P is even and that P has l-1 edges. Furthermore, P is alternating and thus: $|X \cap P| = ½\, l-1$ and $|P \setminus X| = ½\, l$.
This implies:

$$\begin{aligned} |X'| = |X \Delta P| &= |X \setminus P| + |P \setminus X| \\ &= (|X - |X \cap P|) + |P \setminus X| \\ &= |X| - ½\, l+1 + 1/2\, l \\ &= |X| + 1 \end{aligned}$$

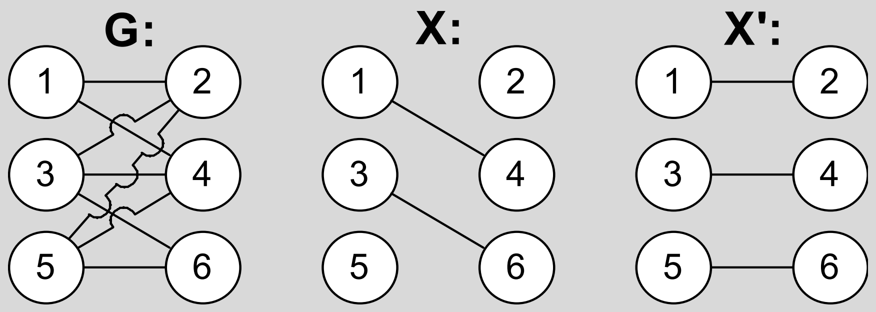
We consider the vertices $v \in V$ to show that X' is a matching.

- $v \notin P$ ➔ v is the endpoint of at most one edge in X and therefore also of at most one edge in X'.
- $v \in P$ ➔ For $v = i_1$, v is an endpoint of the edge $[i_1, i_2] \in X'$. For $v = i_e$, v is an endpoint of the edge $[i_{e-1}, i_e] \in X'$.
  Since i_1 and i_e were exposed in X, these two edges are the only edges in X' that are incident with i_1 and i_e respectively.
  For $v \in [i_2, \ldots, i_{e-1}]$ the edge that is matched by v changes under the transition from X to X'.

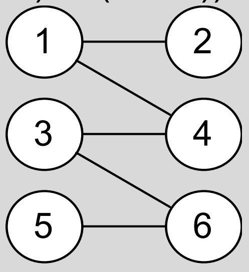
Therefore all vertices $v \in V$ are incident to at most one edge in X'.
➔ X' is a matching and larger than X.

"➔" Let X' be a maximal matching, i.e. $|X'| > |X|$. We consider the graph $G_\Delta := (V, X \Delta X')$

Example:



$G_\Delta := (V, X \Delta X') = (V, (X \setminus X') \cup (X' \setminus X))$



Every vertex in G_Δ is incident to 0, 1 or 2 edges since X and X' are matchings.
Thus G_Δ decomposes into subgraphs which are isolated vertices and alternationg path wrt. X or X', *or* which are alternating cycles wrt. X or X'.
In every alternating cycle C, $|X \cap C| = |X' \cap C|$.
Therefore, using $|X'| > |X|$, there must exist an alternating path P wrt. X with $|X' \cap P| > |X \cap P|$. This is only possible if both endpoints of P are exposed wrt. X, i.e. P is an ma-path wrt. X.

q.e.d.

The second theorem we need for the maximum cardinality matching algorithm is König's theorem.

2.2.2. König's Theorem

Let $\nu(G)$ denote the cardinality of a maximal matching in a given bipartite graph g.

Definition:
A node cover of $G = (L \dot{\cup} R, E)$ is a subset $K \subset (L \dot{\cup} R)$ such that $[l, r] \in E \Rightarrow l \in K$ or $r \in K$.

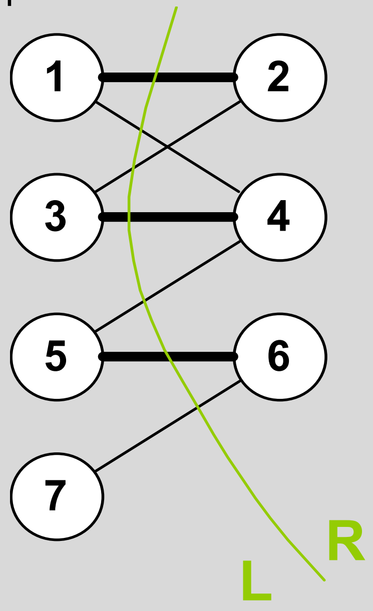
Let $\tau(G)$ denote the cardinality of a minimal covering of G, i.e.
$\tau(G) := \min \{|K| : K \text{ is a cover of } G\}$.

König's Theorem:
For any bipartite graph G the cardinality of a maximal matching is equal to the cardinality of a minimal covering

$$\nu(G) = \tau(G).$$

Proof:
"$\nu(G) \leq \tau(G)$":
If X is an arbitrary matching and if K is an arbitrary covering, then all $[i, j] \in X$ satisfy: $i \in K$ or $j \in K$.
It follows, that $|X| \leq |K|$ and therefore
$\nu(G) = \max \{|X|: X \text{ matching}\} \leq \min \{|K|: K \text{ covering}\} = \tau(G)$.
"$\tau(G) \leq \nu(G)$":
Let X be a maximal matching, i.e. $|X| = \nu(G)$, and let $L_X := \{l \in L : \exists\ e = [l, r] \in X\}$ be the set of vertices in L that are saturated by X, and define $L'_X := L \setminus L_X$.

Example:



$X = \{(1, 2), (3, 4), (5, 6)\}$
$L_X = \{1, 3, 5\}$
$L'_X = \{7\}$

We show: $\tau(G) \leq \nu(G)$

- Case 1: $L'_X = \{ \}$
  Then $K = L_X$ is a covering and $|X| = |K|$
- Case 2: $L'_X \neq \{ \}$
  Let V_X be the set of all vertices in $R \cup L$ that lie on an alternating path wrt. X starting in a vertex $l \in L'_X$ and let $R_x := R \cap V_X$.

  Example:
  $P = (7, 6, 5, 4, 3, 2, 1)$
  $V_X = \{1, 2, 3, 4, 5, 6, 7\}$
  $R_X = \{2, 4, 6\}$

  Then all $r \in R_X$ are saturated (else an ma-path would exist and X would not be maximal).
  Define: $K := (L_X \setminus V_X) \cup R_X$.

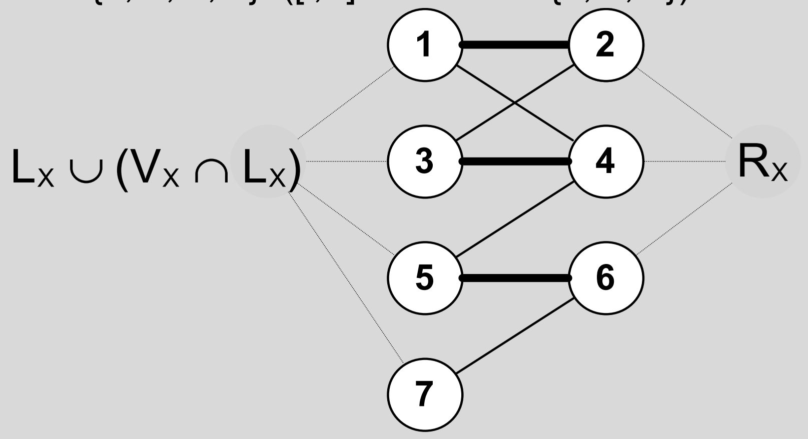
  Example:
  $K = \{2, 4, 6\}$

  We show that K is a covering satisfying $|K| \leq |X|$.
  a) If K was not a covering, there would exist an edge with one endpoint in $L \setminus (L_X \setminus V_X) = L'_X \cup (V_X \cap L_X)$ and the other endpoint in $R \setminus R_X$.
  This contradicts the definition of R_X and of alternating paths, which implies that
  $\forall l \in L'_X \cup (V_X \cap L_X) : ([l, r] \in E \Rightarrow r \in R_X)$.

Example:

$L'_X = \{7]$; $L_X = \{1, 3, 5\}$; $V_X = \{1, 2, 3, 4, 5, 6, 7\}$

$\Rightarrow \forall l \in \{1, 3, 4, 7\}: ([l, r] \in E \Rightarrow r \in \{2, 4, 6\})$



$\Rightarrow$ K is a cover.

b) $|K| \leq |X|$ since

$\forall r \in R_X$: $([l, r] \in X \Rightarrow l \in (V_X \cap L_X)$

$\forall l \in (L_X \setminus V_X)$: $([l, r] \in X \Rightarrow r \notin R_X)$

$\Rightarrow$Combining both cases we can conclude that $\tau(G) \leq \nu(G)$.

$\Rightarrow \tau(G) = \nu(G)$.

q.e.d.

2.3 Maximum Cardinality Matching Algorithm

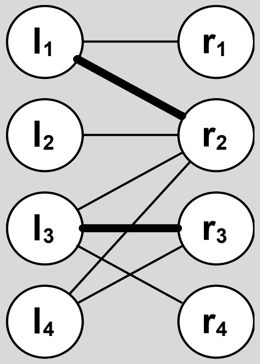
The theorems and their proofs show, that we can obtain a maximal matching algorithm and a minimal covering by iteratively searching for ma-path.
Orienting the edges e from L to R if $e \notin X$ and from R to L if $e \in X$, we can reduce the search for ma-path to a path problem in digraphs.
The following cardinality matching algorithm was developed by Hopcroft and Karp:

Maximum Cardinality Algorithm for Bipartite Graphs

(Input) $G = (L \cup R, E)$ bipartite graph; X matching (e.g. $X = \{ \}$)

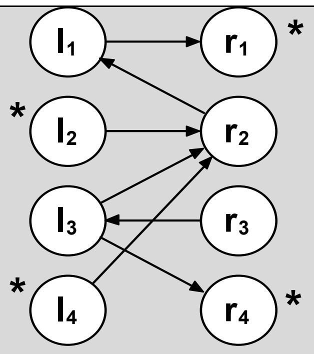
1) Orient all edges $e = [l, r] \in E$ from left to right, i.e. $e = (l, r)$ if $e \in E \setminus X$, and from right to left, i.e. $e = (r, l)$ if $e \in X$.
   Set $L' := L$, $R' := R$, $V_X = \{ \}$
2) If all vertices in L' are matched, then (STOP), X is a maximal matching.
   ELSE determine a minimal covering K (for example like in “case 2” of the proof of König’s theorem).
3) a) Choose an exposed vertex $l \in L'$.
   b) Set $V_l := \{v \in (L \cup R')$: v lies on an alternating path which only uses vertices from the set $(L \cup R')$ and that starts in L$\}$.
   As soon as $v \in V_l$ is exposed, set $X := X \,\Delta\, P$,where P is a path from l to v in G, (GOTO 1).
   If no vertex $v \in V_l$ is exposed, set $L' := L \setminus V_l$, $R' := R \setminus V_l$,
   $V_X := V_X \cup V_L$. (GOTO 2).

Example:
Input:



$G = (L \cup R, E)$
$X = \{(l_1, r_2), (l_3, r_3)\}$

Step 1:

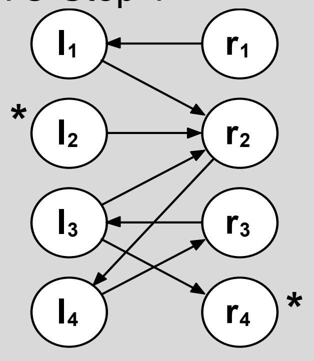


Direction of G
Exposed vertices are marked by *.
L’ = L = {l_1, l_2, l_3, l_4}
R’ = R = {r_1, r_2, r_3, r_4}
V_X = { }

Step 2: l_2 and l_4 in L’ are not matched.
Determine minimal covering K.
K = {r_1, r_2, l_3}

Step 3: a) Choose $l_4 \in$ L’ exposed.
b) V_l := {l_4, r_2, l_1, r_1} = P
X := (X \ P) $\cup$ (P \ X) = {(l_3, r_3), (l_4, r_2), (l_1, r_1)}
GOTO Step 1

Step 1:

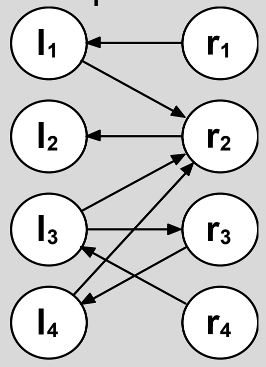


L’ = {l_1, l_2, l_3, l_4}
R’ = {r_1, r_2, r_3, r_4}
V_X = { }

Step 2: l_2 in L’ is not matched.
Determine minimal covering K.
K = {l_2, r_2, r_3, r_4}

Step 3: a) Choose $l_4 \in$ L’ exposed.
b) V_l := {l_2, r_2, l_4, r_3, l_3, r_4} = P
X := (X \ P) $\cup$ (P \ X) = {(l_1, r_1), (l_2, r_2), (l_4, r_3), (l_3, r_4)}
GOTO Step 1

Step 1:



L’ = L
R’ = R
V_X = { }

Step 2: All l $\in$ L’ are matched. (STOP)

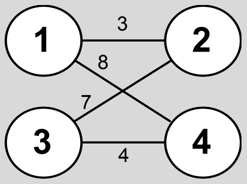
2.4 Developing a faster Algorithm

The algorithm from Hopcroft and Carp as well as the algorithm for the general case from Micali and Vazirani for computing maximal cardinality matchings have another important property. They can be used as approximation algorithms.
If M^* is such a maximum cardinality matching, then in $O(km)$ time for any k either algorithm will compute a matching M such that $|M^*| - |M| \leq n/k$.

The first step in the new algorithm is to sort all edges by their cost, which takes $O(m \log n)$ time.
Let $e_1, ..., e_m$ be the row of edges with non-decreasing cost.
Using a maximum cardinality matching algorithm, we compute l, which is the size of a maximum cardinality matching. Then we use binary search to find the minimum value i^* of i, such that the graph $G_i = (V, E_i = \{ e_1, ..., e_i\})$ contains a matching of size l.
To test if our guess i for i^* is high or low, we use a maximum cardinality matching subroutine.
The total running time of this method is $O(\sqrt{n}\ m \log n)$.

Example:

$V = \{1, 2, 3, 4\}$
$E = \{(1, 2), (1, 4), (2, 3), (3, 4)\}$



Sort: $e_1 = (1, 2)$; $e_2 = (3, 4)$; $e_3 = (2, 3)$, $e_4 = (1, 4)$
Compute l with maximal cardinality matching algorithm: $l=2$

- $i = 1$: $G_1 = \{(1, 2)\} \Rightarrow$ no matching.
- $i = 2$: $G_2 = \{(1, 2), (3, 4)\} \Rightarrow$ matching!

2.5 Enhancing the Algorithm's Performance

Now, that we got an idea how the algorithm works, we do better by using binary search to find a bottleneck matching that is only approximately of maximum cardinality.
For a parameter k we compute a value i' such that the graph $G_{i'}$ has a matching size of at least l – n/k, and $G_{i'-1}$ does not have a matching size of l.
To test whether a guess i for i' is high or low, we use the approximation version of our maximum cardinality matching algorithm.
We compute a matching M in $G_{i,}$ such that |M| is within n/k of maximum cardinality.
If |M| < l – n/k, the i is chosen to low.
The time to find i' using this approach is O(km log n). This gives us not only a value for i', but also a matching M of size at least l – n/k, all of whose edges have cost at most $c(e_{i'}) \leq c(e_{i^*})$.
Now this matching M can be augmented to form a bottleneck matching. To do so, we have to find a bottleneck augmenting path, augmenting M accordingly, and repeating this at most n/k – times.
The search for an augmenting path can be done in O(m) time by using a standard method (as described below) and making it incremental as in 1.5.

Before we do so, let us have a short look at the Ford-Fulknerson Method for finding augmenting path:

1) Mark nodes as matched or unmatched.
2) Start a search from each of the unmatched left nodes.
3) Traverse unmatched edges from left to right.
4) Traverse matched edges from right to left.
5) Stop successfully if an unmatched right node is reached.
6) Stop with failure if search terminates without finding an unmatched right node.

This method corresponds to "case 2" in the proof of König's Theorem.

To make this search now incremental, we initialise i to be equal ti i', let the search advance only along edges in E_i, and increase l by one each time the algorithm stops with failure.

2.6 Computing times

The total time to augment M to form a bottleneck matching is O(n m/k).
The overall time to find a bottleneck matching is O(mk log n + n m/k).
Choosing $k = \sqrt{\log n}$ gives us a time bound of $O(\sqrt{n \log n}\ m)$.

3. Further Developments

The bottleneck spanning tree algorithm from Tarjan and Gabow performs as we saw in 1.4 in O(m log*n) time.
In 1994 Punnen and Nair developed an algorithm that solves the problem in O(min (m log n log β(m, n), m log n log log C) time, with β(m, n) = min (i/$\log^{(i)}$n ≤ m/n).
For m ≥ O(n log n log log*n) their algorithm runs in O(m) time and hence is the best possible in this case.
The bottleneck maximum cardinality matching algorithm proposed by Tarjan and Gabow performs as stated in 2.6 in O ($\sqrt{n \log n}$ m) time.
Meanwhile several authors developed algorithms for this problem that all perform close to O($\sqrt[3]{n^2}$).