

various

Lecture Notes of Spring 2013

Algorithms I

University of Mannheim
2013

This script originates from the course "Algorithms I" at the University of Mannheim as lecture notes.

The accuracy of its content is not guaranteed and the author(s) do not assume responsibility for possible damages of any kind. This lecture notes are not an official document released by employees of the University of Mannheim, hence those do not assume responsibility, as well.

Many thanks to Ingo Bürk, who initially published the underlying LaTeX template [here](#). This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Germany License](#).

Contents

1 Elementary Notions about Graphs	4
1.1 Undirected Graphs	4
1.2 Directed Graphs	9
1.3 Implementation of Graphs on Computers	10
1.4 Trees	10
2 Euler Graphs and Hamilton Graphs	12
2.1 Euler Graphs	12
2.1.1 Euler 1736: Königsberger Brückenproblem	12
2.2 Hamiltonian Graphs	14
2.3 Bipartite Graphs	16
3 (Network) Flow Problems	19
3.1 Network Flow Problems	19
3.1.1 The Algorithm of Ford Fulkerson	24
3.1.2 The Edmonds-Karp Algorithm	27
3.1.3 The Algorithm of Dinic	27
4 Matching	36
4.1 Elementary Definitions	36
4.2 Matching in bipartite graphs	37
4.3 Stable matching (marriage)	41
4.4 Matching in general graphs	43
5 Networks with costs, respectively upper/lower bounds	50
5.1 Networks with costs	50
5.2 Networks with upper and lower bounds	54
6 NP-Completeness	57
6.0.1 Motivation/Examples	57
6.0.2 Introduction	57
6.0.3 Problem Types/Issues	58
6.1 Polynomial Time	59
6.1.1 Coding and concrete problems	60
6.2 Formal Language Representation	61
6.3 Polynomial Verification	62
6.4 NP-completeness and reducibility	63

7	Approximation Algorithms	67
7.1	Heuristic Approaches	67
7.1.1	Traveling Salesman	69
7.2	Probabilistic Analysis and randomized algorithms	71
7.2.1	Randomized Algorithms	73
4.3	Two processor scheduling	74

1

Elementary Notions about Graphs



1.1 Undirected Graphs

Definition 1.1

An undirected graph is a pair $G=(V, E)$ where V is a set of nodes and E is a set of edges, together with a function $i: E \rightarrow \gamma(v)$ such that $0 < |i(e)| \leq 2$.

If $u, v \in i(e)$ we call u, v endpoints of e .

If V and E are a finite set we call G a finite graph.

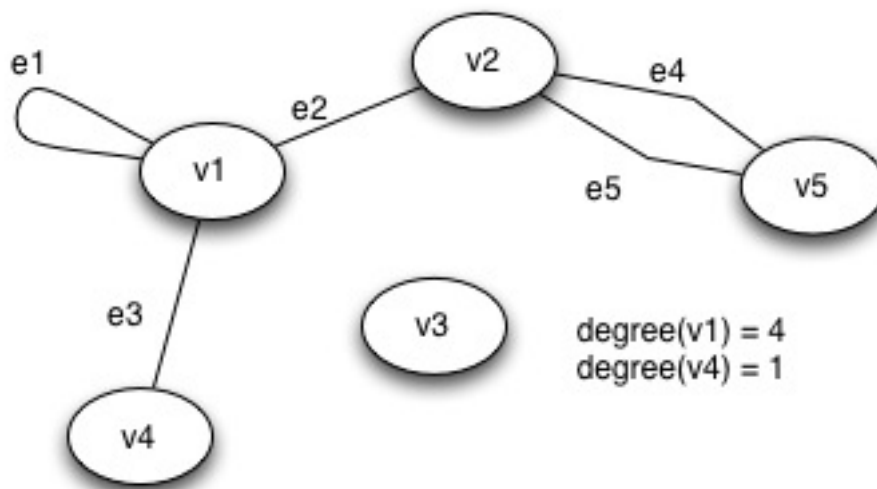
If $i(e_1) = i(e_2)$ we call e_1, e_2 parallel edges.

If $|i(e)| = 1$ we call e a loop.

The degree of a node v is the number of edges for which v is an endpoint where loops are counted twice.

If the degree of $v = 0$ then we call v isolated.

Example



Lemma 1.1

In a finite graph the number of nodes with odd degree is even.

Proof: $\sum_{i=1}^n \text{degree}(v_i) = 2 * |E|$

This is because we start with a graph, where each node is isolated. Then we insert one edge after another.

Case 1: $i(e) = x$ then the degree of x is increased by 2

Case 2: $i(e) = x, y$ then the degree of x and y are increased by 1

We assume that $v_1 \dots v_i$ have an even degree and $v_{i+1} \dots v_n$ have odd degree.

$$\sum_{k=1}^i \text{degree}(v_k) + \sum_{k=i+1}^n \text{degree}(v_k) = 2 |E|$$

$$\sum_{k=1}^i \text{degree}(v_k) \text{ is an even number}$$

$\sum_{k=i+1}^n \text{degree}(v_k)$ must be an even number and hence the number of nodes with odd degree must be even

$2 |E|$ is an even number

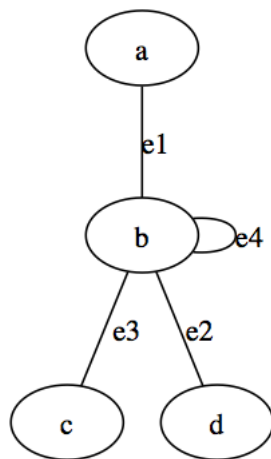
Definition 1.2

If $G=(V,E)$ is a graph and $v_1, v_2 \in V$ with $i(e) = \{v_1, v_2\}$ then we say that v_1, v_2 are neighbours.

A path in G is a sequence of edges e_1, e_2, \dots such that:

- i) e_i, e_{i+1} share an endpoint
- ii) if e_i is not a loop and neither the first nor the last edge. Then e_i shares one endpoint with e_{i-1} and the other with e_{i+1} [MS: does this make sense? sounds strange!][NW: Yes, draw it]

Example



A finite graph is graphically represented by: $v_0 - v_1 - v_2 - \dots - v_i$
 v_0 is called start point and v_i is called end point. The length of the path $e_1 \dots e_i$ is i .

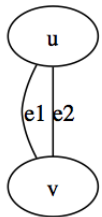
A cycle (circle) is a path where the end point coincide with the start point.

A path is called simple if every node in V occurs at most once.

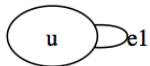
A cycle of length $\neq 2$ is called simple if every node except of the start/end node occurs at most once.

A cycle of length 2 (e_1, e_2) is called simple if $e_1 \neq e_2$ and if each node except for the start/end node occurs at most once.

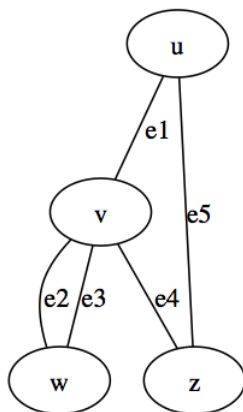
Example



Simple

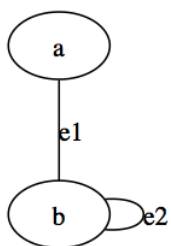


Simple



Not simple

A Graph is connected if for every pair of nodes (u, v) there is a path between u and v .



An infinite Graph has finite and infinite paths. Every path between two nodes is finite. The graph is connected. There are infinitely paths.

e.G. e_1 (finite)

or e_1, e_2

...

and there is also an infinite path $e_1, e_2, e_2, e_2, e_2, e_2, \dots$

Definition 1.3

Let $G(V, E)$ be a connected graph $a \in V$ is called a separation point (articulation point) if there are nodes v, w such that every path connecting v and w visits a . If G has such a point G is called [word missing - MS]

An edge is called bridge if there exist nodes v, w such that every path connecting v and w contains this edge.

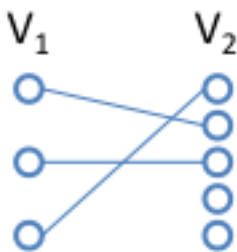
Example

Real-Life examples where separation points are important are in computer networks or the information distribution (flow of information) within a company. So a separation point can be regarded as a kind of coordinator between a and b .

Definition 1.4

Let $G = (V, E)$ be a graph without loops. If there exists $V_1, V_2 \subseteq V$ and $V_1 \cup V_2 = V$ such that $V_1 \cap V_2 = \emptyset$ and every edge e has one endpoint in V_1 and the other in V_2 , then we call G a **bipartite**.

Example



Definition 1.5

A directed graph is a pair $G = (V, E)$ where V is a set of nodes (vertices) and E is a set of edges together with a function $i : E \rightarrow V \times V$. If $i(e) = (v_1, v_2)$ then v_1 is called start point, v_2 is called end point.

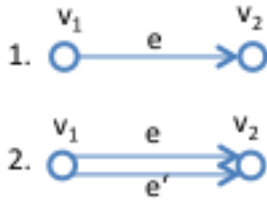
Graphically:

If $i(e) = (v_1, v_2)$ we draw 1.

If $i(e') = (v_1, v_2)$ then this indicates a second edge (2.).

If $i(e_1) = i(e_2)$ we call e_1, e_2 parallel.

If $i(e) = (v, v)$ then e is called a directed loop.



$g_{out}(v)$ is the number of edges that have starting point v .

$g_{in}(v)$ is the number of edges with endpoint v .

Lemma 1.2

$$\sum_{v \in V} g_{in}(v) = \sum_{v \in V} g_{out}(v)$$

Proof: We start with a graph without edges. Then we insert one after the other edges in E . Each edge contributes 1 to both sides of the equation.

1.2 Directed Graphs

Definition 1.6

A directed path is a sequence of edges e_1, e_2, \dots such that the end point of e_i is the start point of e_{i+1} .

A directed path $e_1 \dots e_k$ is called a (directed) cycle, if the start point of e_1 and the end point of e_k coincide.

A simple (directed) path is a path where every node occurs at most once.

A directed cycle is called simple if every node except for the start and end node occurs at most once.

Definition 1.7

A graph directed or undirected is called simple, if it does not contain parallel edges.

Definition 1.8

A directed graph is called strongly connected if for any pair of nodes (u, v) there is a directed path from u to v .

Let G be a directed graph $G = (V, E)$. $x, y \in V$ $x \sim y$ ([NW] does \sim mean are "connected"?) if there is a directed path from x to y and vice versa.

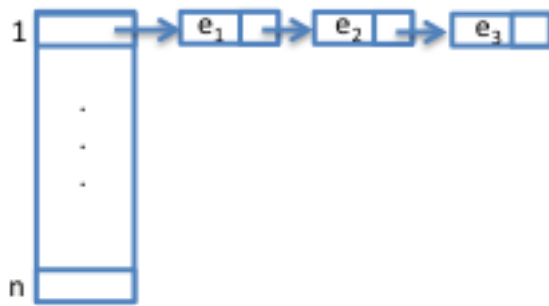
The equivalence classes of this relation $cVxV$ are called strongly connected components. (Analogously: Define connected components for undirected graphs)



We should know how the following terms are defined: reflexivity, symmetry, transitivity.

1.3 Implementation of Graphs on Computers

1. Adjacency Lists $V = 1 \dots n, E = e_1 \dots e_t$



2. Dynamically changing graphs:

e.g. multi user databases: Nodes \equiv transactions of user; Edges \equiv waiting situations

1	1	2
2	1	3
3	1	2

Graph is used to detect dead locks. Waiting arises when data are locked by a user that modifies these data.

$U_1 \text{ write}(d), \text{read}(d')$

$U_2 \text{ read}(d), \text{write}(d')$

1.4 Trees

Definition 1.9

An undirected graph is called a tree if it is connected and does not have simple cycles.

Let G be a directed graph, $G = (V, E)$. A node r is called root if every other node can be reached from r via a directed path.

A directed graph is called a tree if it has a root and the underlying undirected graph is a

tree.

Let G be a directed graph. A node is called source if $g_{in}(v) = 0$. v is called sink if $g_{out}(v) = 0$

Lemma 1.3

NW: what was lemma 1.3? the next one was 1.4 in my notes

Lemma 1.4

If $G = (V, E)$ is a directed graph without directed cycles, then there is always a source and sink.

We use this theorem to detect cycles

Proof: Source (sink analogously): Select an arbitrary node v_1 . If v_1 is a source we are done. If it is not, then there must be an edge e_1 leading to it $v_2 \xrightarrow{e_1} v_1$.

If v_2 is a source we are done. If not, there must be an edge e_2 leading to it $v_3 \xrightarrow{e_2} v_2 \xrightarrow{e_1} v_1$. We continue this process. It must stop because there are only finitely many nodes and if a node would appear once more on such a path, there would be a directed cycle.

2

Euler Graphs and Hamilton Graphs

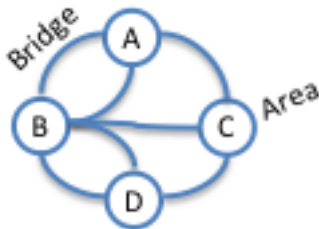
┌
TODO

└

2.1 Euler Graphs

2.1.1 Euler 1736: Königsberger Brückenproblem

Is it possible to do a round walk crossing every bridge exactly once?



Example 2.1



Definition 2.1

Let G be a finite undirected graph. A path $e_1..e_t$ is called a **euler path** if every edge in E occurs exactly once in the list.

A graph is a **euler graph** if it has a euler path.

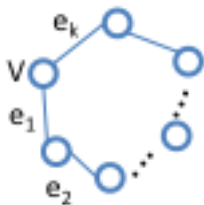
Theorem 2.1

A finite connected graph is a euler graph if and only if:

- i) It has either exactly two nodes of odd degree. or
- ii) All nodes have even degree.

In the last case the path is a cycle. In the first case no euler path is a cycle. Check is possible in linear time.

Proof: " $>$ " Let $G = (V, E)$ be a graph that has a euler path that is not a cycle. Let $|E| = k$
 $\circ \xrightarrow{e_1} \circ \xrightarrow{e_2} \dots \circ \xrightarrow{e_k}$ In this path v_1 and v_{k+1} have odd degree and all other nodes have even degree. Now consider the case that G has a euler cycle.



Hence every node has even degree.

" $<$ " Let G be a graph with exactly two nodes with odd degree, let this be a and b . We contradict a euler path as follows:

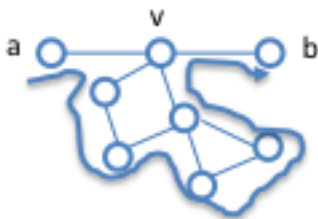
Start at node a and follow an edge on a. $a \rightarrow \circ \rightarrow \dots \rightarrow \circ \rightarrow b$

Case 1: All edges have been used \rightarrow done

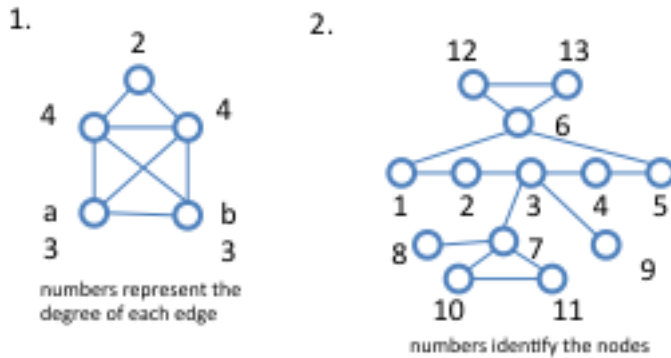
Case 2: Still edges unused. Then because G is connected there must be some node v on the path from which there is an unused edge. We construct a path starting from v as before. This path must end in v .

\Rightarrow Repeat until there are no more unused edges.

Analogously we proceed when the degree of all nodes is even.



Example



In the directed case a **directed Euler path** is a directed path on which every edge appears exactly once. Directed **Euler cycle** analogously.

Theorem 2.2

A finite directed graph is a **directed Euler graph** if and only if its underlying undirected graph is connected.

- i) There is one node a with $g_{out}(a) = g_{in}(a) + 1$ and another node b with $g_{in}(b) = g_{out}(b) + 1$ and for all other nodes v $g_{in}(v) = g_{out}(v)$. Or
- ii) For all nodes $g_{in}(v) = g_{out}(v)$ (**directed Euler cycle**)

2.2 Hamiltonian Graphs

Definition 2.2

Let $G = (V, E)$ be a graph. A **Hamiltonian cycle** C is a cycle on which every node $v \in V$ occurs exactly once. If G has a Hamiltonian cycle it is called **Hamiltonian**.

Example

1.



is hamiltonian!

2.



is hamiltonian!

3.



not hamiltonian!

The problem, given an arbitrary undirected graph: Is it **Hamiltonian**? \Rightarrow NP complete \Rightarrow no polynomial time algorithm is known and it is assumed there is no such.

One way out of the complexity issue is to derive conditions that can be tested explicitly and if they are satisfied the desired property is ensured.

Theorem 2.3

Let $G = (V, E)$ be an undirected finite graph without loops and without parallel edges. Let $|V| = n$. If for all $x, y \in V$ with $x \neq y$ and no edge with end points x, y the following holds:

$$\deg(x) + \deg(y) \geq |V| = n$$

Then G has a **Hamiltonian Cycle**.

Example



K_3



K_4

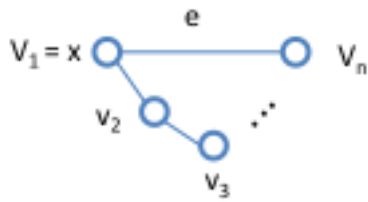


K_5

Proof: Assume there is a graph $G = (V, E)$ with $\deg(x) + \deg(y) \geq |V|$ for all x and y with $x \neq y$ and no edges between them, but is not **Hamiltonian**. Among all graphs with nodes in V , we choose one that has this property and has the maximal number of edges, we call graph $G_0 = (V, E_0)$. As the complete graph (every node is connected with every other node) is **Hamiltonian**, we know there must be an edge e connecting some x and y and $e \in E_0$.

We add edge e to the graph and obtain a new graph $G_1 = (V, E_1)$ that still satisfies the degree conditions and must be **Hamiltonian** because G_0 was the one with the largest number of edges.

We know that the **Hamiltonian cycle** must contain the edge e .



$v_i \neq v_j$ for $i \neq j$

$S = \{v_i : 1 \leq i \leq n \text{ x,y are connected with an edge in } E_0\}$

$T = \{v_i : 1 \leq i \leq n \text{ there is an edge between y and } v_i \text{ in } E_0\}$

Observation:

i) $y = v_n \in S \cup T$

ii) $|S \cup T| < |V| = n$

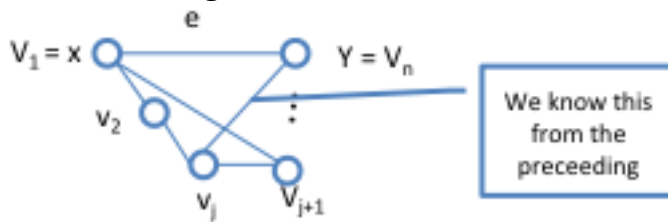
iii) $\deg(x) = |S|$

$\deg(y) = |T|$

Hence $S \cap T \neq \emptyset$, let $v_j \in S \cap T$ hence there is an edge between x, v_{j+1} and an edge between y, v_j .

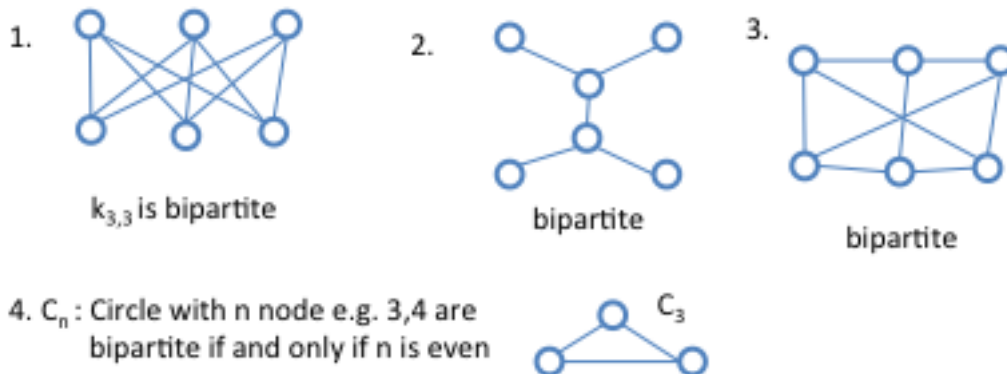
Now remove edge e and there is a Hamiltonian left.

Cost of checking the condition $O(M^2)|E| \leq |V|^2$



2.3 Bipartite Graphs

Example



Theorem 2.4

Let $G = (V, E)$ be a connected undirected graph without loops and parallel edges. G is bipartite if and only if it does not contain any cycle of odd length.

Corollary: All trees are bipartite

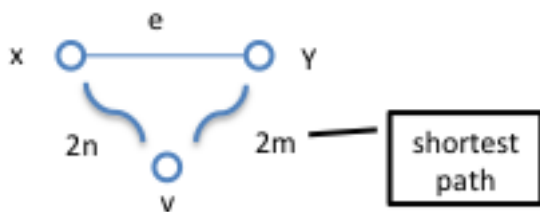
Proof: \Rightarrow IF G contains a cycle of odd length then it is not bipartite. \Leftarrow Let G not have any cycle of odd length we choose node v .

$V_1 = \{u \in V \text{ a shortest path between } u \text{ and } v \text{ is of odd length}\}$

$V_2 = \{u \in V \text{ a shortest path between } u \text{ and } v \text{ is of even length}\}$

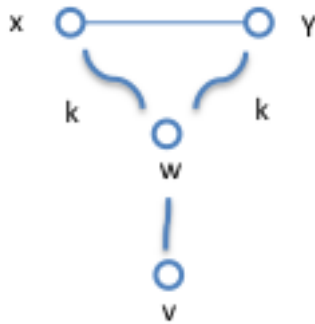
$V = V_1 \cup V_2$ (disjoint union)

Claim: There is no edge e with both endpoints in V_1 respectively V_2 . Assume there is an edge e with both endpoints in V_1 . Let the end points be x, y



$$2m \leq 2n + 1 \text{ and } 2n \leq 2m + 1 \Rightarrow m = n$$

Let $P(x)$ a shortest path from v to x , analogously $P(y)$ let w be the last node on the paths starting at v that lies on both paths.

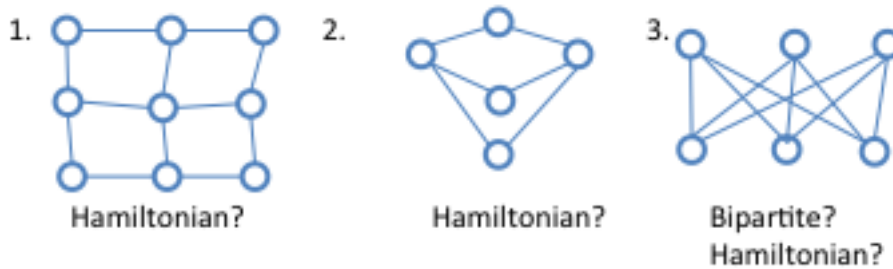


The length of the path from w to x coincides with the length of path from w to y .
 The circle $w - x - y - w$ is of odd length i.e. $2k + 1 \Rightarrow$ contradiction!

Corollary 2.5: A bipartite graph with an odd number of nodes cannot be [Hamiltonian](#)

Proof: Assume if were Hamiltonian then there is a cycle where node appears exactly once.
 This cycle is of odd length \Rightarrow contradicts Theorem 2.4

Example 2.2



\Rightarrow We have two theorems to check:

- i) Count degrees
- ii) Corollary 2.5

3

(Network) Flow Problems

┌
 TODO

└

3.1 Network Flow Problems

Example 3.1

Example: Oil field + transportation

Definition 3.1

A network N consists of

- i) A finite directed graph $G = (V, E)$ without loops and parallel edges
- ii) a function $c : E \rightarrow \mathbb{R}^+$, which assigns a capacity to each edge
- iii) two designated nodes s and t , called **source** and **sink**

Short: $N = (G, c, \{s, t\})$

Definition 3.2

Let $N = (G, c, \{s, t\})$ be a network. A flow function on N is a function $f : E \rightarrow \mathbb{R}$ such that

- $0 \leq f(e) \leq c(e), \forall e \in E$
- $\alpha(v) := \{e : \text{endpoint of } e \text{ is } v\}, v \in V$
 $\beta(v) := \{e : \text{startpoint of } e \text{ is } v\}, v \in V$

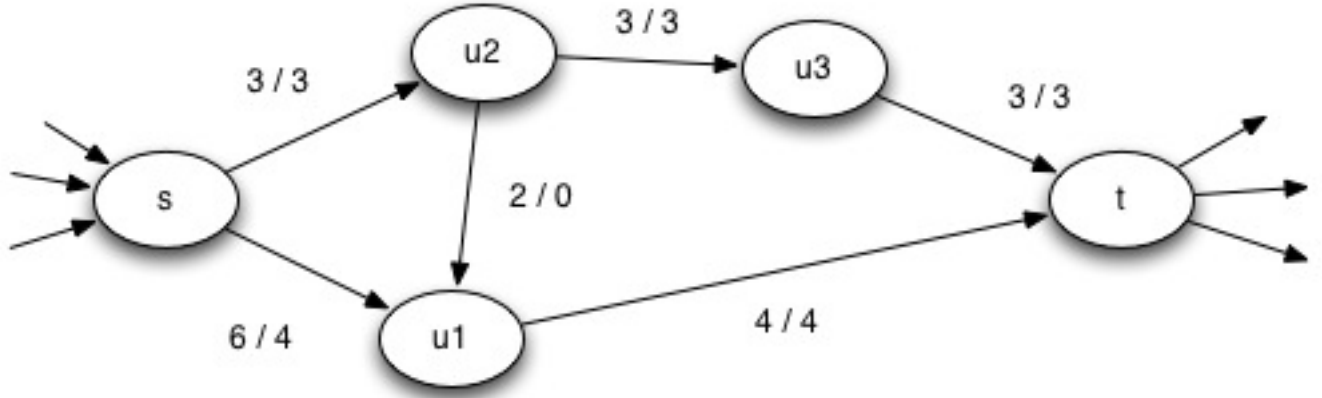
For every $v \in V \setminus \{s, t\}$

$$\sum_{e \in \alpha(v)} f(e) = \sum_{e \in \beta(v)} f(e)$$

This is called "**conservation function**".

The **total flow** of the flow function is given by $F = \sum_{e \in \alpha(t)} f(e) - \sum_{e \in \beta(t)} f(e)$

Example 3.2



Notion: 6/4 describes the capacity and the flow of an edge. In this example the capacity of the edge is 6 and the flow is 4.

Problem:

Given an arbitrary network N , find a flow function f , where the total flow F is maximal.

Definition 3.3

Let $N = (G, c, \{s, t\})$ be a network. Let $S \subseteq V$ with $s \in S, t \notin S$

$\bar{S} = V \setminus S$ (i.e. $t \in \bar{S}$)

$E_{S\bar{S}} = \{e : \text{all edges with starting point in } S \text{ and endpoint in } \bar{S}\}$

$E_{\bar{S}S} = \{e : \text{all edges with start point in } \bar{S} \text{ and end point in } S\}$

$E_{S\bar{S}} \cup E_{\bar{S}S}$ is the **cut** defined by S .

The capacity of a cut defined by S : $c(S) = \sum_{e \in E_{S\bar{S}}} c(e)$

Lemma 3.1

Let $N = (G, c, \{s, t\})$ be a network, $f : E \rightarrow \mathbb{R}$ be a flow function then for any $S \subseteq V$ with $s \in S, t \notin S$:

$$F = \sum_{e \in E_{S\bar{S}}} f(e) - \sum_{e \in E_{\bar{S}S}} f(e)$$

Proof.

$$F = \sum_{e \in \alpha(t)} f(e) - \sum_{e \in \beta(t)} f(e)$$

$$0 = \sum_{e \in \alpha(v)} f(e) - \sum_{e \in \beta(v)} f(e); \forall v \in \bar{S} \setminus \{t\}$$

We add all these equations up. Left hand side: F remains. Right hand side: Let $x \xrightarrow{e} y$ be an edge. We need to consider 4 cases:

- i) $x, y \in S$, then the value $f(e)$ does not occur in the summation
- ii) $x, y \in \bar{S}$, then $f(e)$ occurs one time positive in the summation, namely for y
 $f(e)$ occurs one time negative in the summation, namely for x
- iii) $x \in S; y \in \bar{S}$, $f(e)$ occurs positive for y and nowhere else and $e \in E_{S\bar{S}}$
- iv) $x \in \bar{S}; y \in S$, then $f(e)$ occurs negative for x and nowhere else and $e \in E_{\bar{S}S}$

This leads to the following equation:

$$F = \sum_{e \in E_{S\bar{S}}} f(e) - \sum_{e \in E_{\bar{S}S}} f(e)$$

Only case iii) and iv) contribute. □

Lemma 3.2

For every flow function f with total flow F and any set $S \subseteq V$, $s \in S$, $t \notin S$

$$F \leq c(S)$$

Proof. From lemma 3.1 we know

$$F = \sum_{e \in E_{S\bar{S}}} f(e) - \sum_{e \in E_{\bar{S}S}} f(e) \leq \sum_{e \in E_{S\bar{S}}} f(e) \leq \sum_{e \in E_{S\bar{S}}} c(e) = c(S)$$

□

Korollar 3.1 Max Flow - Min Cut Statement

If $F = c(S)$ then the total flow F is maximal and the capacity of the cut defined by S is minimal.

Proof. Let $F = c(S)$, consider another flow function f' with total flow F' .

- i) $F' \leq c(S)$ (Lemma 3.2) // $F' \leq c(S) = F$

Hence, f is a flow function with maximal total flow.

- ii) Let S' with $s \in S'$, $t \notin S'$ be given. $c(S) = F \leq c(S')$. Hence the capacity $c(S)$ is minimal among all other capacities.

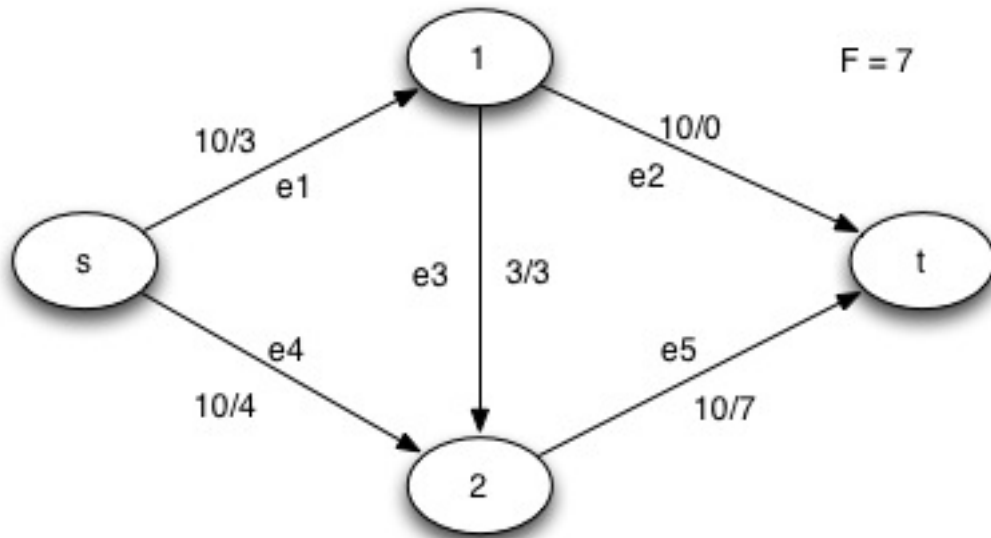
□

An **augmenting path** is a simple path from s to t , that is not necessarily directed. And for which the following two cases hold: Let e be an edge on this path:

i) $s \rightarrow \circ \rightarrow \circ \rightarrow \dots \rightarrow \underset{s_i}{\circ} \xrightarrow{e} \underset{s_{i+1}}{\circ} \dots \circ \xrightarrow{\quad} t$ then we request that $f(e) < c(e)$

ii) $s \rightarrow \dots \circ \xleftarrow{e} \underset{s_{i+1}}{\circ} \dots \circ \xrightarrow{\quad} t$ then we request that $f(e) > 0$

Example 3.3



Which of the following is an augmenting path?

- $e_1 e_2$
- $e_1 e_3 e_5$
- $e_4 e_3 e_2$
- $e_4 e_5$

Solution:

The first, third and fourth example are augmenting paths. The second path violates case 1.

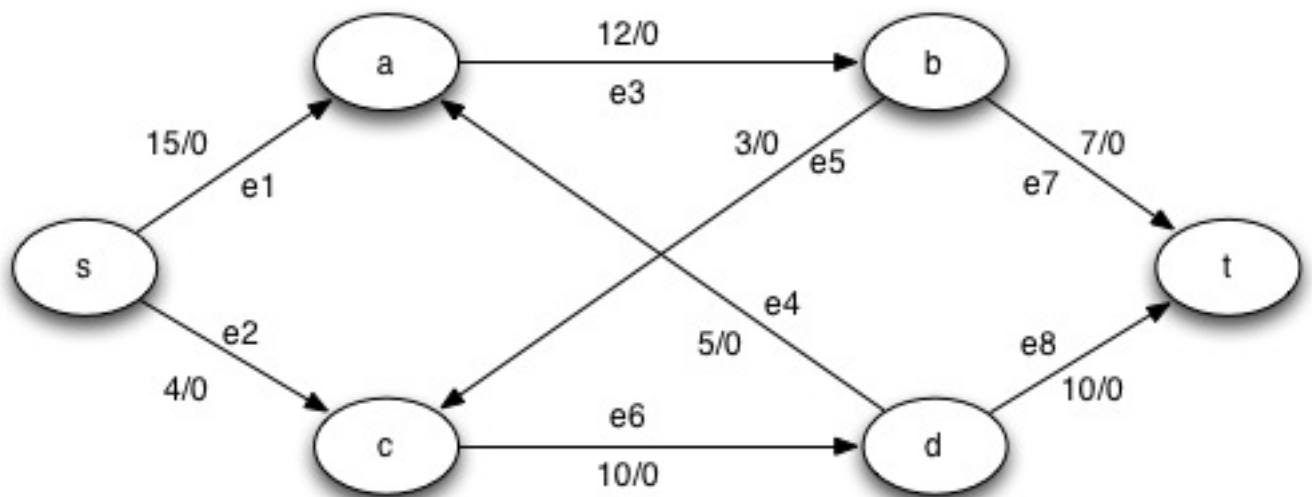
We use $e_4 e_3 e_2$ to improve the flow function as follows:

- For forward edges $e : c(e) - f(e)$:
 - $e_4 : 6$
 - $e_2 : 10$
- For backward edges $e : f(e)$
 - $e_3 : 3$

We chose the minimum from the values and add the value to the flow of forward edges and subtract it from backward edges. The flows of the edges change as follows:

- $e_4 = 10/7$
- $e_2 = 10/3$
- $e_3 = 3/0$

Example 3.4



Augmenting path:

$$s * \xrightarrow{e_2} c * \xrightarrow{e_6} d * \xrightarrow{e_4} a * \xrightarrow{e_3} b * \xrightarrow{e_7} t *$$

Compute deltas:

- $\Delta_{(e_2)} = 4$
- $\Delta_{(e_6)} = 10$
- $\Delta_{(e_4)} = 5$
- $\Delta_{(e_3)} = 12$
- $\Delta_{(e_7)} = 7$

The minimum Δ is 4, so the flow of the edges will be increased by 4.

- $e_2 = 4/4$
- $e_6 = 10/4$
- $e_4 = 5/4$
- $e_3 = 12/4$
- $e_7 = 7/4$

The next steps or paths would be:

- $s \rightarrow a \rightarrow b \rightarrow c \rightarrow d \rightarrow t$
- $s \rightarrow a \rightarrow b \rightarrow t$
- $s \rightarrow a \rightarrow d \rightarrow t$

The application of this paths leads to a new flow: $F = 14$.

3.1.1 The Algorithm of Ford Fulkerson

See handout for a description of the algorithm.

Lemma 3.3

When executing a step in the algorithm, the actual function f is a flow function.

Proof. The assumption is obviously true for step 1 because $f \equiv 0$ is a flow function. It is obviously true for steps 2, 3 and 5, too, because f is not modified.

Step 4:

Let f be a flow function when we enter step 4. We have to show that after performing step 4, the newly calculated function f is still a flow function.

Let f_{old} be the function with which we enter step 4 and f_{new} the newly calculated one. f_{old} is a flow function. Hence,

$$\sum_{e \in \alpha(v)} f_{old}(e) = \sum_{e \in \beta(v)} f_{old}(e); \forall v : v \neq s, v \neq t$$

Let $s \rightarrow v_0 \rightarrow v_1 \dots v_{f_{e-1}} \rightarrow v_{f_e} \rightarrow t$ be an augmenting path used in step 4. By definition of $\Delta f_{new}(e) < c(e)$ and $f_{new}(e) > 0$.

For step 4: Let $s = v_0 \rightarrow \dots \rightarrow v_2 = t$ be the path along which we achieved the marking. Only the flow value of the edges along this path is modified, so we have to check only the edges respectively nodes along this path. We have to check:

- i) $0 \leq f_{new}(e) \leq c(e) \forall e: e \text{ edge on the path}$
- ii) $\sum_{e \in \alpha(v)} f_{new}(e) = \sum_{e \in \beta(v)} f_{new}(e), \forall v, v \text{ on the path}, v \neq s, v \neq t$

The check:

- i) 1 holds by the definition of Δ
- ii) Let $v_i, v_i \neq s, v_i \neq t$ be a node on the path:
 - a) $\xrightarrow{e_i} v_i \xrightarrow{e_{i+1}}, e_i \in \alpha(v_i), e_{i+1} \in \beta(v_i)$ for both edges f_{new} is obtained from f_{old} by adding Δ , so 2 holds in this case

- b) $\xrightarrow{e_i} v_i \xleftarrow{e_{i+1}}, e_i, e_{i+1} \in \alpha(v_i)$. One contributes Δ , the other contributes $-\Delta$, so 2 holds for v_i
- c) $\leftarrow v_i \leftarrow$ analogously
- d) $\leftarrow v_i \rightarrow$ analogously, too

□

Lemma 3.4

If the algorithm by Ford-Fulkerson terminates then the determined flow function has maximal total flow.

Proof. If the algorithm terminates, then it does so in step 3. That means we started labeling but did not reach t . Let S be the set of nodes that were marked in the last round. Then $s \in S$ and $t \notin S$. $\tilde{S} = V \setminus S$. Let $x \xrightarrow{e} y$ be an edge in $E_{S\tilde{S}}$, this means x is labelled, y is not labelled. So, we know that $f(e) = c(e)$ because otherwise we could have marked y .

If $x \xrightarrow{e} y$ is an edge in $E_{\tilde{S}S}$ ($x \in \tilde{S}, y \in S$). So, y is labelled and x is not. Then we can conclude that $f(e) = 0$, otherwise we could have marked x .

$$F \stackrel{(3.1)}{=} \sum_{e \in E_{S\tilde{S}}} f(e) - \underbrace{\sum_{e \in E_{\tilde{S}S}} f(e)}_{=0} = \sum_{e \in E_{S\tilde{S}}} c(e) = c(e)$$

f is a flow function with maximal total flow.

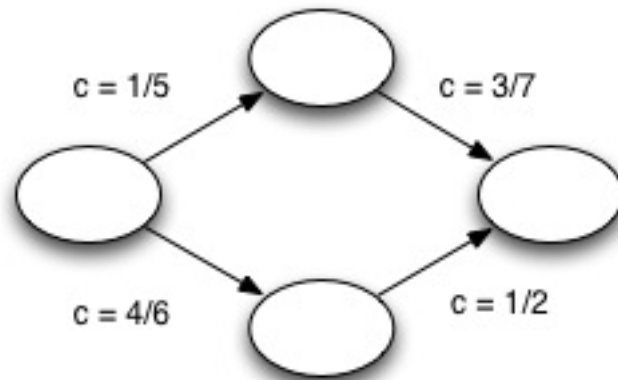
□

Termination

Lemma 3.5

If $c : E \rightarrow \mathbf{N}$ then the algorithm terminates.

Proof. This holds because the algorithm starts with $f \equiv 0$ and the total flow F is increased by Δ and Δ is a natural number as $c(e) \in \mathbf{N} \forall e$ and because $F \leq c(s)$ for all S with $s \in S, t \notin S$ i.e. F is bounded.



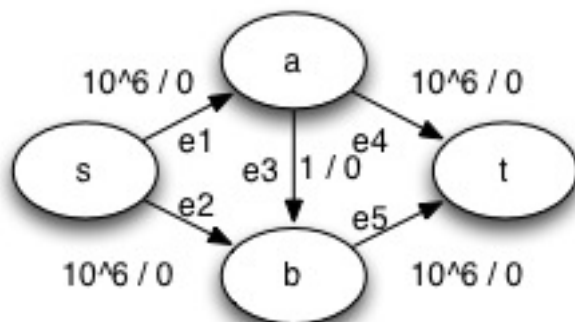
What about $c : E \rightarrow \mathbf{R}$?

Turn the capacities into natural numbers, calculate the results and divide them later. \square

Proposition 3.1

The example with $c : E \rightarrow \mathbf{R} \setminus \mathbf{Q}$ is an example, for which we can apply the algorithm in a way that it does not terminate.

Example 3.5



Maximal total flow: $2 \cdot 10^6$

Selecting the augmenting paths:

- $e_1 e_4$

- e_2e_5

This would solve the problem within two steps, but...

Choosing the following augmenting paths:

- $e_1e_3e_5$ with $\Delta = 1$ and
- $e_2e_3e_4$ with $\Delta = 1$

would lead to $2 * 10^6$ rounds to solve the problem.

3.1.2 The Edmonds-Karp Algorithm

An algorithm which improves Ford Fulkerson by finding the shortest augmenting path and therefore reducing the runtime bound.

Theorem 3.1

If we use breadth-first-search when labelling and always select the shortest augmenting path, then the algorithm terminates and uses $O(|V|^3 * |E|)$ steps for any $c : E \rightarrow \mathbf{R}$ (where it is assumed that any real number can be manipulated in one step).

3.1.3 The Algorithm of Dinic

Definition 3.4

Let e be an edge between u and v with flow value $f(e)$. e is called **useful** from u to v if

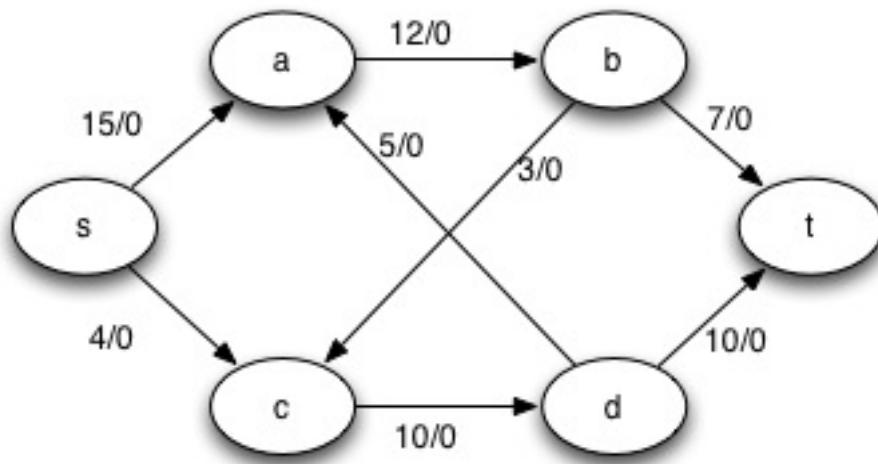
- i) $u \xrightarrow{e} v$ and $f(e) < c(e)$
- ii) $u \leftarrow v$ and $f(e) > 0$

Let $G = (V, E)$ be the directed graph for the network and f a flow function for the network. A **layering** for the network is defined as follows:

- i) $V_0 = \{s\}, i \leftarrow 0$
- ii) $T := \{v \in V, v \notin V_j, j \leq i \text{ and there is an useful edge from a node in } V_i \text{ to } v\}$
- iii) If $T = \emptyset$ then the actual flow function has maximal total flow and the algorithm stops.
- iv) If $t \in T$ then put $l := i + 1, V_l = \{t\}$ and stop the algorithm.
- v) $V_{i+1} := T, i \leftarrow i + 1$ and go to step 2

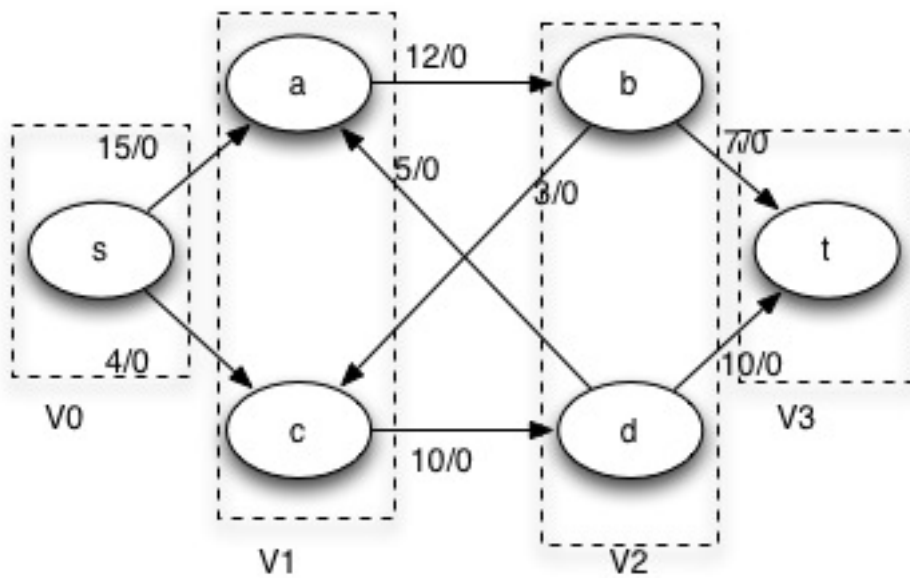
$E(i) = \{e : e \text{ is an useful edge between some node in } V_{i-1} \text{ and } V_i\}$

Example 3.6



Layers:

- $V_0 = \{s\}$
- $V_1 = \{a, c\}$
- $V_2 = \{b, d\}$
- $V_3 = \{t\}$



Theorem 3.2

When the algorithm for layering stops in step 3, then the actual flow function has maximal total flow.

Proof. Determine a set S with $F = c(S)$. What S ?

$$S = \bigcup_{i=0}^i V_j, \bar{S} = V \setminus S, s \in S, t \notin S$$

For any edge $u \xrightarrow{e} v, e \in E_{S\bar{S}}$, we know $f(e) = c(e)$ because otherwise T would not be \emptyset . And for any edge $u \xleftarrow{e} v \in E_{\bar{S}S}$ we know $f(e) = 0$ and continue as for lemma 3.5. \square

Definition 3.5

Associate capacities \bar{c} to the edges in $E(i)$. Let c be an edge in $E(i)$, $u \xrightarrow{e} v$

- i) if $u \in V_{i-1}, v \in V_i$, then $\bar{c}(e) := c(e) - f(e)$
- ii) $u \in V_{i-1}, v \in V_i, u \xleftarrow{e} v$, then $\bar{c}(e) := f(e)$

Remark:

$\bar{c}(e) > 0$ in both cases as only useful edges are considered in $E(i)$

In the new network we look for a flow function \bar{f} such that on any path from s to t

$$s - v_1 - v_2 - \dots - t, v_j \in V_j, e_j \in E_j$$

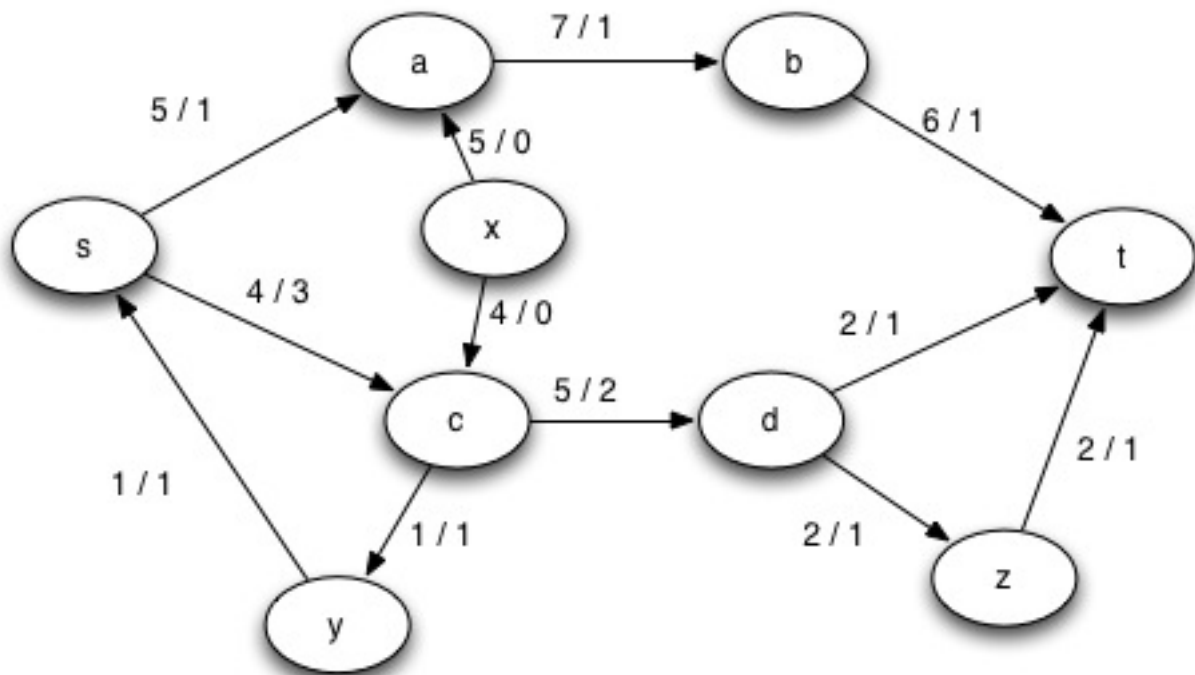
there is at least one edge e with $\bar{f}(e) = c(e)$.

Given such a function \bar{f} (see handout: Dinic's algorithm) we modify the original flow function f_{old} as follows:

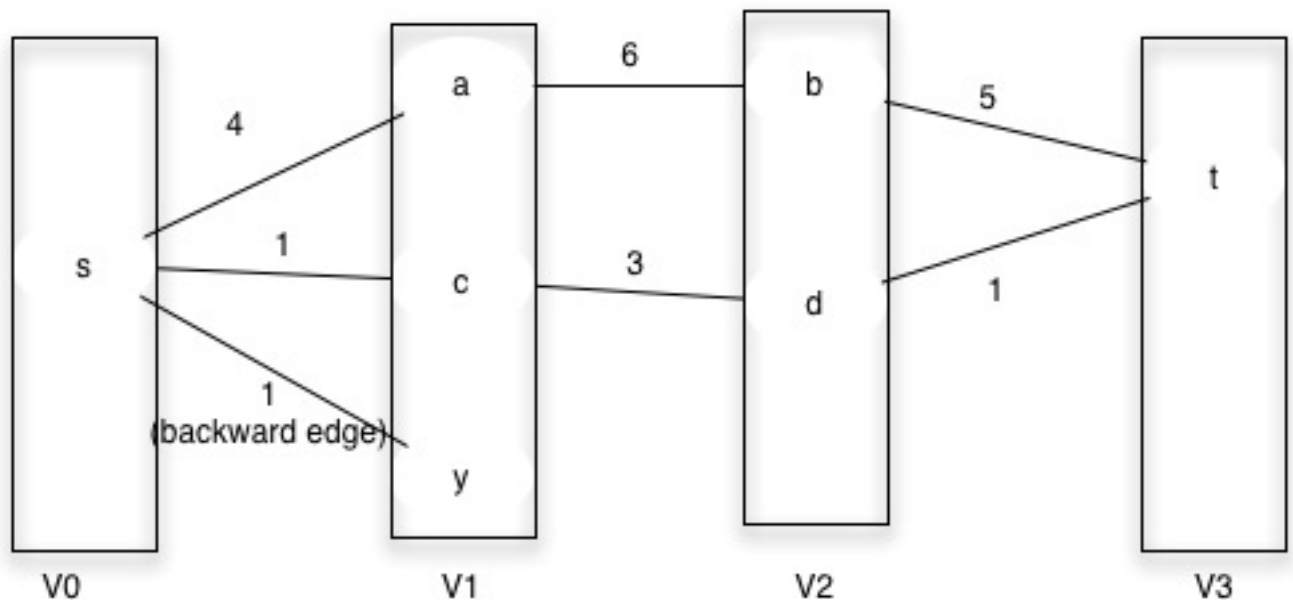
- i) if $u \xrightarrow{e} v$ with $u \in V_{i-1}, v \in V_i$ then $f_{new}(e) := f_{old}(e) + \bar{f}(e)$
- ii) if $u \xleftarrow{e} v$ with $u \in V_{i-1}, v \in V_i$ then $f_{new}(e) := f_{old}(e) - \bar{f}(e)$

Algorithm of Dinic:

- i) Initialize with a flow function f , e.g. $f(e) \equiv 0 \forall e \in E$
- ii) Construct a layering with respect of f (remember: halt twice)
- iii) Determine \bar{f}
- iv) From f and \bar{f} determine the new flow function
- v) Go to step 2

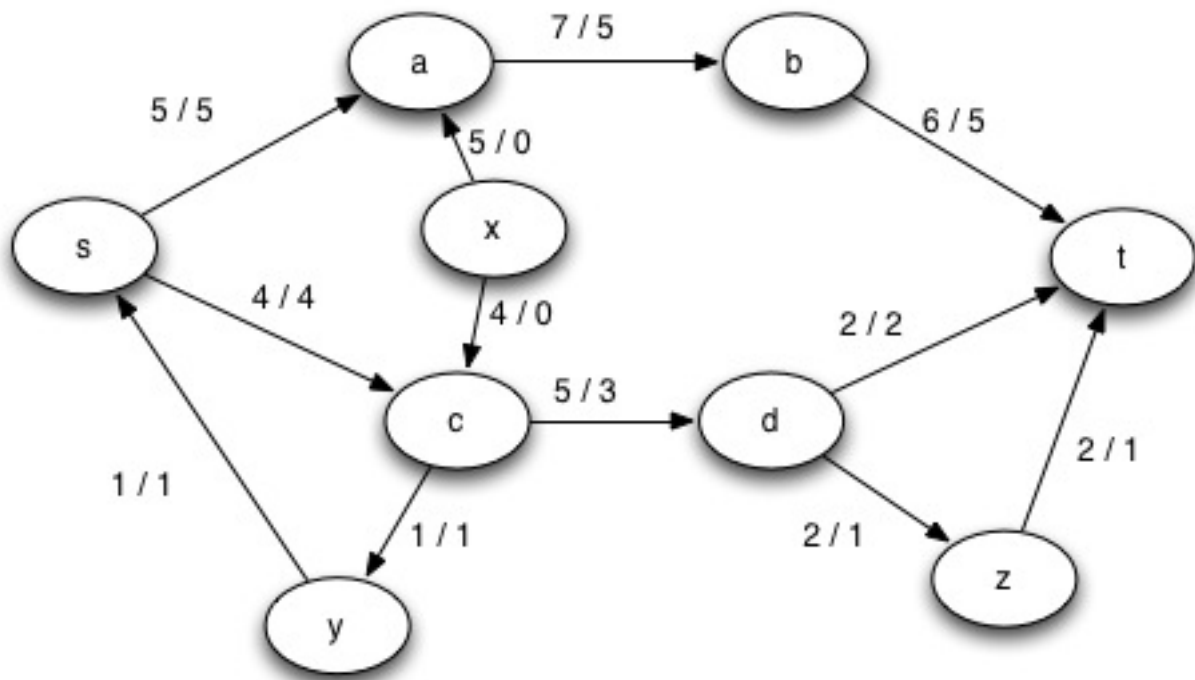
Example 3.7


Given the example above the total flow of the network is $F = 3$. We will proceed with the first layering as follows:

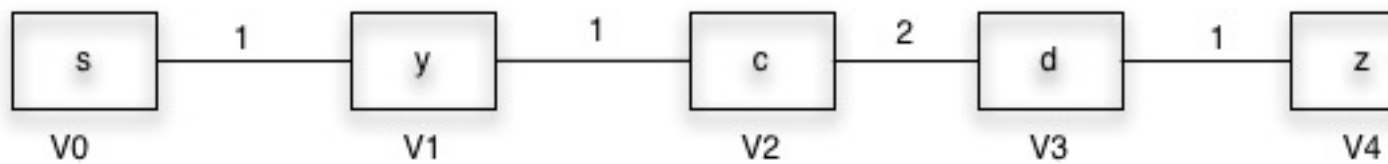


The nodes x, z disappeared while layering.

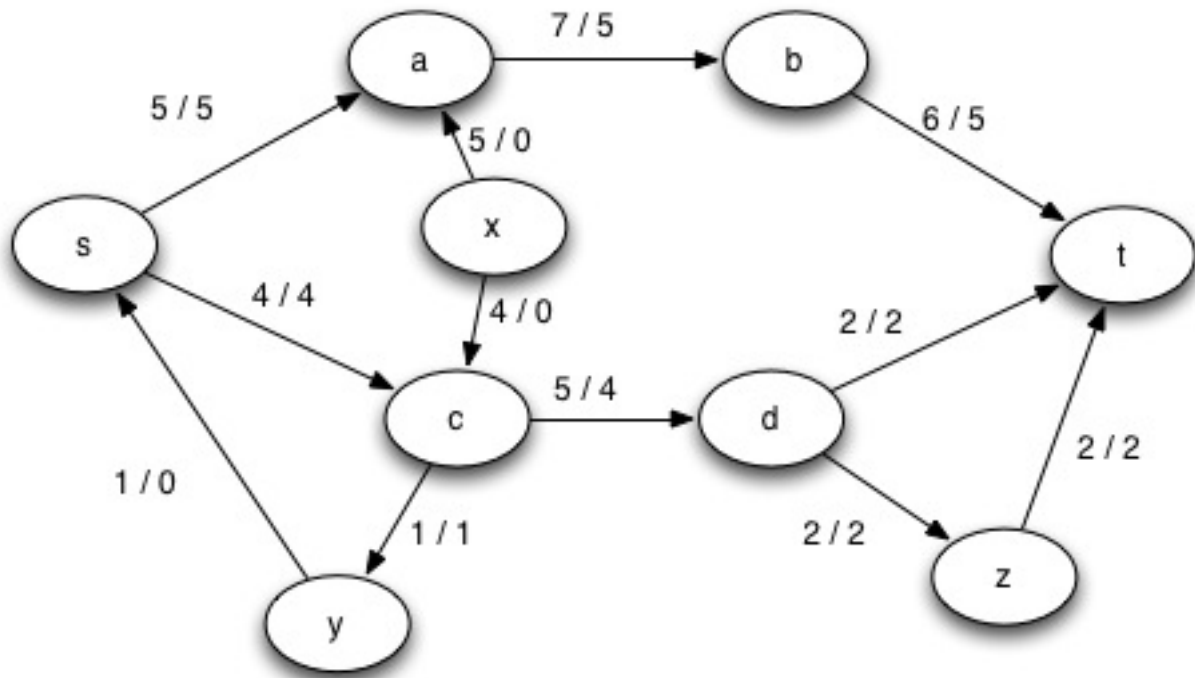
Determine \tilde{f} : there are two paths from s to t , find \tilde{f} for each of them that saturates at least one edge on the path. The lower path yields a flow function that transports one unit from s to t . The upper path yields a flow function that transports four units from s to t .



After determining \bar{f} we have to update the flows on our graph and gain a new total flow $F = 8$.
Second layering:



There is only one path from s to t. A flow function \bar{f} that saturates at least one edge is the one that sends one unit along this path.



We update again the flows of the used edges and the total flow increases by one, so we have: $F = 9$.

Third layering

$V_0 = \{s\}$ can not be continued as there are no more useful edges from V_0 .

Lemma 3.6

Let N be a network with flow function f . \bar{f} is the flow function of the layered network. Then

- i) the new calculated function f is a flow function and
- ii) the new total flow is obtained by adding the old total flow and \bar{F} ($F_{old} + \bar{F} = F_{new}$).

Proof. For i) we have to show that $0 \leq f_{new}(e) \leq c(e) \forall e \in E$ and that the flow condition holds for every node.

- Show $0 \leq f_{new}(e) \leq c(e)$:
 - i) Observation: $\forall e \notin \bigcup E(i)$ nothing has changed

ii) Let $e \in E(i)$, that is $u \xrightarrow{e} v, u \in V_{i-1}, v \in V_i$ and e is useful.

– Case 1:

$$u \xrightarrow{e} v$$

$$\bar{f}(e) \leq \bar{c}(e) \stackrel{Def.}{=} c(e) - f(e)$$

$$f_{new}(e) = f(e) + \bar{f}(e) \leq f(e) - \bar{f}(e) = c(e)$$

M: Bitte prÃ¼ft mal jemand den Teil mit $f(e) - \bar{f}(e)$!

– Case 2:

analogously

• Show $0 \leq f_{new}(e) \forall e \in E$:

i) Observation: all edges $\notin \cup E(i)$ are not affected

ii) Let $e \in E(i), u \xrightarrow{e} v, u \in V_{i-1}, v \in V_i$

– Case 1: $u \xrightarrow{e} v$

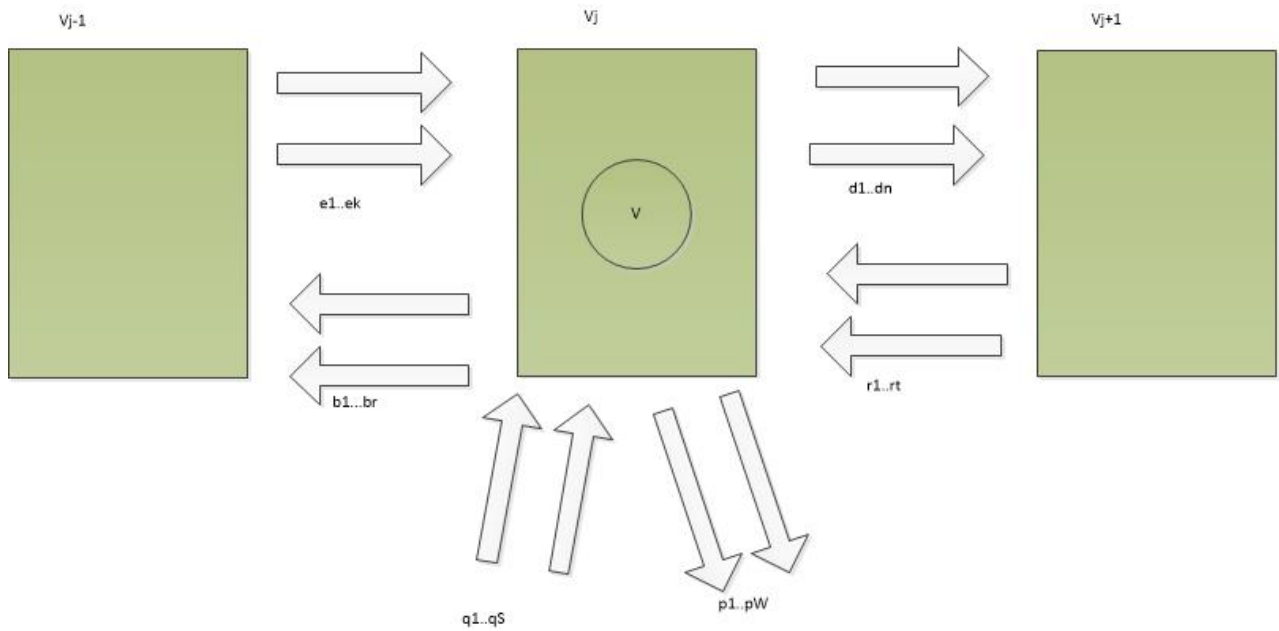
$$f_{new}(e) = f_{old}(e) + \bar{f}(e) \geq 0$$

– Case2: $u \leftarrow v$

$$f_{new}(e) = f_{old}(e) - \bar{f}(e) \geq f_{old}(e) - \bar{c}(e) \stackrel{Def.}{=} 0$$

Flow condition:

i) Observation: Nodes that do not appear in a layer are unaffected



ii) \bar{f} is a flow function in the layered network.

$$a) \sum_{e \in E \cup B} \bar{f}(e) = \sum_{e \in D \cup R} \bar{f}(e)$$

Because f_{old} is a flow function, we know that

b) $\sum_{e \in E \cup Q \cup R} f_{old}(e) = \sum_{e \in B \cup F \cup D} f_{old}(e)$ (original direction of edges)

For f_{new} we get

$$\begin{aligned} \sum_{e \in E \cup P \cup R} f_{new}(e) &= \sum_{e \in E} f_{new}(e) + \sum_{e \in Q} f_{new}(e) + \sum_{e \in R} f_{new}(e) \\ &= \sum_{e \in E} (f_{old}(e) + \bar{f}(e)) + \sum_{e \in Q} f_{old}(e) + \sum_{e \in R} (f_{old}(e) - \bar{f}(e)) \\ \sum_{e \in B \cup P \cup D} f_{new}(e) &= \sum_{e \in B} f_{new}(e) + \sum_{e \in P} f_{new}(e) + \sum_{e \in D} f_{new}(e) \\ &= \sum_{e \in B} (f_{old}(e) + \bar{f}(e)) + \sum_{e \in D} (f_{old}(e) + \bar{f}(e)) + \sum_{e \in P} f_{old}(e) \end{aligned}$$

Because of a) the \bar{f} terms are compensated and because of b) the f_{old} terms are compensated.

□

Lemma 3.7

Let l_k be the index of the last layer (the one which contains t) in the k th layering step. Then $l_{k+1} > l_k$.

Proof. When we reached the $(k+1)$ -layering (where we assume that it's not the last layering), then there is a part in the layered network $s = v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} \dots \xrightarrow{e_{l_{k+1}}} v_{l_{k+1}} = t$, $v_i \in V_i^{k+1}$ (i th layer in the $(k+1)$ -layering).

case 1:

All nodes of this path appeared already in the k -th layering step. $v_i \in \bigcup_{j=1}^{l_k} V_j^{(k)}$, $i = 0, \dots, l_{k+1}$.

- Claim: if $v_a \in V_b^{(b)}$ then $a \geq b$
Proof by induction on a
- Ing stduction: $a = 0$, $a = v_0 \in V_0^{(k)} = b \geq b$
- Induction assumption: let $v_a \in V_b^{(k)}$ then $a \geq b$
- Induction step: $a \rightarrow a + 1$, so let $v_a \in V_b^{(b)}$

Let $v_{a+1} \in V_c^{(k)}$, show that $a + 1 \geq c$.

- i) $c \leq b + 1$
 $v_{a+1} \in V_c^{(k)}$, by induction assumption $a \geq b$ then $a + 1 \geq b + 1 = c$

ii) $c > b + 1$

$$V_b^{(k)} V_{b+1}^{(k)} V_c^{(k)}$$

This would mean that the edge l_{a+1} , that is useful for $(k+1)$ layering was not used in the k -th layering, but it was already useful at that time, hence this case can not happen.

In particular we know:

$$t = v_{l_{k+1}} \in V_{l_k}^{(k)}, \text{ hence } l_{k+1} \geq l_k$$

The equality symbol in $l_{k+1} \geq l_k$ can not occur because otherwise the whole path $v_0 \rightarrow v_1 \dots v_{l_{k+1}}$ would have existed in the k -th layering. But when we determine the function \bar{f} we do it in a way that at least one edge e of the path is saturated. This edge e is no longer useful for the next layering.

case 2:

→ Not all v_i are contained in the case layering. Proof is analogously to case 1. □

Korollar 3.2

The repeated part of Dimic's algorithm

- i) Initialize f , e.g. $f \equiv 0$
- ii) Determine a layering
- iii) Determine \bar{f}
- iv) Calculate $f = f_{old} + \bar{f}$
- v) Go to step 1

is performed at most $|V| - 1$ times.

Proof. As $l_k < l_{k+1}$ and each layer contains at least one node. □

4

Matching

┌

Matchings in undirected graphs without parallel edges and loops

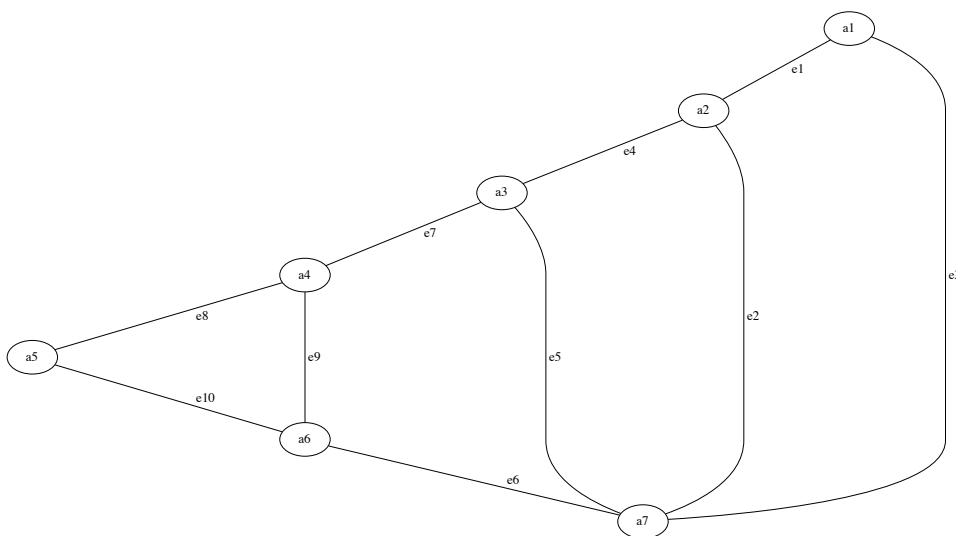
└

4.1 Elementary Definitions

Definition 4.1

A matching in an undirected (finite) graph $G = (V, E)$ is a set $M \subseteq E$ such that no two edges share a node. A matching M is called *maximal* if, for any matching M' with $M \subseteq M'$, $M = M'$ holds. M is of maximum cardinality if $|M| \geq |M'|$ for any matching M' . M is *perfect* if every node is incident on one edge.

Example



Remark: Perfect matching not possible with $|V|$ being odd. Maximum cardinality = 3. Empty set is a matching. All subsets of matchings are matchings.

Maximum cardinality: $\{e_3, e_1, e_9\}$ $\{e_{10}, e_5, e_1\}$

Maximal matching: $\{e_2, e_9\}$

Lemma 4.1

If M is a matching of maximum cardinality it is a maximal matching.

Proof. Let M' be a matching with $M \subseteq M'$, assume $M \subset M'$ then $|M'| > |M|$ (as M has maximum cardinality). \square

Remark: There are maximum matchings without maximum cardinality.

Lemma 4.2

If M is perfect then $2|M| = |V|$.

Proof. Obvious. \square

Lemma 4.3

If a graph has a perfect matching M then every matching of maximum cardinality is perfect. Clearly, M is of maximum cardinality.

Proof. Obvious. \square

4.2 Matching in bipartite graphs

Problem: Find a matching of maximum cardinality in a bipartite graph.

Solution: Construct an associated network as follows:

Let $G = (V, E)$ in a bipartite graph.

$$V = X \cup Y$$

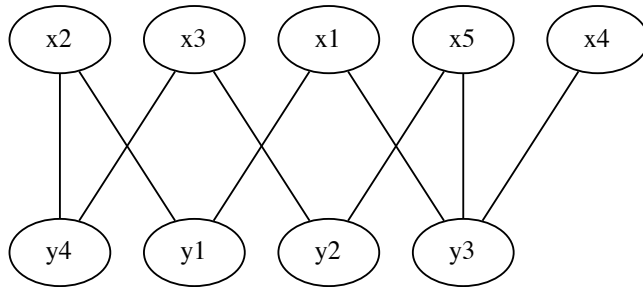
$$\bar{V} = V \cup \{s, t\}$$

$$X \cap Y = \emptyset$$

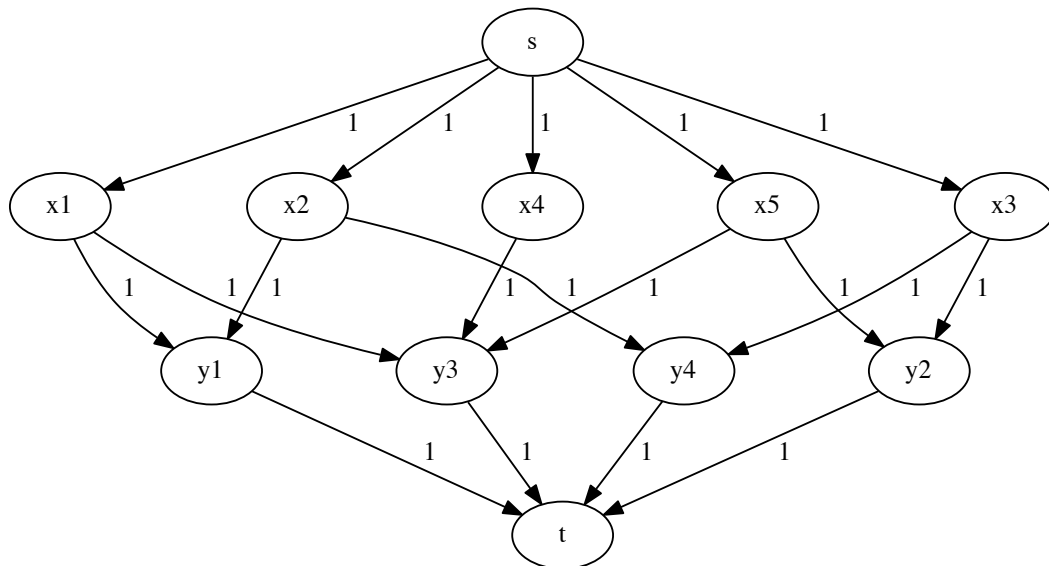
$$\bar{E} = \{s \rightarrow x : x \in X\} \cup \{y \rightarrow t : y \in Y\}$$

$$\bar{c}(\bar{e}) = 1, \forall \bar{e} \in \bar{E}$$

Example

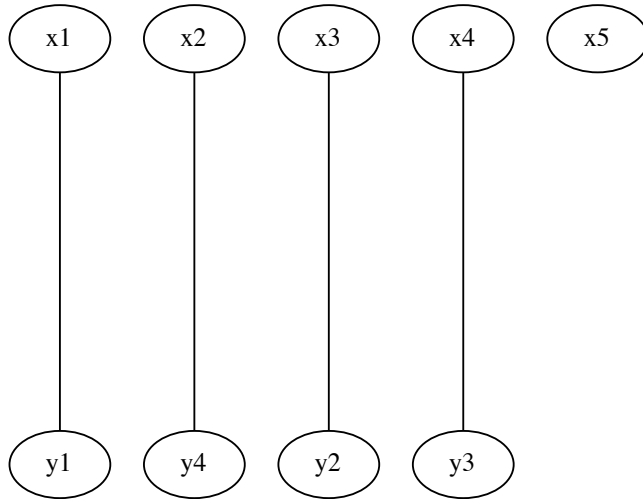


One solution:



- i) s marked
- ii) $x_1 : \overrightarrow{(s, x_1)}$
- iii) $x_2 : \overrightarrow{(s, x_2)}$
- iv) $x_3 : \overrightarrow{(s, x_3)}$
- v) $x_4 : \overrightarrow{(s, x_4)}$
- vi) $x_5 : \overrightarrow{(s, x_5)}$

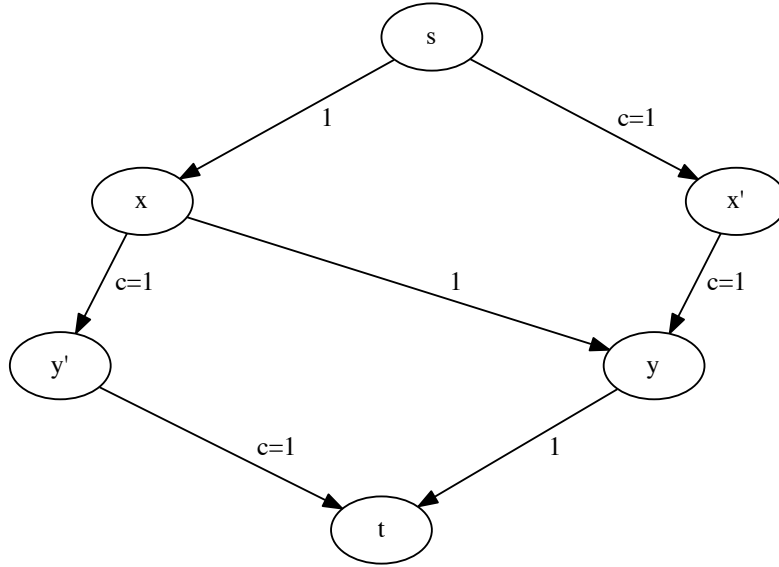
Solution:



Theorem 4.1

The number of edges in a matching M of maximum cardinality coincides with F_{max} , i.e. the maximum total flow in the associated network.

Proof. Let M be a matching of maximum cardinality. For every edge $x - y$ in M we transport one unit along the path $s - x - y - t$. We can do that as the edges in M are node disjoint, i.e. they do not share nodes. This defines a flow function f' with $F' = |M|$ and hence $F_{max} \geq F' = |M|$. Now let f be an arbitrary flow function for the associated network, without loss of generality we may choose f such that $f(e) \in \mathbb{N} \forall e$. All paths that connect s with t have the form $s \rightarrow x \rightarrow y \rightarrow t$. If such a path is used to transport a unit value then it is clear that no flow can happen along edges of the form $x \rightarrow y'$ or $x' \rightarrow y$.



Let $N = \{x - y : f(x \rightarrow y) = 1\}$. N is a matching and the total flow of f , i.e. F , satisfies $F = |N| \leq |M|$, hence also F_{max} , because M was assumed to be of maximum cardinality. Hence $|M| = F_{max}$ because we chose f arbitrarily. \square

This theorem yields an algorithm to determine a matching of maximum cardinality. Given a bipartite graph $G = (V, E)$ with $V = X \cup Y$ and $X \cap Y = \emptyset$:

- i) Construct the associated network.
- ii) On this network, calculate a flow function with maximum total flow.
- iii) The matching of maximum cardinality is obtained by $N = \{x - y : f(x \rightarrow y) = 1\}$.

Definition 4.2

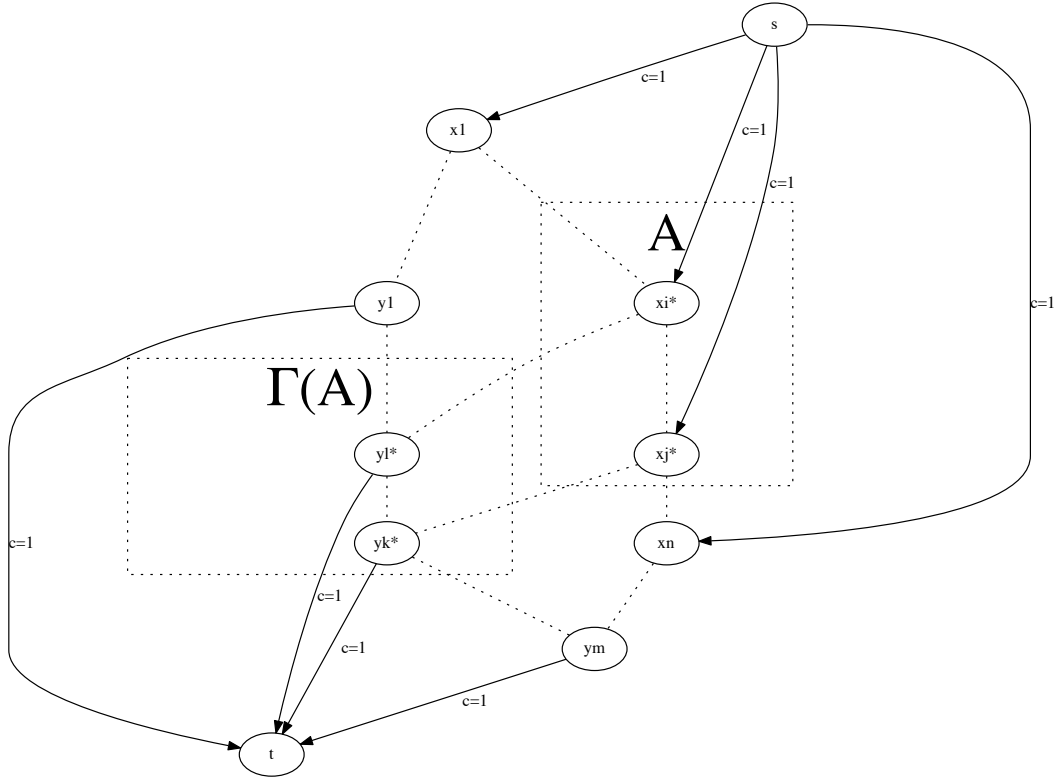
Let G be a bipartite graph, $V = X \cup Y$; $X \cap Y = \emptyset$. Let $A \subseteq X$. Let $\Gamma(A) = \{y \in Y : \exists x \in A : x - y \in E\}$. ($\Gamma(A)$: potential partners for the elements in A). A matching M is called *complete* if $|M| = |X|$, i.e. every $x \in X$ gets a partner.

Theorem 4.2

Wedding or marriage theorem: A bipartite graph has a complete matching if and only if for every $A \subseteq X$ $|\Gamma(A)| \geq |A|$.

Proof. “ \geq ” Let G have a complete matching M , let $A \subseteq X$. Then every $x \in X$ has a partner for itself in M , hence for every $A \subseteq X$ $|\Gamma(A)| \geq |A|$.

“<” Let $|\Gamma(A)| \geq |A|$ for every $A \subseteq X$. Assume there is no complete matching. Let S be the set of nodes that were marked in the last step of Ford-Fulkerson in the associated network. The calculated flow function determines a matching M with i) $F = |M|$. As we assumed that there is no complete matching we get ii) $|M| < |X|$.



Marking in the last round of Ford-Fulkerson. S = the marked nodes.

$A := X \cap S$

Let $y \in \Gamma(A)$ then there is an $x \in A = X \cap S$ marked and $x - y$ is an edge in E .

In the last successful construction of an augmenting path there was no flow determined along the edge $s \rightarrow x$, otherwise we could not mark x now. Hence the flow along the edge $x \rightarrow y$ must have been 0 before as well. Hence the remaining capacity $x \rightarrow y = 1$.

4.3 Stable matching (marriage)

Given:

1. n men $b_1 \dots b_n$, n women $a_1 \dots a_n$
2. each person has a preference list of the other sex:
 $a_1 : b_2, b_3, b_1 \dots$, where a_1 likes b_2 best.

□

Task: Form couples respecting the preferences.

A matching or coupling associates with each man exactly one woman. A matching is called

unstable if there is a man a and a woman A such that: a prefers A to the woman B he is matched with and b prefers B to the woman A he is matched with. A matching is called *stable* if it is not unstable.

Given n men, n women. Here are two approaches:

1. Attempt: Try all possible matchings

There are $n!$ possible matchings. There are therefore $\binom{n}{k} = O(n^2)$ checks to find out if one given matching is stable altogether $O(n! * n^2)$. For $n=10$ this equals 300,000,000 checks.

2. Gale-Shapley Algorithm

i) Mark every person as “free”.

ii) As long as there is a free man m do: Let F be the first woman to which m has not yet made an offer. If F is free then match m with F ((F,m)) and mark both as “bound”. If F is bound to m' then check if F prefers m to m' . If this is so, substitute the pair (F, m') by (F,m) . Mark m' as free and m as “bound”.

If F does not prefer m to m' then nothing is changed.

iii) Output all pairs.

Statement: For any input the algorithm terminates and yields a stable matching.

1. *Observation:* It cannot happen that a man is rejected by all women. This is because if a woman rejects or leaves a man she is bound. When the man reaches the end of his list then all women would be bound. ~~Because~~ there are n women and $n - 1$ other men.

2. *Observation:* The algorithm consists of offers made by men. Each man makes at most n offers $\Rightarrow O(n^2)$ offers in the worst case. \hat{A}

3. *Observation:* The result is a matching because the algorithm terminates when all men are bound and every man is bound to exactly one woman.

4. *Oberservation:* The resulting matching is stable. Assume this is not the case.

$m: F' \dots F$

$F': m \dots m'$

F matched with m .

F' matched with m' .

m makes offers according to his list. So he made an offer to F' before he reached F .

Case 1: F' was free. So she accepted his offer, but later left him. This can only happen because of men she likes better than m , but m' could not break into such a marriage.

Case 2: F' was bound.

Case 2.1: F' was bound to a partner she prefers to m , then she does not leave this partner because of m and def. not because of m' .

Case 2.2: F' leaves her partner because of m , but m' cannot break into this marriage.

Hence the matching is stable. The cost of the algorithm is $O(n^2)$.

$n = 10$: algorithm takes 100 Steps; trying all takes 300,000,000 steps.

$n = 1009$: algorithm takes 10^6 steps; trying all takes 500^{500} steps. Age of planet earth: $4 * 10^9$ years.

A man and a woman F are called *stable partners* if there is a stable matching in which they

are bound to each other. \hat{A}

Statement: All executions of the algorithm yield the same result. In this matching each man gets the best partner he can get in any stable matching. Let E be an arbitrary execution of the algorithm, let the result be the matching M_E . Assume there is another matching M .

$M_E: m - F$

$M: m - F'$

$m: \dots F' \dots F \dots$

Then deriving E , F' must have rejected / left m . Without loss of generality, let this be the first time that a woman rejected / left a stable partner. Let m' be the reason for this.

$F': \dots m' \dots m \dots$

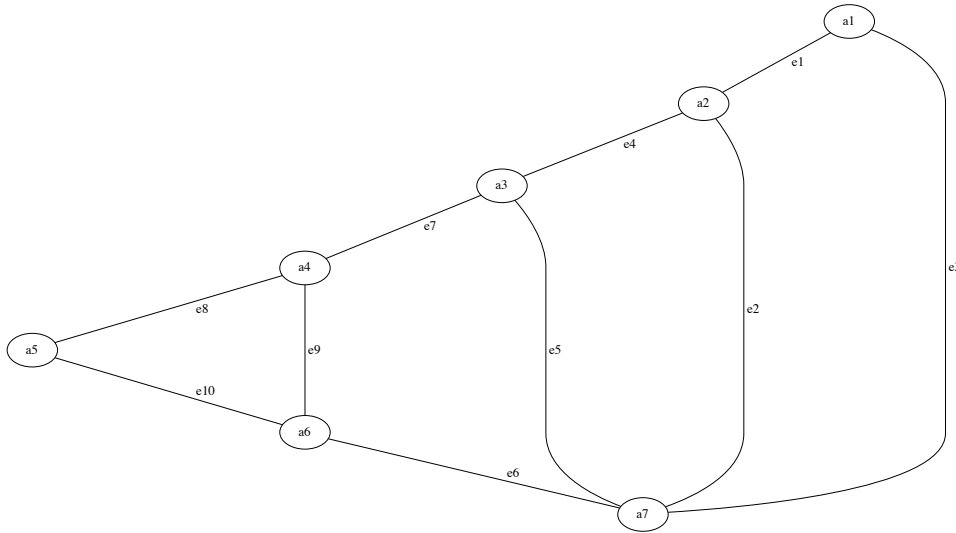
Then m' cannot have a stable partner he prefers to F' , hence m' prefers F' to his partner in M .

$M: m - F'; m' - F''$

So M is not stable.

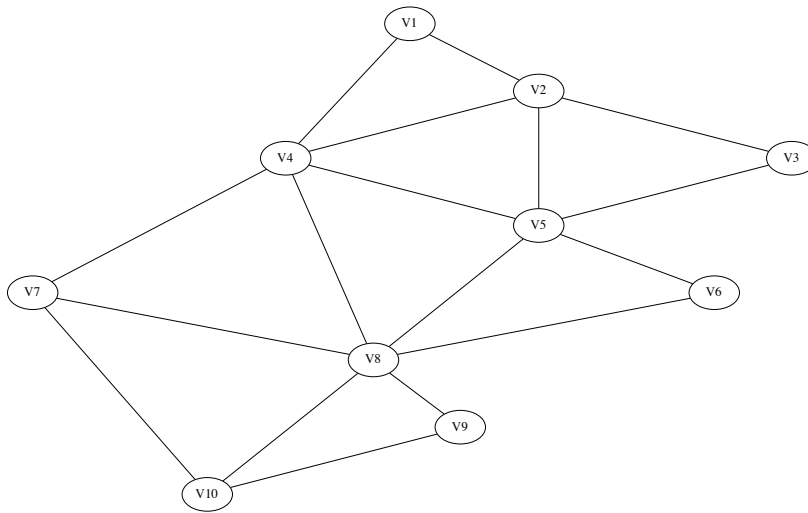
4.4 Maching in general graphs

Example



$\{e_1, e_5, e_9\}, \{e_1, e_5, e_{10}\}$ are both matchings of maximal cardinality because $|V| = 7$. Every subset of a matching is a matching.

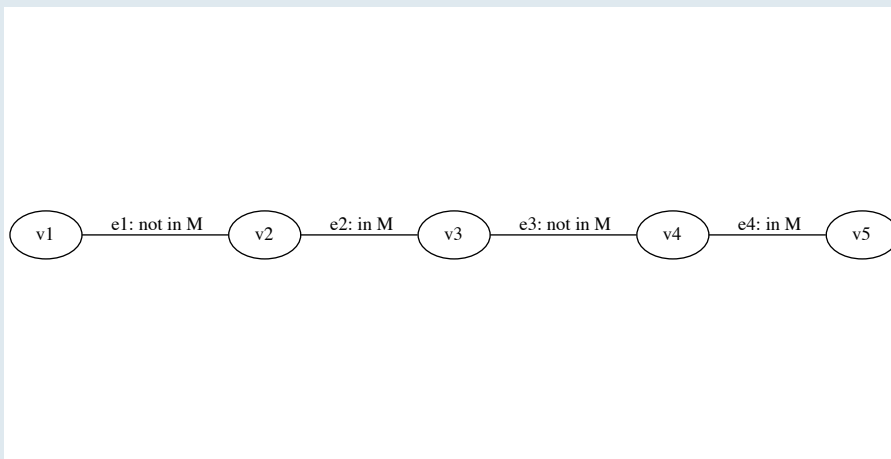
Example



$$M_0 = \{\{V_2, V_3\}, \{V_4, V_5\}, \{V_7, V_{10}\}\}$$

Definition 4.3

Let $G = (V, E)$ be a graph and M a matching for G . The edges in M are called “matched”, the others are called “free”. If $\{u, v\} \in M$ then u, v are called partners and “matched”. If $v \in V$ lies on no edge of M then v is called “free”. A simple path e_1, e_2, \dots, e_k is called M -alternating if $e_1, e_3, e_5 \dots$ are free edges and e_2, e_4, \dots are matched.



An M -alternating path is called M -augmenting if the start and end points of the path are free.

Lemma 4.4

If p is an M -augmenting path, $p = e_1, \dots, e_l$, then l is odd and the number of nodes in the path is even.

Proof. As the first and last node on an M -augmenting path must be free, the first and last edge must be free edges.

$\frac{e_1}{\text{free}} \frac{e_2}{\text{free}} (e_1 e_2)(e_3 e_4) \dots \hat{A}$

Assume l is even, then we can group the edges into pairs $(e_1 e_2)(e_3 e_4) \dots$ and so on and the last edge would then be matched.

Contradiction: In a simple path of odd length the number of nodes is even.

□

Lemma 4.5

Let $G = (V, E)$ be a graph, M a matching and p an M -augmenting path. Then $M' = M \setminus p \cup p \setminus m$ is a matching and $|M'| = |M| + 1$. (See ex. E and M_0).

- $p = v_6 - v_8$
 M_0 -alternating? Yes
 M_0 -alternating? Yes
- $p' = v_1 - v_2$
 M_0 -alternating? Yes
 M_0 -alternating? No
- $M_1 = M_0 \setminus p \cup p \setminus M_0 = \{\{v_2, v_3\}, \{v_4, v_5\}, \{v_7, v_{10}\}, \{v_6, v_8\}\}$
- $p_1 : v_1 - v_2 - v_3 - v_5 - v_4 - v_8$
 M_1 -alternating? Yes
 M_1 -alternating? No
- $p_2 : v_1 - v_4 - v_5 - v_6 - v_8 - v_7 - v_{10} - v_9$
 M_1 -alternating? Yes
 M_1 -alternating? Yes
- $M_1 = M_1 \setminus p_2 \cup p_2 \setminus M_1 = \{\{v_2, v_3\}, \{v_1, v_4\}, \{v_5, v_6\}, \{v_8, v_7\}, v_{10}, v_9\}$
 $|M_2| = 5$ is of maximum cardinality as there are 10 nodes.

Proof. Let p be an M -augmenting path.

$p: v_1 - v_2 - v_3 \dots - v_{2k}$

Show 1. M' is a matching ($M' = M \setminus p \cup p \setminus M$), i.e. no two edges share a node. Assume there are two edges in M' that share a node, e and e' . There are 3 cases to consider:

i) $e, e' \in M \setminus p$

ii) $e, e' \in p \setminus M$

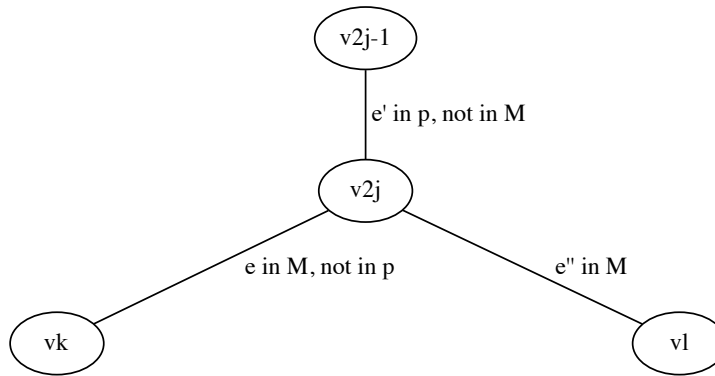
iii) $e \in M \setminus p, e' \in p \setminus M$

Case i) e, e' share a node and are in $M \setminus p$, in particular $e, e' \in M \setminus p$

Case ii) $e, e' \in p \setminus M$, in particular $e, e' \in p$ and not in M . But on p only edges share a node if one is in M and the other isn't.

Case iii) Let $e \in M \setminus p, e' \in p \setminus M$, i.e. e' is free.

Example



Without loss of generality, assume that v_{2j} also lies on e .

e'' is in M , hence $e, e'' \in M$ and share a node. (If v_{2j} is the last node, use v_{2j-1} for a similar argument.)

Show 2. $|M'| = |M| + 1$

p : $v_1 - v_2 \dots v_{2k-1} - v_{2k}$, where $\{v_1, v_2\}$ and $\{v_{2k-1}, v_{2k}\}$ are not in M .

p contains $2k - 1$ edges and $k - 1$ belong to M and k do not belong to M .

$e_1, e_3, \dots, e_{2k-1}$

$$|(M \setminus p) \cup (p \setminus M)| = |M \setminus p| + |p \setminus M|$$

$$= |M| - (k - 1) + k$$

$$= |M| + 1$$

//

Lemma 4.6

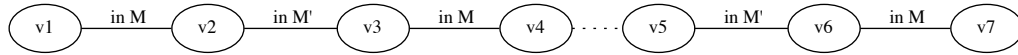
$G = (V, E)$ graph, M matching, $u \in V$ free. Let us assume that there is no M -augmenting path starting in u . Let p be an M -augmenting path with end points v, w . Then there will be no $(M \setminus p \cup p)$ -augmenting path starting from u .

Lemma 4.7

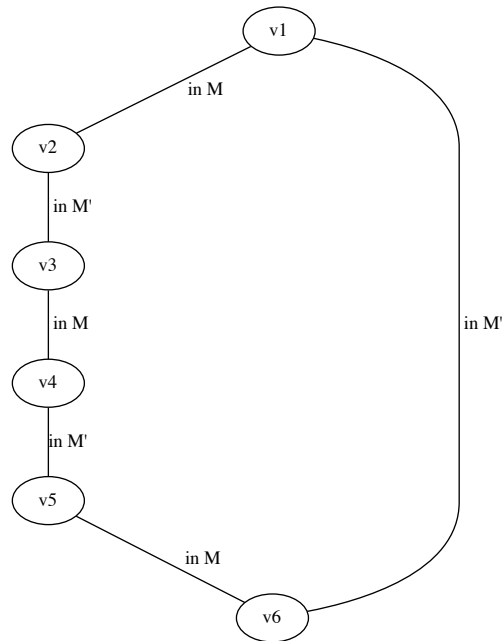
A matching M is of max. cardinality if and only if there is no M -augmenting path.

Proof. “ $>$ ” Let M be of max. cardinality. Assume there is an augmenting path, then construct $M' = M \setminus p \cup p$ and $|M'| > |M|$. \hat{A}

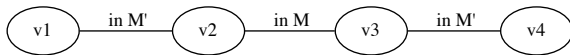
“ $<$ ” Let there be no M -augmenting path. Assume that there is a matching M' with $|M'| > |M|$. Consider the edges in $M \setminus M' \cup M' \setminus M$. $G' = (V, (M \setminus M') \cup (M' \setminus M))$. In this graph every node has degree 2 or less. If v has degree 2, then one edge belongs to M , the other to M' . Two connected components of G' are either paths or circles.



or



where the number of edges in the circle belonging to M equals the number of edges belonging to M' . As $|M'| > |M|$ and the circle cannot contribute to this fact, we conclude that there must be a path with more edges in M' than in M . This path must look like this:



⚡

□

Here is an idea for an algorithm which finds a matching of maximal cardinality:

- i) Start with an initial matching, e.g. consisting of one edge

- ii) Find an M-augmenting path. Put $M = M \setminus p \cup p \setminus M$. Go to 2. If there is no M-augmenting path, stop; the current matching is of maximal cardinality.

5

Networks with costs, respectively upper/lower bounds

5.1 Networks with costs

Get rid of parallel and antiparallel edges, cost: $E \rightarrow \mathbb{R}$

Example 5.1

case a)

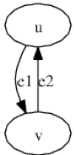


Figure 5.1: With antiparallel edges

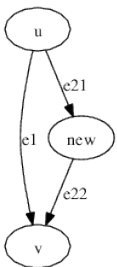


Figure 5.2: With antiparallel edges

5.1 is transformed to 5.2

$c(e_1)$ as before c .

$cost(e_{21}) = cost(e_2)$

$cost(e_{22}) = 0$

$$\text{cost}(e_21) = \text{cost}(e_22) = \text{cost}(e_2)$$

case b)

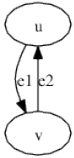


Figure 5.3: With antiparallel edges

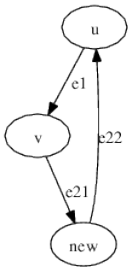


Figure 5.4: After transformation - without antiparallel edges

5.3 is transformed to 5.4

From now on we consider only networks without parallel/antiparallel edges.

Definition 5.1

Let $G = (V, E)$ with $s, t \in V, c : E \rightarrow \mathbb{R}^+$ be a network. Let $\text{cost} : E \rightarrow \mathbb{R}$ be a function that associates “cost” to every edge. Let f be a flow function for this network.

$$\text{cost}(f) = \sum_{e \in E} f(e) * \text{cost}(e)$$

Task:

Given:

- i) a network $G = (V, E), s, t \in V, c, \text{cost}$
- ii) $w \in \mathbb{R}^+$

Find a flow function f with total flow $F = w$ and minimal costs.

Definition 5.2

$G = (V, E), s, t \in V, c, cost, f$ flow function

The graph $G_f = (V, E_f)$ with \tilde{c}, \tilde{cost} s.

Let $e = (u, v) \in E$

i) If $f(e) < c(e)$ then $e_1 = (u, v) \in E_f$
 $\tilde{c}(e_1) = c(e) - f(e) > 0$
 $\tilde{cost}(e_1) = cost(e)$

ii) If $0 < f(e)$ then $e_2 = (v, u) \in E_f$
 $\tilde{c}(e_2) = f(e)$
 $\tilde{cost}(e_2) = -cost(e)$

Observation:

a If $e \in E$ is an edge with $0 < f(e) < c(e)$

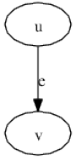


Figure 5.5: $G = (V, E)$

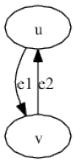


Figure 5.6: G_f

b If $e \in E$ is an edge with $0 < f(e) = c(e)$ - as shown in 5.7.

c If $e \in E$ is an edge with $0 = f(e) < c(e)$ - as shown in 5.8.

d If $e \in E$ is an edge with $0 = f(0) = c(0)$ - as shown in 5.9.

Definition 5.3

Let p be a directed cycle in G_f . $cost(p) := \sum_{e \in p} cost(e)$

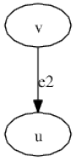


Figure 5.7: G_f

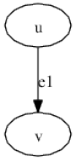


Figure 5.8: G_f

Example 5.2

directed cycles of negative costs: $t - h - b - d - t \rightarrow -1$
 $s - c - g - h - b - a - s \rightarrow -5$

Theorem 5.1

Let a network with cost function be given and a flow function f with total flow $F = w$. f has least costs among all flow functions with total flow w if and only if G_f does not have any directed cycles of negative cost.

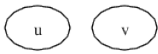
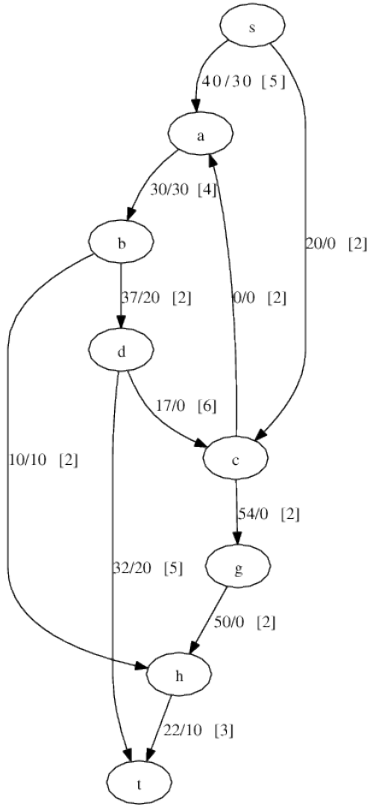
Proof. “ \implies ” Let f be a cost minimal flow function with total flow $F = w$. Assume that G_f contains a directed cycle of negative costs. Adapting the flow values along this cycle in the original network we can reduce the cost while maintaining the total flow. \square

Example 5.3

For edges that correspond to forward edges (of type e_1) we may raise the flow value. For edges that correspond to backward edges (of type e_1) we may reduce the flow.

Lemma 5.1

Let f be flow function with total flow $F = w$ and let f be cost minimal. Let p be an augmenting path that is cost minimal then the resulting flow f' constructed from f and p is cost minimal for $w + \Delta$ (Δ from the augmenting path).


 Figure 5.9: G_f

 Figure 5.10: G

5.2 Networks with upper and lower bounds

we associate with each edge e a lower bound $b(e)$ and request for the flow function additionally that:

$$b(e) \leq f(e)$$

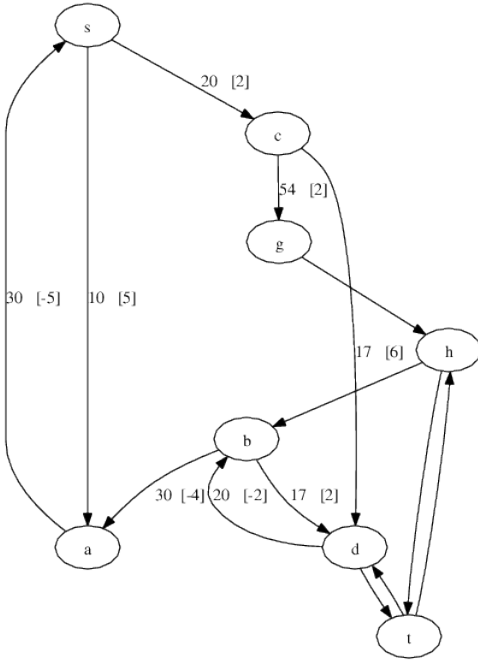
A flow function obeying this additional constraint is called legal flow function.

Example 5.4

Graphic of no legal flow missing

To answer the question if there is a legal flow for a network with lower/upper bounds we proceed as follows. Construct the Graph $\tilde{G} = (\tilde{V}, \tilde{E})$.

i) $\tilde{V} = V \cup \{\bar{s}, \bar{t}\}$


 Figure 5.11: G_f

ii) for every node $v \in V$ introduce an edge e from v to \bar{t} .

$$\bar{c}(e) = \sum_{e \in \beta(v)} b(e)$$

$$\bar{b}(e) = 0$$

iii) for every node $v \in V$ introduce an edge e from \bar{s} to v .

$$\bar{c}(e) = \sum_{e \in \alpha(v)} b(e)$$

$$\bar{b}(e) = 0$$

iv) The edges in E remain but with new bounds:

$$\bar{c}(e) = c(e) - b(e)$$

$$\bar{b}(e) = 0$$

v) Introduce edges from $s \xrightarrow{e} t$, $t \xrightarrow{e'} s$ with:

$$\bar{c}(e) = \bar{c}(e') = \infty$$

$$\bar{b}(e) = \bar{b}(e') = 0$$

BarE consists of the edges in E plus the newly introduced edges.

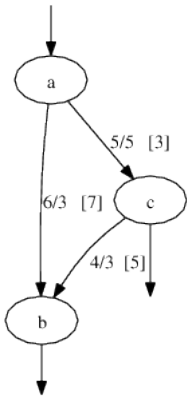


Figure 5.12: G_f

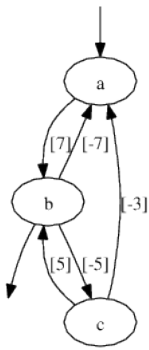


Figure 5.13: G_f

Lemma 5.2

The original network with upper and lower bounds has a legal flow if and only if the maximum flow of the auxiliary network saturates all edges emanating from \bar{s} .

Hint:

Let Barf be a flow function for the associated network with maximal total flow then $f(e) = \text{Barf}(e) + b(e)$ is a legal flow function ($e \in E$).

6

NP-Completeness

┌

Definition and analysis of different complexity classes

└

6.0.1 Motivation/Examples

Solvable in Polynomial Time	NP complete
Euler Path "a path with all edges" Shortest path from X to Y \Rightarrow Dijkstra's algorithm	Hamiltonian Path "node occurs exactly once" longest simple Path between X and Y
2-CNF (conjunctive normal form) e.g. $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3)$ Can we find an assignment, that the formula becomes true in poly. time?	3-CNF
Two processor scheduling	3 or more processor scheduling \Rightarrow NP-complete or unknown
Football game until 1955 loss 0 points, draw 1 point, win 3 points. The season has already started, is my team still able to win the championship?	loss 0 points, draw 1 point, win 3 points

6.0.2 Introduction

Definition 6.1

P is the class of problems that can be solved in polynomial time.

NP is the class of problems, where given a solution one may check in polynomial time, that it is indeed one.

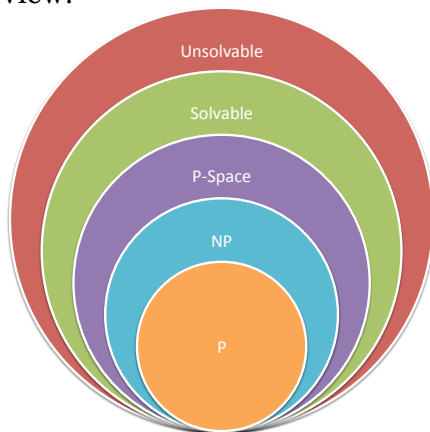
Example 6.1

Example:

- i) Given a connected undirected-finite graph. Is there a circle such that every node appears exactly once on the circle. Given such a circle $\langle v_1, \dots, v_n \rangle$ it can easily be checked that it has the property. \Rightarrow Hamiltonian Cycle \in NP.
- ii) 3-CNF; given values for the variables, we can check that the formula becomes true, with these values in polynomial time.

Clarity: $P \subseteq NP$ open problem; $P = NP$??

Overview:



6.0.3 Problem Types/Issues

Decision Problems	Optimization Problems
Yes or no answer, is there a path from X to Y?	Find the shortest path from X to Y?

Definition 6.2

Complexity Theory deals with **decision problems**. Optimization problems are tried to transform into decision problems.

Example 6.2

Choose an integer k , treat the decision problem: "Is there a path from X to Y no longer than k ?"

To show that an optimization problem is hard to solve it is sufficient to consider the related decision problem. If the latter is hard to solve, so is the first.

Reductions

Reductions are used to show that problem are hard to solve. A reduction reduces a decision problem to another one.

$A \Rightarrow B$ (reduction)

Properties of the reduction should be: "it can be done in polynomial time".

If A and B the transformation of AB is an instance of B then A should have the answer yes if and only if β has the answer yes.

Call this a **polynomial reduction**.

If $A \rightarrow B$ and B can be solved in poly. time, then instances of A can be solved in poly. time.

If $A \rightarrow B$ and A is hard to solve, then we may conclude that B is hard to solve.

Lecture of 08.05.13 - beginning: redundancy to former parts?

Consider the following classes: P, NP, NPC

Class $A \xrightarrow{\text{poly}} B$

Instance $\alpha \rightarrow \beta$

1 Observation: If the instances of the problem B can be solved in poly time, then this is true for instances of A .

2 Observation: If the instances of A are "hard" to solve, so are the instances of B .

3 Observation: If A is in NPC and $B \in NP$, then we can conclude B is also in NPC.

Problem: To find a first problem class in NPC (St. Cook 1971 3-SAT problem)

6.1 Polynomial Time

1. Problem solvable in $O(n^{100})$?

Speed-up theorem of Blum \rightarrow reduce exponent

2. Polynomial time is independent of the computer model

3. The class of polynomials is closed with respect to addition, multiplication and composition.

4. Practical reasons.

Definition 6.3 Abstract Decision Problem

An **Abstract Decision Problem** is a mapping $Q : I \rightarrow \{0, 1\}$. I : is a set of problem instances

Example 6.3

Is there a path from u to v in the directed graph G of length $\leq k$ $I = \{(G, u, v, k) : k \in \mathbb{N}, G \text{ finite directed Graph, } u, v \text{ are nodes in } G\}$

$Q((G, u, v, k)) = 1$ if \exists path ... and $Q((G, u, v, k)) = 0$ otherwise.

6.1.1 Coding and concrete problems

A coding of a set S is a mapping $e : S \rightarrow \{0, 1\}^*$.

A concrete problem is a problem, the instances of which are in $\{0, 1\}^*$.

$Q : S' \subseteq \{0, 1\}^* \rightarrow \{0, 1\}$

An algorithm solves a concrete problem in $O(T(n))$ if it delivers the answer for an input x with $|x| = n$ on $O(T(n))$ steps.

A concrete problem is said to be solvable in polynomial time if there is an algorithm solving it in $O(n^k)$ for some k .

P is the class of concrete problems that can be solved in poly time.

Let $Q : I \rightarrow \{0, 1\}$ be an abstract problem and $e : I \rightarrow \{0, 1\}^*$ be an encoding.

The resulting concrete problem $Qe : e(I) \rightarrow \{0, 1\}$ such that $Qe(i) = 1$ iff $Q(e(i)) = 1 \subseteq \{0, 1\}^*$.

Example 6.4

Consider an algorithm for an abstract problem on \mathbb{N} that needs $\Theta(k)$ steps for input $k \in \mathbb{N}$.

encoding e : $\mathbb{N} \rightarrow \{0, 1\}^*$ $e(k) = 11111$ (k times), then the algorithm is linear in the length of the input.

encoding e' : $e'(k) = \text{binary representation of } k$ $\text{length } e'(k) = |e'(k)| = \log_2 k + 1$ (log hat eigentlich so eckige klammern)

The algorithm with this encoding produces costs exponential in the length $|e'(k)|$ of the input $f : \{0, 1\}^* \rightarrow \{0, 1\}$ is said to be compatible in poly time if there is an algorithm that computes $f(x), x \in \{0, 1\}^*$ in poly time.

Two encodings are called polynomially connected if there are two functions f_{12} and f_{21} with $f_{12}(e_1(i)) = e_2(i), f_{21}(e_2(i)) = e_1(i), i \in I$ that are polynomially computable.

Lemma 6.1

Let Q be an abstract decision problem.

$Q : I \rightarrow \{0, 1\}$

$e_1, e_2 : I \rightarrow \{0, 1\}^*$ encodings that are polynomially connected, then $Q_{e_1} \in P$ iff $Q_{e_2} \in P$

Proof. Steps:

i) $Q_{e_1} : e_1(I) \subseteq \{0, 1\}^* \rightarrow \{0, 1\}$ be solvable in $O(n^k)$ for some k for input of length n

ii) Let $e_1(i)$ be computable from $e_2(i)$ in $O(n^c)$ where $n = |e_2(i)|$.

We want to show, that $Q_{e_2} \in P$ to solve Q_{e_2} we proceed as follows:

$Q_{e_2} : e_2(I) \rightarrow \{0, 1\}$ consider $e_2(i)$. First determine $e_1(i)$ from $e_2(i)$ and then run the algorithm for Q_{e_1} on the input $e_1(i)$.

The first step costs $O(n^c)$, $n = |e_2(i)|$ and the result is $e_1(i)$.

In the second step the algorithm for Q_{e_1} is run on $e_1(i)$, $|e_1(i)| = O(|e_2(i)|^c)$.

It needs $O((n^c)^k) = O(n^{ck})$

Sidenote: $\text{length} \rightarrow |e_1(i)| \leq O(n^2) \rightarrow P \leq PSPACE$

□

6.2 Formal Language Representation

Let ϵ be an alphabet (=finite set, the elements are called symbols).

A language over ϵ is a subset of ϵ^* , ϵ^* all finite sequences of over ϵ .

$L \subseteq \epsilon^*$ (Grammar density?)

Start lecture on 13.05.2013

Decidability vs. Acceptance

ϵ finite set of symbols, ϵ^* all finite sequences.

Language is $L \subseteq \epsilon^*$.

Let Q be a concrete decision problem $L = \{x \in \{0, 1\}^* \mid Q(x) = 1\}$. An algorithm A accepts its output

Sentence Missing!!

A rejects x if $A(x) = 0$.

A decides a language $L = \{x \in \{0, 1\}^*\}$ if $A(x) = 1 \forall x \in L$ and $A(x) = 0 \forall x \notin L$. L is accepted by A in polynomial time, if A accepts L and there is a $k \in \mathbb{N}$, such that for every $x \in L$, if $|x| = n$ A accepts x in $O(n^k)$ with $n \in \mathbb{N}$.

Algorithm A decides L in polynomial time if there is $k \in \mathbb{N}$ such that if $|x| = n$ then A decides if $x \in L$ in $O(n^k)$.

Given a program P_r and input y , the question: "Will P terminate?" is **undecidable** **!important theorem!**

But there is an algorithm $\{(P_r, y) : P_r \text{ terminates on } y\}$ that can be accepted.

Now we can defer the class $P = \{L \subseteq \{0, 1\}^* : \exists \text{ algorithm deciding } L \text{ in polynomial time}\}$.

Lemma 6.2

$P = \{L \subseteq \{0, 1\}^* : \exists \text{ algorithm that accepts } L \text{ in polynomial time}\}$.

Proof. Each problem that can be decided (in polynomial time) can be accepted (in polynomial time). Let L be accepted in polynomial time by an algorithm A , that there is a $k \in \mathbb{N}$ such that for each $|x| = n$ A accepts $x \in L$ in $O(n^k)$.

So an algorithm deciding L , works as follows:

Let $x \in \{0, 1\}^*$ with $|x| = n$ be given. Run algorithm A $O(n^k)$ steps. If it delivers 1 then deliver 1, otherwise 0, because if x was in L , the answer 1 must have been produced with $O(n^k)$ steps. \square

6.3 Polynomial Verification

A verification algorithm is an algorithm with two input parameters $A(x, y)$.

A verifies x if there is a y with $A(x, y) = 1$.

y is called a certificate.

$L = \{x \in \{0, 1\}^* : \exists y \in \{0, 1\}^* : A(x, y) = 1\}$ the language verified by A .

A language belongs to the class NP if there is a polynomial time verification algorithm A and a constant c such that:

$$L = \underbrace{\{x \in \{0, 1\}^* : \text{a graph}\}}_{\text{Hamiltonian Cycle}} : \underbrace{\exists y}_{\text{Hamiltonian Cycle}} \text{ with } |y| = O(|x|^c) \text{ and } \underbrace{A(x, y) = 1}_{\text{algo that shows that } y \text{ is a Hamiltonian Cycle}} \}$$

We say A verifies L in polynomial time.

Some complexity classes and their relationships:

$$\text{co-NP} = \{L : \epsilon^* \setminus L \in \text{NP}\}$$

$$\text{co-AnyClass} = \{L : \epsilon^* \setminus L \in \text{AnyClass}\} \text{ So far proven or not:}$$

$$P \subseteq \text{NP}$$

$$P \stackrel{?}{=} \text{NP}$$

$$P \stackrel{?}{=} \text{co-NP} \cap \text{NP}$$

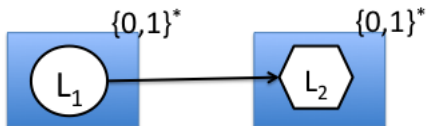
$$\text{co-NP} \stackrel{?}{=} \text{NP}$$

6.4 NP-completeness and reducibility

L_1 is reducible in polynomial time to L_2 ($L_1 \leq_p L_2$) if there is a function f computable in poly. time with:

$$x \in L \text{ iff } f(x) \in L_2 \quad f : \{0,1\}^* \rightarrow \{0,1\}^*$$

f is called a reduction function



Lemma 6.3

$L_1, L_2 \in \{0,1\}^*$ and $L_1 \leq_p L_2$.

If $L_2 \in P$ then $L_1 \in P$. $L \in \{0,1\}^*$ is called **NP-complete** (NPC) if

- i) $L \in NP$
- ii) $L' \leq_p L \forall L' \in NP$

If L satisfies ii) L is called **NP-hard**.

$NPC = \{L : L \text{ is NP-complete}\}$

Lemma 6.4

If some L in NPC can be solved in polynomial time, then $P = NP$.

If there is a $L \in NP$ that cannot be solved in poly. time then no problem in NPC can be solved in polynomial time.

Lemma 6.5

If $L' \in NPC$ and $L' \leq_p L$ the L is NP-hard.

Proof. $L' \in NPC$ and $\forall L'' \in NP$ it holds $L'' \leq_p L' \leq_p L$

">" $\forall L'' \in NP \quad L'' \leq_p L$

□

The first problem to show NP-complete (remember check for NP \rightarrow check in poly. time if you have a solution): **SAT**. Satisfiability of Boolean Formulas (Cook 1971)

The second problem to show NP-complete: 3-SAT. $SAT \leq_p 3\text{-CNF-SAT}$

Sidenote:

CNF, Conjunctive Normal Form $\rightarrow (\vee \vee) \wedge (\vee \vee) \wedge (\vee \vee)$

3-CNF in every bracket three arguments.

Example: $(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$

CLIQUE:

Let $G = (V, E)$ be an undirected graph. $V' \subseteq V$ is called a clique if any two nodes in V' are connected by an edge.

Size of a clique: number of nodes.

CLIQUE = $\{(G, k) : G \text{ has a clique of size } k\}$ Naive Solution:

Given G, k consider all subsets of size k in V , $|V| = n$. There are $\binom{n}{k}$ subsets. For each we have to perform $\binom{k}{2}$ checks for edges, all together $\binom{n}{k} * \binom{k}{2}$ steps $\rightarrow O(n^k)$

Lemma 6.6

CLIQUE is NP-complete

Proof. CLIQUE is in NP: Given a set $V', |V'| = k$: check $\binom{k}{2}$ times that there is an edge, $\binom{k}{2} = O(k^2)$

CLIQUE is NP-hard: We reduce 3-CNF SAT to CLIQUE. $SAT \leq_p 3\text{-CNF-SAT} \leq CLIQUE$

So take an arbitrary formula in 3 CNF $\Phi = C_1 \wedge \dots \wedge C_r$, $C_i = (l_1^i \vee l_2^i \vee l_3^i)$. We construct a graph G_Φ such that G_Φ has a k -CLIQUE iff G_Φ is satisfiable.

For each C_r we introduce 3 nodes $v_1 \dots v_2 \dots v_3$ corresponding to

ONE SENTENCE MISSING!

$V = \{v_j^r, r = 1 \dots k, j = 1, \dots, 3\}$ E: There is an edge between v_i^r and v_j^s if

- $r \neq s$
- l_i^r is not the negation of l_j^s

□

Example 6.5

$$\Phi = \underbrace{(x_1 \vee \neg x_2 \vee \neg x_3)}_{C_1} \wedge \underbrace{(\neg x_1 \vee x_2 \vee x_3)}_{C_2} \wedge \underbrace{(x_1 \vee x_2 \vee \neg x_3)}_{C_3}$$

Solution see handout

Proof. CLIQUE is NP-complete cont'd $\Phi = C_1 \wedge \dots \wedge C_k$ $C_i = (l_1^i \vee l_2^i \vee l_3^i) \Rightarrow$: Let α be satisfiable let α be an assignment to the variable that makes Φ true, then under α each C_i must be true. Hence every C_i at least one literal must be α . Choose from each C_i one literal that becomes true under α . The corresponding nodes in G from α set V' with $|V'| = k$ for any two nodes in $V' : r \neq s$ and for the corresponding literals we know that one is not the negation of the other because both obtained the truth value under α . Hence for any two nodes in V' there is an edge between. Hence V' is a clique.

$$\underbrace{C_1}_{(l_1^1 \vee l_2^1 \vee l_3^1)} \wedge \underbrace{C_i}_{(l_1^i \vee l_2^i \vee l_3^i)} \wedge C_k$$

\Leftarrow : Let $V' \subseteq V$ be a clique with $|V'| = k$. As for any two nodes v_i^r, v_j^s in V' there is an edge between them, we know that $r \neq s$ and l_i^r is not the negation of l_j^s . For every literal that

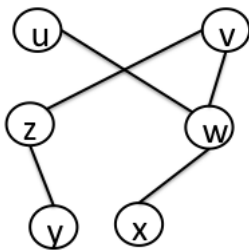
corresponds to a node in V' we put the truth-value true. So independent of the truth value of the remaining variables each C_i gets the truth value true and hence ϕ gets the truth-value true. \square

Reduction: VERTEX-COVER

Let $G = (V; E)$ be an undirected graph. A subset $V' \subseteq V$ is called a **VERTEX-COVER** if every edge $e \in E$ has at least one endpoint in V' . The size of a **VERTEX-COVER** V' is the number of nodes in V' .

VERTEX-COVER = $\{(G, k) : G \text{ has a Vertex-Cover of size } k\}$

Example 6.6

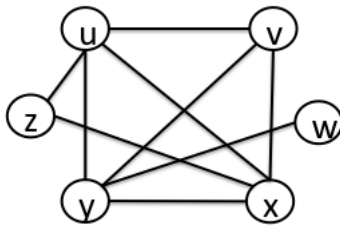


a minimal vertex cover: $\{z, w\}$

Statement: Vertex-Cover is NP-Complete

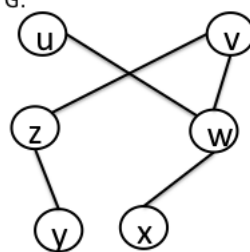
1. VC is in NP: Given a graph G and $k \in \mathbb{N}$ and a set V' we first check if V' has k elements and go through all edges in E and check if one end point is in V' . This can be done in polynomial time.
2. To show that VC is NP-hard: Reduce clique to vertex-cover let $G = (V, E)$ be an undirected graph. We construct a graph $\bar{G} = (V, \bar{E})$ as follows: $\bar{E} = \{\{u, v\} : u \neq v \text{ and } \{u, v\} \notin E\}$

G :



max Clique = $\{u, v, y, x\}$

\bar{G} :



min Vertex-Cover = $\{z, w\}$

Statement: G has a CLIQUE of size k if and only if \bar{G} has a vertex cover of size $|V| - k$

Proof. \Rightarrow : Let G be a graph with a clique V' . $|V'| = k$. $V \setminus V'$ is a Vertex Cover in \bar{G} . We have to show that every edge in \bar{E} has an endpoint in $V \setminus V'$. So let $\{u, v\} \in \bar{E}$, hence $\{u, v\} \notin E$. At least one of u, v is not in V' hence at least one of $u, v \in V \setminus V'$. So $V \setminus V'$ is a vertex cover. \Leftarrow :

$\bar{G} = (V, \bar{E})$ and let $V' \subseteq V$ be vertex-cover with $|V'| = |V| - k$ we show $V \setminus V'$ is a (clique of size k).
Let $(u, v) \in V \setminus V'$: Assume $\{u, v\} \notin \bar{E}$ then $\{u, v\} \in E$ hence as V' is a vertex cover, we know that u or v are in V' .
!!Contradiction: $u, v \in V \setminus V'$ □

7

Approximation Algorithms

An algorithm for an optimization problem that yields a “nearly” optimal solution is called an approximation algorithm.

Quality of Approximation

$0 < C_n^*$ cost of an optimal solution for an input of size n (e.g. length of a shortest path). $0 < C_n$ cost of a solution produced by an approximation algorithm. If $\max(\frac{C_n}{C_n^*}, \frac{C_n^*}{C_n}) = \rho(n)$ then the algorithm has an approximation ratio of $\rho(n)$.

(Remark: $\rho(n) \geq 1$)

$\rho(n) = 1$: optimal, $\rho(n) \gg 1$: bad approximation

Trade-off: Cost of the approximation algorithm vs. quality of approximation.

An approximation scheme for an approximation problem is an algorithm with 2 inputs:

Problem + $\epsilon > 0$ such that the algorithm is a $(1 + \epsilon)$ - approximation algorithm.

The approximation scheme is a polynomial time approximation scheme if for each $\epsilon > 0$ the induced algorithm runs in polynomial time in n . It is fully polynomial if it is polynomial in n and in $1/\epsilon$.

Example 7.1

$O(n^{(\frac{2}{3})})$, if $\epsilon' = \frac{3}{4} O(n^{(\frac{8}{\epsilon})})$ cost, i.e. the costs are rising but not by a constant factor.

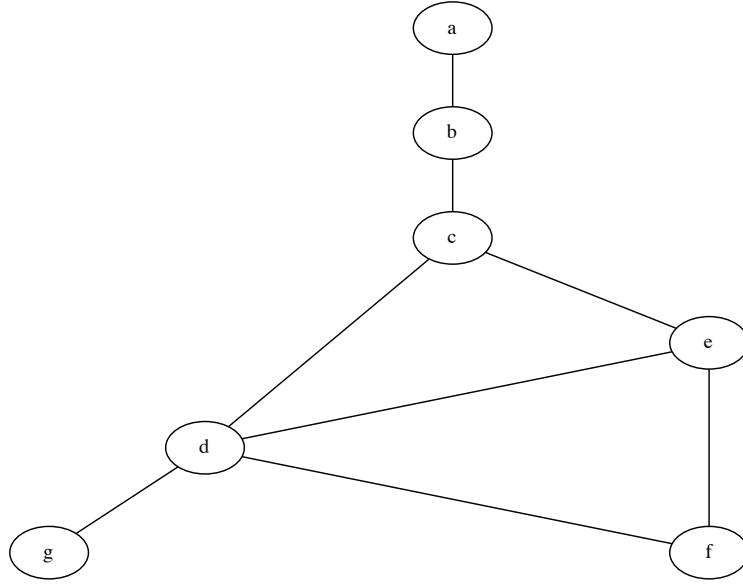
Example 7.2

$O((1/\epsilon)^2 * n^3)$ fully polynomial. $\epsilon = \frac{\epsilon}{4} O(16 * (\frac{1}{\epsilon}^2 * n^3))$.

7.1 Heuristic Approaches

Vertex Cover: Given an undirected graph $G = (V, E)$ without parallel edges and without loops, we look for a minimal set V' of nodes such that each edge has at least one endpoint in V' .

Example



Algorithm 1 APPROX-VERTEX-COVER($G = (V, E)$)

```

1:  $C \leftarrow \emptyset$ 
2:  $E' \leftarrow E$ 
3: while  $E' \neq \emptyset$  do
4:   Choose an edge  $\{u, v\}$  in  $E'$ .
5:    $C \leftarrow C \cup \{u, v\}$ 
6:   Remove all edges from  $E'$  with the end point  $u$  or  $v$ .
7: end while
  
```

- i) $\{b, c\}$ Remove from E' : $\{b, a\}, \{c, d\}, \{c, e\}, \{b, c\}$
- ii) $e, f\}$ Remove from E' : $\{e, f\}, \{e, d\}, \{d, f\}$
- iii) $\{d, g\}$ $C = \{b, c, e, f, d, g\}$, $E' = \emptyset$ twice as big as the optimum one.

Theorem 7.1

APPROX-VERTEX-COVER is a polynomial time 2-approximation ($\rho(n)$) algorithm.

Proof. 1. The algorithm runs in polynomial time in the size of the input $G = (V, E)$, because every edge and every node is inspected at most once.

2. Let A be the set of edges chosen in line 4 of the algorithm. In order to cover the edges in A ,

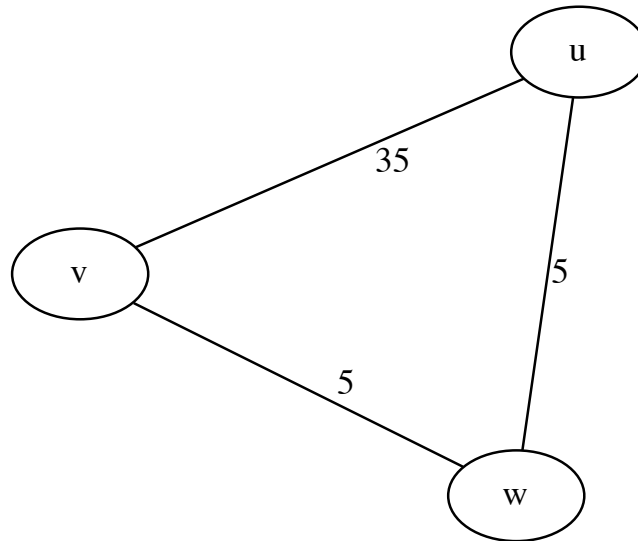
each vertex cover for G , in particular an optimal vertex cover c^* , must contain at least one endpoint of each edge in A . No two edges in A share an end point. Hence no two edges in A are covered by the same node in c^* , hence $|c^*| \geq |A|$. In line 4 an edge is chosen, the endpoints of which are not in C and hence:

$|C| = 2|A| = 2|c^*|$, where C is the calculated set cover and c^* is the optimal vertex cover. \square

7.1.1 Traveling Salesman

Given a complete undirected graph $G = (V, E)$ and a cost function $c : V \times V \leftarrow \mathbb{N}$, find a cycle which visits every node exactly once and is of least cost, where : Let $A \subseteq E$ then $cost(A) = \sum_{e \in A} c(e)$
 c satisfies the triangle inequality if $c(u, v) \leq c(u, w) + c(w, v)$ for any v, u, w .

Example



Algorithm 2 APPROX-TSP-TOUR($G = (V, E), c$)

- 1: Chose a node $r \in V$.
 - 2: Calculate the minimum cost spanning tree starting from r using Prim's algorithm.
 - 3: Let H be the list of nodes that is obtained traversing the tree T in preorder.
 - 4: **return** H
-

Theorem 7.2

APPROX-TSP-TOUR is a polynomial time 2-approximation algorithm for TSP satisfying the triangle inequality.

Proof. 1. Polynomial cost: Because Prim and preorder have polynomial cost.

2. Let H^* be an optimal tour for G . We remove one edge in this cycle, the result is a spanning tree for G , called \tilde{T} . The $cost(\tilde{T}) \leq cost(H^*)$ because $c(e) \geq 0$. A full preorder run through a tree is obtained if, after visiting a subtree of x , x is visited again. \square

Example

a b c b h b a d e f e g e d a

Every edge of the tree T occurs twice in this list. (TSP tree of Prim).

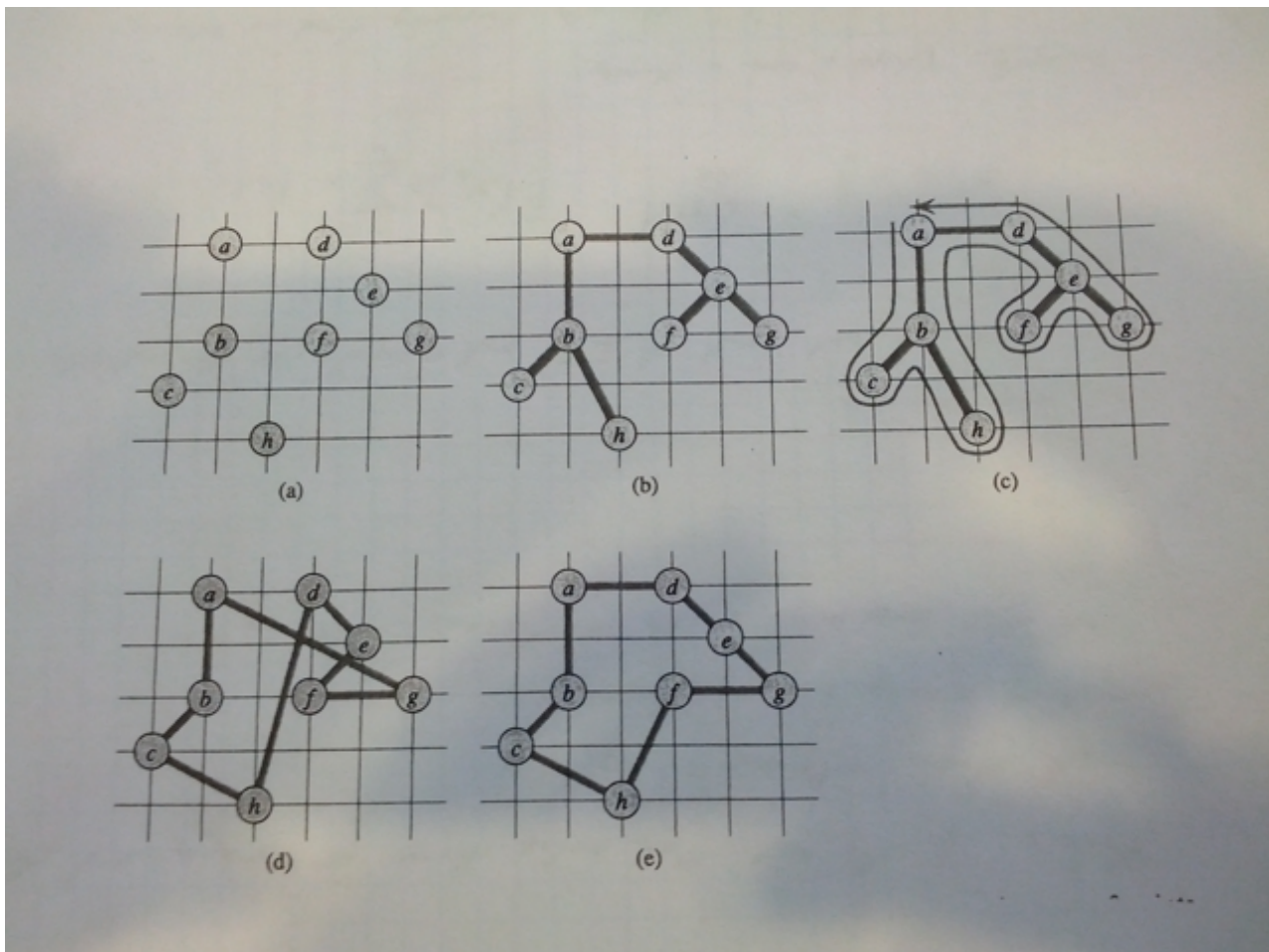
$$c(W) = 2 * c(T) \leq 2 * c(\tilde{T}) \leq 2 * cost(H^*)$$

W visits nodes more than once. We remove these repeated visits and use the triangle inequality to estimate the costs.

a b c a h ... \leadsto a b c h ...

The result: a b c h d e f g.

This is the preorder-run through T . $c_S(H) \leq Cost(W) \leq 2 * cost(H^*)$ where $c_S(H)$ is the result of the algorithm and H^* is optimal.



7.2 Probabilistic Analysis and randomized algorithms

Assistant search

- 1 best = 0
- 2 for $i = 1 \dots n$ do
 - interview candidate i
 - if candidate i is better than best
 - then best := i
 - employ candidate i (fire previous candidate)

Analysis of the cost that arise from interviews and hiring.

cost of an interview c_I (low)

cost of hiring: c_E (high)

Cost that arise in the worst case $O(n * c_I + m * c_E)$ if m people are hired and

$O(n * c_I + n * c_E)$ if n people are hired

worst case: if candidates show up in the order: worst first, ... best last

Indicator Variable Let S be a set and $A \subseteq S$ and execute the indicator variable.

$$X(s) = \begin{cases} 1, & \text{if } s \in A \\ 0, & \text{else} \end{cases}$$

$$E[x_1] = \sum_x x \Pr(X_A = x) = \Pr(A)$$

Example Throw a coin n times. $X(s) = \begin{cases} 1, & \text{if in the } i\text{-th throwing head is shown} \\ 0, & \text{otherwise} \end{cases}$

$S = (a_1, \dots, a_n) : a_i \in H, N$, Let X be the random variable that describes how often “H” appears when you throw n -times.

$$X : \{H, N\}^n \rightarrow \mathbb{N} \quad X = \sum x_i$$

The expected number of “H”

$$E[X] = E[\sum x_i] = \sum_{i=1}^n E[x_i] = \sum_{i=1}^n \frac{1}{2} = \frac{n}{2}$$

Example Hiring:

$$X_i = \begin{cases} 1, & \text{if the candidate is hired} \\ 0, & \text{else} \end{cases}$$

$$X = \sum_{i=1}^n x_i \text{ describes how many candidates are hired}$$

$$E[X] = E[\sum_{i=1}^n x_i] = \sum_{i=1}^n E[x_i]$$

$E[X_i]$ = Probability that the i -th candidate is hired.

Candidate i is hired if this person is better than the first $i-1$ candidates. If the order in which people are chosen is random, each of the first i candidates is the best with same probability. So

$$E[x_i] = \frac{1}{i}$$

$$E[X] = \sum_{i=1}^n \frac{1}{i} = \ln n + O(1)$$

Lemma 7.1

If the order in which the candidates appear is random then the average cost of finding a suitable employee is $O(c_E * \ln n + C_I * n)$

Question: How can we know that the order is random ?

7.2.1 Randomized Algorithms

In order to guarantee randomness of the input, we randomize it.

Example 7.3

Hiring Given the list of the n candidates, use random numbers to select a candidate for an interview. $RANDOM(a, b)$ produces an integer z with $a \leq z \leq b$ where all numbers in this interval are equally probable. $RANDOM(0, 1)$ produces 0 or 1 with equal probability $\frac{1}{2}$. $RANDOM(3, 7)$ produces 4 with probability $\frac{1}{5}$.

Randomized-Assistant Search

1. Permute the list of candidates using a random number generator
2. best = 0
3. ... as before

Lemma 7.2

The expected cost for hiring using the randomized assistant search algorithm is $O(c_E * \ln n + C_I * n)$

How to randomize an input of n elements $[A[1...n]]$?

PERMUTE-BY-SORTING(A)

1. $n = \text{length}(A)$
2. for $i = 1...n$ do
3. $P[i] = \text{RANDOM}[1, n^3]$ (something arbitrary bigger than n)
4. Sort the array A using P as sort keys
5. return A

RANDOMIZE-IN-PLACE(A)

1. $n = \text{length}(A)$
2. for $i = 1...n$ do
3. swap $A[i] \leftrightarrow A[\text{RANDOM}(i, n)]$

»»»» 537543df52b95855cfa2f207fc7482bd87d8b8b6

Appendix 4: Application of general matching

4.3 Two processor scheduling

Given:

- i) Two identical agents/processors
- ii) Collection of n jobs or tasks together with a directed acyclic graph with n nodes which correspond to the jobs, that describe the precedence among the jobs.

Example 4.4

2 Graphics missing

How can the jobs be scheduled on the two processors such that the jobs are completed as quickly as possible?

Solution:

using matching!

But:

for three processors the problem is NP-Complete.

To solve the two processors problem we proceed as follows. We construct a graph G^* that has the same nodes as G and there is an edge x,y in G^* if and only if there is no path from x to y in G and no path from y to x in G .

1. Observation:

if we have a schedule we can obtain a matching. If the schedule is optimal the matching is of maximal cardinality.

2. Interesting:

We can use matching of maximal cardinality to produce a schedule that is optimal.

Let S be a set of nodes with indegree 0. Let M^* be a matching of max. cardinality for G^* . Apply the following rules repeatedly:

- i) If there is an unmatched node in S then schedule it. Remove it from G .
- ii) If there is a pair of nodes (jobs) in S that is matched in M^* then we schedule this pair and delete the two nodes (corresponding to the jobs) in G (i.e. that an edge in M^* starts/ends at the nodes of the pair).

«« Start Michi

If neither rule 1 nor rule 2 applies and there are still jobs left to be scheduled then we know that S contains only nodes that are matched and for every node in S its partner (M^*) is not in S .

$$J_1 \in S \text{ --- } M^* \text{ --- } J'_1 \notin S$$

$$J_2 \in S \text{ --- } M^* \text{ --- } J'_2 \notin S$$

Observation:

\nexists path J_1 to J_2

«« End Michi

«« Start Norman

If none of the above rules is applicable anymore we know that S contains only nodes that are matched, otherwise rule 1 would be applicable.

Moreover each job in S is matched to a job not in S , otherwise we could schedule the matched pair.

$$J_2 \notin S \text{ --- } e \in M^* \text{ --- } J'_2 \in S \rightarrow J'_1 \notin S \text{ --- } e \in M^* \text{ --- } J_1 \in S$$

To continue the scheduling we consider the following:

Observation i):

There is no path from J'_1 to J'_2 because otherwise there could be the path $J_2 \rightarrow J'_1 \rightarrow J'_2$ and J_2, J'_2 could not be matched in M^* .

Observation ii):

There is no path from J_1 to J_2 for the same reason.

Observation iii):

There is no path from J_2 to J_1 because $\text{indegree}(J_1) = 0$ and there are no cycles in G .

Observation iv):

case1: If there is no path from J'_2 to J'_1 , then we can substitute in M^* the pairs $\{J_2, J'_2\}$ and $\{J_1, J'_1\}$ by $\{J_1, J_2\}$ and $\{J'_1, J'_2\}$ in G^* . Now $\{J_1, J'_2\}$ is $\in M^*$ and can be scheduled according to rule ii).

case2: If there is a path from J'_2 to J'_1 we repeat the same argumentation as before.

The process terminates, because the graph G is finite and cycle free.

«« End Norman