Aalto University

Lecture Notes - Part I

Graphs Problems I

Fernando Dias, Philine Schiewe and Piyalee Pattanaki

January 29, 2024

CHAPTER 1
Introduction

1.1 DEFINITIONS

Here are a few definitions to start the course. First, **combinatorial**:

- *adjective*
- relating to selecting a given number of elements from a larger number without regard to their arrangement.

Now optimization (or optimization, whichever spelling is your favourite):

- *noun*
- the action of making the best or **most effective** use of a **situation** or **resource**.

As any part of optimization, it can be achieved by either analyzing/Visualizing properties of functions / extreme points or by applying numerical methods. Finally, optimization has important applications in fields such as economics, statistics, bioinformatics, machine learning, and artificial intelligence.

1.2 MATHEMATICAL PROGRAMMING AND OPTIMIZATION

In this course, optimization is viewed as the core element of mathematical programming, which is a central OR modelling paradigm. It can be simply defined using three major concepts: **variables**, **domain** and **functions**.

Variables correspond to decisions/points of interest (business decisions, parameter definitions, settings, geometries, among others). In our formulations, it will be the values where changes will be applied, and the goal is to find the best values, according to each particular problem. Limiting which values each variable can assume, the **domain** which represents constraints and limitations (such as logic, design, engineering, etc.). **Objective functions** (which represent performance and quality measurements) are used to evaluate which variable has the best value, considering the limitations present in the constraints.

However, mathematical programming has many applications in fields other than OR, which causes some confusion. In this course, we will study mathematical programming in its most general form: both constraints and objectives are nonlinear functions.

1.3 TYPES OF MATHEMATICAL OPTIMIZATION MODELS

As in any field of optimization, the following rule of thumb is always valid:

> *The simpler are the assumptions which define a type of problem, the better are the methods to solve such problems.*

For this course (and optimization in general), the following notations are useful:

- $x \in \mathbb{R}^n$: vector of (decision) variables x_j, $j = 1, \ldots, n$;
- $f : \mathbb{R}^n \to \mathbb{R} \cup \{\pm\infty\}$ - objective function;
- $X \subseteq \mathbb{R}^n$: ground set (physical constraints);
- $g_i, h_i : \mathbb{R}^n \to \mathbb{R}$: constraint functions;
- $g_i(x) \leq 0$ for $i = 1, \ldots, m$: inequality constraints;
- $h_i(x) = 0$ for $i = 1, \ldots, l$: equality constraints.

Our goal will be to solve variations of the general problem P:

$$\begin{aligned} (P): \ \min\ & f(x) \\ \text{s.t. } & g_i(x) \leq 0, i = 1, \ldots, m \\ & h_i(x) = 0, i = 1, \ldots, l \\ & x \in X. \end{aligned}$$

Which applies to any sub-field of optimization:

- **Linear programming (LP):** linear $f(x) = c^\top x$ with $c \in \mathbb{R}^n$; constraint functions $g_i(x)$ and $h_i(x)$ are affine ($a_i^\top x - b_i$, with $a_i \in \mathbb{R}^n$, $b \in \mathbb{R}$); $X = \{x \in \mathbb{R}^n : x_j \geq 0, j = 1, \ldots, n\}$.
- **Nonlinear programming (NLP):** some (or all) of the functions f, g_i or h_i are nonlinear;
- **(Mixed-)integer programming ((M)IP):** LP where (some of the) variables are binary (or integer). $X \subseteq \mathbb{R}^k \times \{0, 1\}^{n-k}$
- **Mixed-integer nonlinear programming (MINLP):** MIP+NLP.

In this course, we might face **any** of the previous sub-field, but the major change is that variables can be **discretized** (binary or integer).

1.4 GRAPHS

Some **useful definitions** for this course are:

- **graph** $G = (V, E, \psi)$: a powerful tool used in **discrete mathematics** and **graph theory**, where objects are represented in the form of "relation";
- **undirected** graph are sub-categories of graphs where the direction of interaction does not matter.
  - vertices V (or nodes and points);
  - edges E (or links, arcs and line);

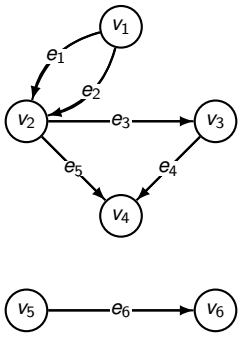
  - function $\psi\colon E \to \{X \subseteq V\colon |X| = 2\}$

- **directed** (where the edge direction is crucial) graph $G = (V, E, \psi)$
  - vertices V
  - edges E
  - function $\psi\colon E \to \{(v, w) \in V \times V\colon v \neq w\}$

- Edges e can have a value associated with it: $\to w \longrightarrow f_{uv}$ called **weight** or **flow**;

- in practice: $e = \{u, v\}$, $e = (u, v)$ respectively, $G = (V, E)$

Remark: E can contain multiple parallel edges.



Several structures can be extracted from graphs, especially based on **edge progression**. Considering W in G from u_1 to u_{k+1}, as an **edge progression** with the following progression:

- sequence $[u_1, a_1, u_2, \ldots, u_k, a_k, u_{k+1}]$ with $k \geq 0$
- $a_i = \{u_i, u_{i+1}\} \in E(G)$
- e.g. $[v_3, e_3, v_2, e_2, v_1, e_1, v_2, e_3, v_3, e_4, v_4]$

Generally, **walks** encompasses any edge progression. It can be separated in **closed** and **open** walks. For the former, a walk is considered an open walk if the starting and ending nodes are different, i.e. the starting node and the finishing are different. At the same time, the latter is a closed walk if the starting and ending nodes are identical, i.e. if a walk starts and ends at the same node, then it is said to be a closed walk.

- edge progression with $a_i \neq a_j$, $1 \leq i < j \leq k$
- e.g. $[v_2, e_2, v_1, e_1, v_2, e_3, v_3, e_4, v_4]$

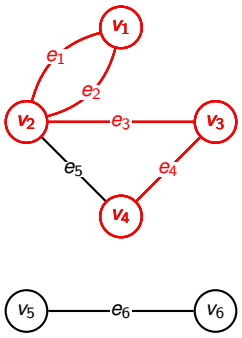
However, it can be decomposed into smaller definitions:

- trail: an open walk in which **no edge is repeated**;
- circuit: **closed** trail;

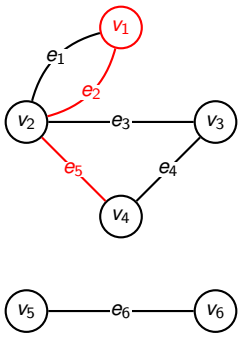
- cycle: same **starting** and **ending** node.

Another case is a **path**, which is a walk with **no repeating** nodes. For a **path** P in G from u_1 to u_{k+1}, $u_1 - u_{k+1}$ path:

- graph $(\{u_1, \ldots, u_{k+1}\}, \{a_1, \ldots, a_k\})$ with $[u_1, a_1, u_2, \ldots, u_k, a_k, u_{k+1}]$ walk and $u_i \neq u_j$, $1 \leq i < j \leq k+1$
- e.g. $[v_1, e_1, v_2, e_3, v_3, e_4, v_4]$

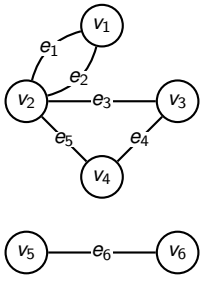


Finally, reachability is a concept in which v is reachable from u if there is a $u - v$ path in G and a graph is **connected** if there is a $u - v$ path in G for all $u, v \in V(G)$.



1.5 ALGORITHMS

For testing connective, the more straightforward approach is a visual representation is available, such as, for example:



If visual tools are not available, there are a few usual computational representations:

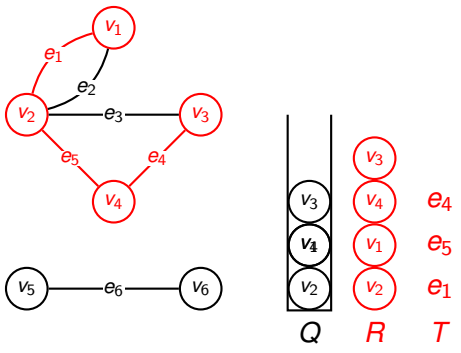
incidence matrix	adjacency matrix	adjacency list
$A \in \{0,1\}^{\lvert V\rvert \times \lvert E\rvert}$, $a_{v,e} = \begin{cases} 1, & \text{if } v \in e \\ 0, & \text{if } v \notin e \end{cases}$	$A \in \mathbb{Z}^{\lvert V\rvert \times \lvert V\rvert}$, $a_{v,w} = \lvert\{e = \{v,w\} \in E\}\rvert$	$L = [\ell(v) \colon v \in V]$, $\ell(v) = [e \colon e = \{u,v\} \in E]$
$\begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 0 & 2 & 0 & 0 & 0 & 0 \\ 2 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$	$\ell(v_1) = [e_1, e_2]$ $\ell(v_2) = [e_1, e_2, e_3, e_5]$ $\ell(v_3) = [e_3, e_4]$ $\ell(v_4) = [e_4, e_5]$ $\ell(v_5) = [e_6]$ $\ell(v_6) = [e_6]$
$O(\lvert V\rvert\lvert E\rvert)$	$O(\lvert V\rvert^2)$	$O(\lvert E\rvert \log \lvert V\rvert)$

The easiest algorithm to verify connectivity is via **DFS** (**Depth First Search**):

Algorithm: DEPTH FIRST SEARCH (DFS)

Input: undirected graph G, vertex $s \in V(G)$
Output: tree $(R, T) \subseteq G$, R reachable from s

1 set $R := \{s\}$, $Q := \{s\}$ and $T = \emptyset$;
2 **if** $Q = \emptyset$ **then return** R, T;
3 **else** $v :=$ last vertex added to Q;
4 choose $w \in V(G) \setminus R$ with $\{v, w\} \in E(G)$;
5 **if** *there is no such* w **then**
6 set $Q := Q \setminus \{v\}$ and **go to** 2
7 set $R := R \cup \{w\}$, $Q := Q \cup \{w\}$, $T := T \cup \{\{v, w\}\}$, **go to** 2;



The concept of the algorithm is as follows:

- suppose $w \in V(G) \setminus R$ is **reachable** from s

$\Rightarrow$ P is $s - w$ path with $\{x, y\} \in E(P)$, $x \in R$, $y \in V(G) \setminus R$

$\Rightarrow$ x is added to Q in line 7

$\Rightarrow$ Algorithm does not stop before x is removed from Q (line 6)

$\Rightarrow$ there is no $w \in V(G) \setminus R$ with $\{v, w\} \in E(G)$ ↯

In terms of runtime, for each node, the incident edges are considered; therefore, the runtime depends on the storage of graphs. If *adjacency lists* are used, the runtime is $O(m) = O(|E(G)|)$.

Analogous to DFS, there is also **BFS** (**Breadth First Search**) with the following algorithm:

Algorithm: BREADTH FIRST SEARCH (BFS)

Input: undirected graph G, vertex $s \in V(G)$
Output: tree $T) \subseteq G$

```
1 set Q := {s} and T = {s};
2 while Q ≠ ∅ do
3     v := first vertex in Q
4     set Q := Q \ {v}
5     while v has a neighbour not in T do
6         w := first neightbour of v not in T
7         set Q := Q ∪ {w}
8         set T := T ∪ {{v, w}}
```

CHAPTER 2
Paths and Trees

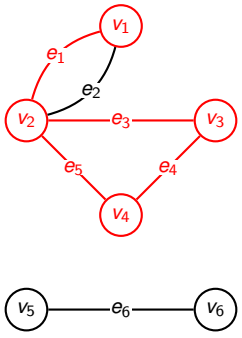
2.1 A FEW MORE DEFINITIONS

Recalling the definitions from the previous lecture, we can further improve the definitions of **paths** and **cycles**. For a path P in G from u_1 to u_{k+1} (as an **edge progressio**n):

- Graph $(\{u_1, \ldots, u_{k+1}\}, \{a_1, \ldots, a_k\})$ with $[u_1, a_1, u_2, \ldots, u_k, a_k, u_{k+1}]$ walk and $u_i \neq u_j$, $1 \leq i < j \leq k+1$
- e.g. $[v_1, e_1, v_2, e_3, v_3, e_4, v_4]$

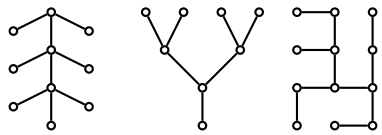
For **cycles** such C in G:

- graph $(\{u_1, \ldots, u_k\}, \{a_1, \ldots, a_k\})$ with $[u_1, a_1, u_2, \ldots, u_k, a_k, u_1]$ (closed) walk, $k \geq 2$ and $u_i \neq u_j$, $1 \leq i < j \leq k$
- e.g. $[v_2, e_3, v_3, e_4, v_4, e_5, v_2]$
- connected if there is a $u - v$ path in G for all $u, v \in V(G)$



2.2 NEWER DEFINITIONS

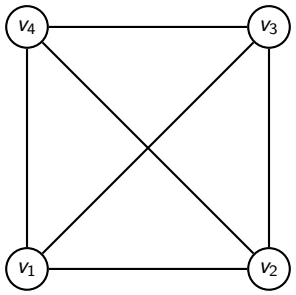
A graph G without a cycle is called *forest*, while a *connected* graph G without a **cycle** is called *tree*.



Now, let $G = (V, E)$ undirected graph with $|V| = n$. Then the following are equivalent:

1. G is a tree, i.e., connected and cycle-free.
2. G is cycle-free and has $n - 1$ edges.
3. G is connected and has $n - 1$ edges.
4. G is minimally connected (removing an edge $\Rightarrow$ not connected anymore).
5. G is maximally cycle-free (adding an edge $\Rightarrow$ cycle).
6. G contains a unique $u - v$ path for any pair of vertices $u, v \in V$.

Let $G = (V, E)$ undirected graph. $T = (V, E')$ with $E' \subseteq E$ is a **spanning tree** of G iff T is a tree. Hence, G is connected if it contains a spanning tree. Let $K_n = (V, E)$ be the complete graph with $|V| = n$ vertices, i.e., for any $u, v \in V$ the edge $\{u, v\} \in E$ exists. Then the number of spanning trees in K_n is n^{n-2}.



2.3 FINDING PATHS

The most useful instance of paths is to **identify the shortest path** in a graph. Finding the **minimum path length** between **two nodes** is trivial, and via **BFS**, it can be easily applied. At the same time, finding the **minimum path length** between **a node and all the others** is also trivial and **BFS** apply to each node individually would suffice.

Challenge: finding the **minimum-cost path** from a node to all the other in a **weighted** graph.

A **weighted graph** is a graph where all the edges have a specific value. It can also named as a **flow network**.

Definition 1 (Flow network) *A tuple $G = (V, E, f)$ is said to be a* flow network *if (V, E) where for every edge $(u, v) \in E$ we have an associated positive integer* flow value *f_{uv}.*

It also satisfying *conservation of flow* for every $v \in V \setminus \{s, t\}$, where s is an unique source and t is unique sink.

$$\sum_{(u,v)\in E} f_{uv} = \sum_{(v,w)\in E} f_{vw}. \tag{2.1}$$

Therefore, the goal is to calculate the shortest path from a node to each other vertices. Unfortunately, BFS will not suffice (because the shortest path may not have the fewest edges).

Alternative: Dijkstra's algorithm.

Edsger Dijkstra (1930-2002) was a Dutch computer scientist, programmer, software engineer, and science essayist and very influential in Computer Science and Discrete Mathematics. One of this most famous quotes is (which is encapsulated in his most famous algorithm):

"Simplicity is a prerequisite for reliability."

Figure 2.1: Edsger W. Dijkstra

Speaking of algorithm, it is **general ide**a for Dijkstra's approach is as follows:

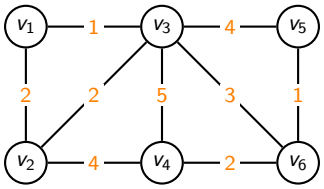
1. Iteratively **increase** the "set of nodes with known shortest distances";
2. Any node **outside** this set will have a "best distance so far";
3. Update the **"best distance so far"** until add all nodes to set.

The resulting algorithm is:

Algorithm: DIJKSTRA'S ALGORITHM

Input: undirected, connected graph G, weights $c\colon E(G) \to \mathbb{R}$, nodes V, source s

```
1  d_v distance to reach node v
2  p_v node predecessor to node v
3  Q ← ∅ set of "unkown distance" nodes.
4  for each node v in V do
5      d_v ← ∞
6      p_v ← FALSE
7      add v in Q
8  d_s ← 0
9  while Q ≠ ∅ do
10     u ← node in Q with min d_u
11     remove u from Q
12     for each neighbor v of u still in Q do
13         d ← d_u + c_uv
14         if alt < d_v then
15             d_v ← alt
16             p_v ← u
17 return d_v,p_v
```



In terms of runtime, this algorithm, when implemented to its best, has a runtime to $O(m + n \cdot \log(n)$, where m is the amount of edges and n is the number of nodes.

Alternatively, there is also an integer linear programming which can be applied (although not recommend):

$$\min \sum_{(u,v)\in E} f_{uv} x_{uv} \tag{2.2a}$$

subject to: (2.2b)

$$\sum_{(s,v)\in E} x_{sv} = 1, \tag{2.2c}$$

$$\sum_{(u,t)\in E} x_{ut} = 1, \tag{2.2d}$$

$$\sum_{(u,v)\in E} x_{uv} - \sum_{(v,w)\in E} x_{vwi} = 0, \tag{2.2e}$$

$$x_{uv} \in \{0,1\}, \qquad \forall (u,v) \in E \tag{2.2f}$$

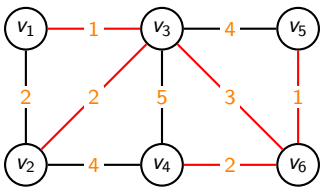
Constraints (2.2c) and (2.2d) ensures that a path **starts** in the source and **ends** in the sink, while constraint (2.2e) guarantees that **intermediary** nodes have a **single** edge in and a **single** edge out. The objective minimizes the total combined **weight** of the edges in that path.

2.4 MINIMAL SPANNING TREES

For spanning trees, the goal is to find an algorithm for a minimum spanning tree (MST). First, formally establishing the problem:

Instance: An undirected, connected graph G, weights $c\colon E(G) \to \mathbb{R}$.

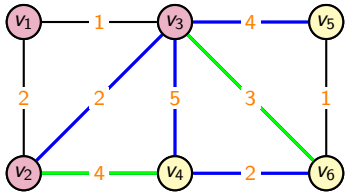
Task: Find a spanning tree T in G of minimum weight.



The **optimality conditions** for such a problem are as follows:

Theorem 1 *Let (G, c) be an instance of the MST problem and T a spanning tree in G. Then the following are equivalent:*

1. *T is optimal.*
2. *For every $e = \{x, y\} \in E(G) \setminus E(T)$, no edge on the $x - y$ path in T has higher cost than e.*
3. *For every $e \in E(T)$, e is a minimum cost edge of $\delta(V(C))$, where C is a connected component of $T - e$.*
4. *We can order $E(T) = \{e_1, \ldots, e_{n-1}\}$ such that for each $i \in \{1, \ldots, n-1\}$ there exists a set $X \subseteq V(G)$ such that e_i is a minimum cost edge of $\delta(X)$ and $e_j \notin \delta(X)$ for all $j \in \{1, \ldots, i-1\}$.*



$\delta(X) = \{\{u, v\} \in E : u \in X, v \notin x\}$
edges from X to $V(G) \setminus X$

This problem has been studied to extension, and **two algorithms** have been proposed from the literature. The starting point comes from the following theorem:

Theorem 2 *Let $G = (V, E)$ undirected graph with $|V| = n$. Then the following are equivalent:*

1. *G is a tree, i.e., connected and cycle-free.*
2. *G is cycle-free and has $n - 1$ edges.*
3. *G is connected and has $n - 1$ edges.*
4. *G is minimally connected (removing an edge $\Rightarrow$ not connected anymore).*
5. *G is maximally cycle-free (adding an edge $\Rightarrow$ cycle).*
6. *G contains a unique $u - v$ path for any pair of vertices $u, v \in V$.*

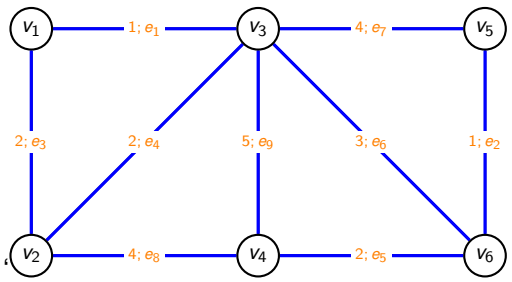
2.4.1 Kruskal's Algorithm

The first option (in no particular order) is **Kruskal's algorithm** (proposed by Joseph Kruskal in 1956).

Algorithm: KRUSKAL'S ALGORITHM
Input: undirected, connected graph G, weights $c : E(G) \to \mathbb{R}$
Output: spanning tree T of minimum weight

```
1 sort edges such that c(e_1) ≤ c(e_2) ≤ ... ≤ c(e_m)
2 set T := (V(G), ∅)
3 for i := 1 to m do
4     if T + e_i contains no cycle then
5         set T := T + e_i
6 return T
```

In the following picture, it is shown how each step is calculated:



Execution:
$E(T) = \emptyset$
$E(T) = \{e_1\}$
$E(T) = \{e_1, e_2\}$
$E(T) = \{e_1, e_2, e_3\}$
$E(T) = \{e_1, e_2, e_3, e_5\}$
$E(T) = \{e_1, e_2, e_3, e_5, e_6\}$

Test:
$e_1 = \{v_1, v_3\}$ ✓
$e_2 = \{v_5, v_6\}$ ✓
$e_3 = \{v_1, v_2\}$ ✓
$e_4 = \{v_2, v_3\}$ ✗$\rightsquigarrow$ cycle
$e_5 = \{v_4, v_6\}$ ✓
$e_6 = \{v_3, v_6\}$ ✓
$e_7 = \{v_3, v_5\}$ ✗ $\rightsquigarrow$ cycle
$e_8 = \{v_2, v_4\}$ ✗ $\rightsquigarrow$ cycle
$e_9 = \{v_3, v_5\}$ ✗ $\rightsquigarrow$ cycle

In terms of **correctness**, T is **maximally cycle-free** (no further edge can be added), which is contemplated as a **tree**. For each edge $e_i = \{x, y\} \in E(G) \setminus E(T)$:

- $T + e_i$ contains a **cycle** in line 4;
- there **exists** a $x - y$ path in T at this point;
- all edges in T have **lower** weight than e_i at this point.

Hence, T is **MST**.

In terms of **runtime**:

- sorting edges: $O(m \log m)$
- loop lines 3-5: checking m times for cycles
- checking for cycle containing $e = \{u, v\}$
  - DFS starting from u with at most n edges, check if v is reachable: $O(n)$

$\rightsquigarrow$ total running time: $O(mn)$

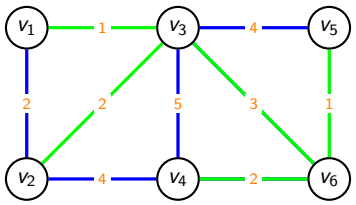
To sum up, Kruskal's algorithm is **guaranteed** to be cycle-free and greedily add edges until *maximally cycle-free*.

2.4.2 Prim's Algorithm

An alternative is Prim's algorithm (developed in 1930 by Czech mathematician **Vojtěch Jarník** and later re-discovered and republished by computer scientists **Robert C. Prim** in 1957 and **Edsger W. Dijkstra** in 1959).

Algorithm: PRIM'S ALGORITHM
Input: undirected, connected graph G, weights $c\colon E(G) \to \mathbb{R}$
Output: spanning tree T of minimum weight
1 choose $v \in V(G)$
2 set $T := (\{v\}, \emptyset)$
3 **while** $V(T) \neq V(G)$ **do**
4 choose an edge $e \in \delta_G(V(T))$ of minimum weight
5 set $T := T + e$
6 **return** T

Using the following figure as an example:



Execution:
$V(T) = \{v_1\}$
$E(T) = \emptyset$
$V(T) = \{v_1, v_3\}$
$E(T) = \{\{v_1, v_3\}\}$
$V(T) = \{v_1, v_3, v_2\}$
$E(T) = \{\{v_1, v_3\}, \{v_2, v_3\}\}$
$V(T) = \{v_1, v_3, v_2, v_6\}$
$E(T) = \{\{v_1, v_3\}, \{v_2, v_3\}, \{v_3, v_6\}\}$
$V(T) = \{v_1, v_3, v_2, v_6, v_5\}$
$E(T) = \{\{v_1, v_3\}, \{v_2, v_3\}, \{v_3, v_6\}, \{v_5, v_6\}\}$
$V(T) = \{v_1, v_3, v_2, v_6, v_5, v_4\}$
$E(T) = \{\{v_1, v_3\}, \{v_2, v_3\}, \{v_3, v_6\}, \{v_5, v_6\}, \{v_4, v_6\}\}$

Test:
$\delta_G(V(T)) =$
$\{\{v_1, v_2\}, \{v_1, v_3\}\}$
$\{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_3, v_5\}, \{v_3, v_6\}\}$
$\{\{v_2, v_4\}, \{v_3, v_4\}, \{v_3, v_5\}, \{v_3, v_6\}\}$
$\{\{v_2, v_4\}, \{v_3, v_4\}, \{v_3, v_5\}, \{v_4, v_6\}, \{v_5, v_6\}\}$
$\{\{v_2, v_4\}, \{v_3, v_4\}, \{v_4, v_6\}\}$

Regarding **runtime**, the best performance can be achieved as $O(m \log n)$.

Finally, a **ILP formulation** for MST (known as **Martin formulation**):

$$\min \sum_{(u,v)\in E} f_{uv} x_{uv} \tag{2.3a}$$

subject to: (2.3b)

$$\sum_{(u,v)\in E} x_{uv} = n - 1, \tag{2.3c}$$

$$y^k_{uv} + y^k_{vi} = x_{uv}, \qquad (u,v) \in E, k \in V \tag{2.3d}$$

$$\sum_{k\in V\setminus\{(u,v)\}} y^v_{uk} + x_{uv} = 1, \qquad \forall (i,j) \in E \tag{2.3e}$$

$$x_{uv}, y^k_{uv}, y^k_{vu} \in \{0,1\}, \qquad \forall (u,v) \in E, k \in V \tag{2.3f}$$

In the formulation above, y^k_{uv} denotes that edge (u, v) is in the spanning tree and node k is on the side of v.

The **constraint** (2.3d) guarantees that if $(u, v) \in E$ is selected into the tree, any node $k \in V$ must be on **either** side of v (depending if $y^k_{uv} = 1$ or $y^k_{vu} = 1$). If $(u, v) \in E$ is **not** in the tree, any node k **cannot** be on the side of v or u.

The final constraint ensures that if $(u, v) \in E$ is in the tree, edges (u, k) which connects u are on the side of u. If it is **not** in the tree, there **must be and edge** (u, k) such that v is **on the side of** k ($y^v_{uk} = 1$ for some k).

CHAPTER 3

Flows and Cuts

In **graph theory**, **flow network** is a **directed** graph $G = (V, E)$ where each edge has a **capacity** $u\colon E \to \mathbb{R}_+$ and each edge receives a **flow** $f\colon E \to \mathbb{R}_+$, where the amount of flow allowed in each edge cannot surpass its capacity ($f(e) \leq u(e)$, $e \in E$). Hence, the *excess* of a flow f at $v \in V$:

$$\mathrm{ex}_f(v) := \sum_{e \in \delta^-(v)} f(e) - \sum_{e \in \delta^+(v)} f(e)$$

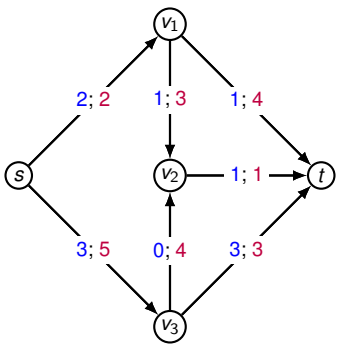
$\delta^-(v) = \{e \in E\colon e = (u, v)\}$ incoming edges

$\delta^+(v) = \{e \in E\colon e = (v, u)\}$ outgoing edges

The flow in this type of graph also have the satisfy **flow conservation** which state that:

Definition 2 *The total net flow entering a node v is zero for **all nodes** in the network except the source s and sink t.*

This can be also expressed based on the vale of flow through through a node. If f satisfies *flow conversation rule* at v, then $\mathrm{ex}_f(v) = 0$. When **all nodes** satisfy flow conservation $\mathrm{ex}_f(v) = 0$ for all $v \in V$, we express such behaviour as *circulation*. Finally, in a path between the source s and the sink t, the *s-t-flow*: $\mathrm{ex}_f(s) \leq 0$, $\mathrm{ex}_f(v) = 0$ for all $v \in V \setminus \{s, t\}$, in which the *value of s-t-flow* can be calculated as $\mathrm{value}(f) = -\mathrm{ex}_f(s) = \mathrm{ex}_f(t)$.



$\mathrm{value}(f) = 5$

A **cut** in graph theory corresponds to a **partition** of the nodes in a graph splitting them into **disjoint subsets**. For example, see Figure 3.1.

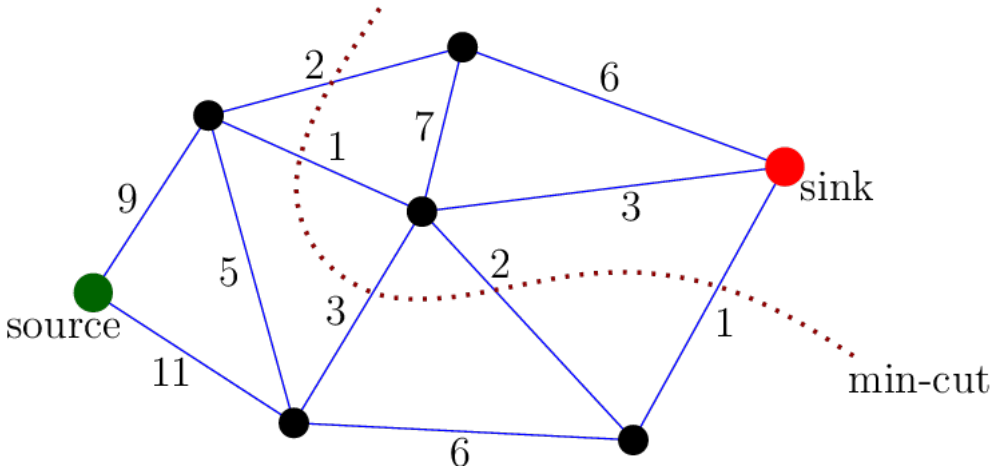


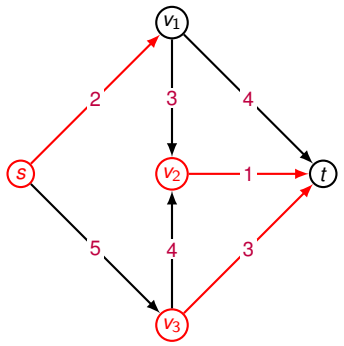
Figure 3.1: Example of a cut in a graph

A specific type of cut is a *s-t-cut* $\delta^+(S)$ where $S \subseteq V$ and $s \in S, t \notin S$. Therefore:

$$\delta^+(S) = \{e = (u, v) \in E : u \in S, v \in V \setminus S\}$$

The **capacity** of such cut can be expressed as:

$$u(\delta^+(S)) = \sum_{e \in \delta^+(S)} u(e)$$



capacity $u(\delta^+(\{s, v_2, v_3\})) = 6$

3.1 WEEK DUALITY

Using the definitions of flows and cuts, we can establish the following conclusion:

Lemma 1 *For any $S \subseteq V$ with $s \in S, t \notin S$ and any s-t-flow f:*

1. $\text{value}(f) = \sum_{e \in \delta^+(S)} f(e) - \sum_{e \in \delta^-(S)} f(e)$
2. $\text{value}(f) \leq u(\delta^+(S))$

Proof 1 *From the flow conservation for $v \in S \setminus \{s\}$:*

$$\begin{aligned}
\text{value}(f) &= -\text{ex}_f(s) \\
&= \sum_{e \in \delta^+(s)} f(e) - \sum_{e \in \delta^-(s)} f(e) \\
&= \sum_{v \in S} \Big(\sum_{e \in \delta^+(v)} f(e) - \sum_{e \in \delta^-(v)} f(e) \Big) \\
&= \sum_{e \in \delta^+(S)} f(e) - \sum_{e \in \delta^-(S)} f(e)
\end{aligned}$$

This can also expressed as:

$$0 \leq f(e) \leq u(e)$$

3.2 MAXIMUM FLOWS AND MINIMAL CUTS

Once again, the task of find which flow and which cuts a graph can accept is **not challenging**. However, whenever **optimal values** (either minimal or maximal) are required, the configuration of such problems becomes challenging.

First, we state both problems:

Problem 1 *Maximum Flow Problem (MaxFlow) Given a flow network represent as a digraph $G = (v, E)$ with capacities u and unique source and unique sink s and t respectively, such that $s, t \in V$.*
*The goal is to find an s-t-flow of **maximum** value.*

Problem 2 *Minimum Cut Problem (MinCut) Given a flow network represent as a digraph $G = (v, E)$ with capacities u and unique source and unique sink s and t respectively, such that $s, t \in V$.*
*The goal is to find an s-t-cut of **minimum capacity**.*

Although those two problems might seem **unrelated** or even **contradictory**, they can be directly connected via the following lemmas:

Lemma 2 *Let $G = (V, E)$ be a digraph with capacities u and $s, t \in V$. Then*

$$\max\{\text{value}(f) \colon f \text{ } s\text{-}t\text{-flow}\} \leq \min\{u(\delta^+(S)) \colon \delta^+(S) \text{ } s\text{-}t\text{-cut}\}.$$

Lemma 3 *Let $G = (V, E)$ be a digraph with capacities u and $s, t \in V$. Let f be an s-t-flow and $\delta^+(S)$ be an s-t-cut. If*

$$\text{value}(f) = u(\delta^+(S))$$

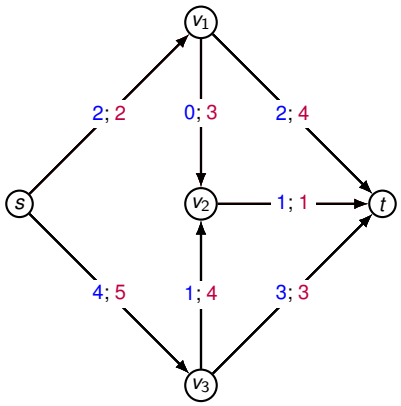
then f is a maximal flow and $\delta^+(S)$ is a minimal cut.

Hence, a single algorithm is enough to solve **both** problems.
Remark: in combinatorics, many problems can be expressed as another. This is a key point for future lectures.

3.3 IDEA FOR FINDING MAXIMAL FLOWS

If there exists non-saturated s-t-path ($f(e) < u(e)$ for all edges), then the flow f can be increased along this path. This means that if the path is not satured, more flow can be put into that path.

However, non-existence of such a path does not guarantee optimality.

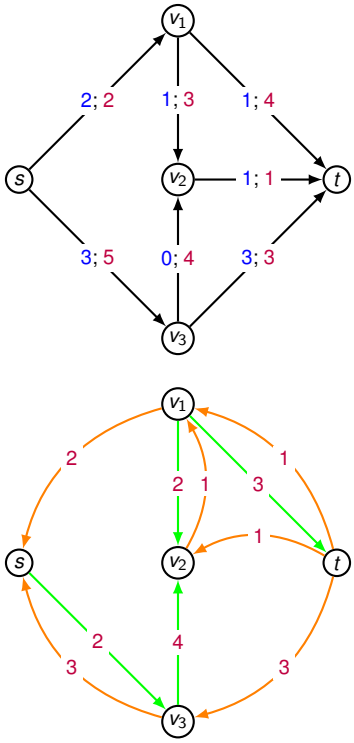


$\text{value}(f) = 4$ $\text{value}(f) = 5$ $\text{value}(f) = 6$
$u(\delta^+(\{s, v_2, v_3\})) = 6$

In this context, we introduced another concept: **residual graphs**. Considering that $G = (V, E)$ is a digraph with capacities u, f be an s-t-flow, a residual graph is the graph $G_f = (V, E_f)$ with $E_f = E_+ \cup E_-$ and capacity u_f:

- *forward edges* $+e \in E_+$:
  for $e = (u, v) \in E$ with $f(e) < u(r)$, add $+e = (u, v)$ with *residual capacity* $u_f(+e) = u(e) - f(e)$
- *backward edges* $-e \in E_-$:
  for $e = (u, v) \in E$ with $f(e) > 0$, add $-e = (v, u)$ with *residual capacity* $u_f(-e) = f(e)$

Remark: G_f can have parallel edges even if G is simple.

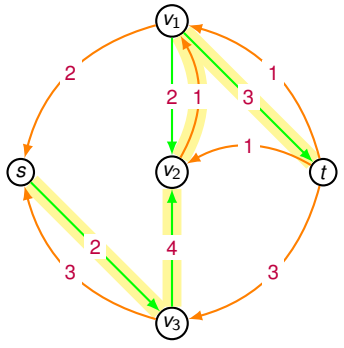


In addition, we can also define *f*-**augmenting paths**:

Definition 3 *An s-t-path P in* G_f *is called* augmenting path. *The value:*

$$\blacksquare_f(P) = \min_{a \in E(P)} u_f(a)$$

is called residual capacity *of P.*
Remark: $\blacksquare_f(P) > 0$ *as* $u_f(a) > 0$ *for all* $a \in E_f$.



$\blacksquare_f(P) = 1$

With this definition in mind, the following theorem is established.

Theorem 3 *An s-t-flow is optimal if and only if there exists no f-augmenting path.*

Proof idea:

$\Rightarrow$ P f-augmenting path. Construct s-t-flow

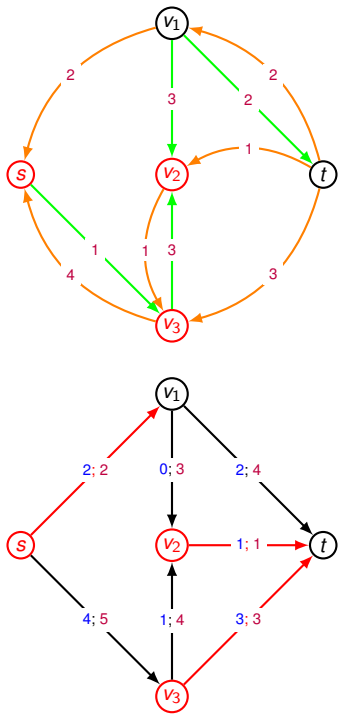
$$\bar{f}(e) = \begin{cases} f(e) + \blacksquare_f(P) & \text{if } +e \in E(P) \\ f(e) - \blacksquare_f(P) & \text{if } -e \in E(P) \\ f(e) & \text{otherwise} \end{cases}$$

with higher value.

Proof idea:

$\Leftarrow$ There exists no f-augmenting path. Consider s-t-cut $\delta^+(S)$ defined by connected component S of s in G_f. Show that

$$\text{value}(f) = u(\delta^+(S)).$$



With this previous theorem in mind, we can conclude that:

Theorem 4 (Ford and Fulkerson, 1956; Dantzig and Fulkerson, 1956)
In a digraph G with capacities u, the maximum value of an s-t-flow equals the minimum capacity of an s-t-cut.

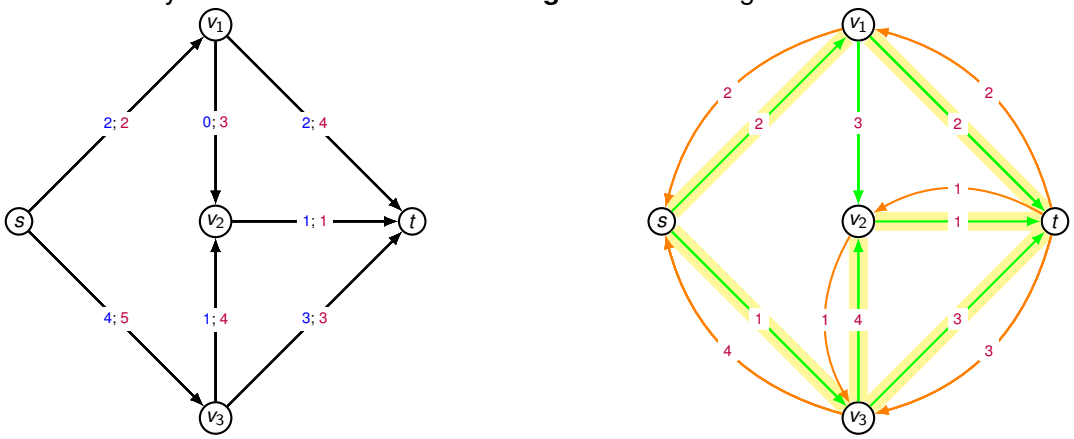
Algorithm: FORD-FULKERSON ALGORITHM

Input: digraph $G = (V, E)$, capacities $u\colon E \to \mathbb{Z}_+$, $s, t, \in V$
Output: maximal s-t-flow f

1 set $f(e) = 0$ for all $e \in E$
2 **while** *there exists f-augmenting path in G_f* **do**
3 choose f-augmenting path P
4 set $\blacksquare_f(P) = \min_{a \in E(P)} u_f(a)$
5 augment f along P by $\blacksquare_f(P)$
6 update G_f
7 **return** f

3.4 FINDING MAXIMAL FLOWS

The most common algorithm for maximum flow was first published by L. R. Ford Jr. and D. R. Fulkerson in in 1956. It is commonly known as **Ford-Fulkerson algorithm**. The algorithm is as follows:



$\blacksquare_f(P) = 3$
$\blacksquare_f(P) = 2$
$\blacksquare_f(P) = 1$

Analysing the previous algorithms allow us to infer a few details. Lines 1, 4, 5 and 6 can be calculated in **linear time** in terms the number of edges m in a graph. An efficient algorithm to apply in Line 3 is actually **DFS** (Depth-First Search) which is also **linear** in the number of edges m. The *WHILE* loop requires up to $n \cdot U$, where n is the number of nodes and U is $max_{e \in E} u(e)$. The entire algorithm has a runtime proportional to $O(n \cdot m \cdot U)$ (**polynomial**).
Remark: flow f is integer.

An improved version of this algorithm allows for **real values in the capacities**. In this case, for non-integer capacities, $\blacksquare_f$ can be arbitrarily small when P is not chosen carefully , resulting in a runtime $O(n \cdot m^2)$.

The resulting algorithm represent such adaption:

Last but not least, there is also linear programming formulation for this problem. See full model below:

Algorithm: EDMONDS-KARP ALGORITHM

Input: digraph $G = (V, E)$, capacities $u\colon E \to \mathbb{R}_+$, $s, t, \in V$
Output: maximal s-t-flow f

1 set $f(e) = 0$ for all $e \in E$
2 **while** *there exists f-augmenting path in G_f* **do**
3 choose f-augmenting path P with minimal number of edges
4 set $\blacksquare_f(P) = \min_{a \in E(P)} u_f(a)$
5 augment f along P by $\blacksquare_f(P)$
6 update G_f
7 **return** f

$$\max \quad \sum_{e \in \delta^+(s)} f_e \tag{3.1a}$$

$$\text{s.t.} \quad \sum_{e \in \delta^-(v)} f_e - \sum_{e \in \delta^+(v)} f_e = 0 \qquad v \in V \setminus \{s, t\} \tag{3.1b}$$

$$f_e \leq u(e) \qquad e \in E \tag{3.1c}$$

$$f_e \geq 0 \qquad e \in E \tag{3.1d}$$

The flow conservation flow conversation constraints (3.1b) are part of many LPs and IPs, e.g. for **shortest path**. The coefficient matrix of flow conversation constraints is **node-arc-incidence matrix** and it is **totally unimodular**, i.e., all extreme points are integer.

CHAPTER 4
Matching

As always, a definition at first:

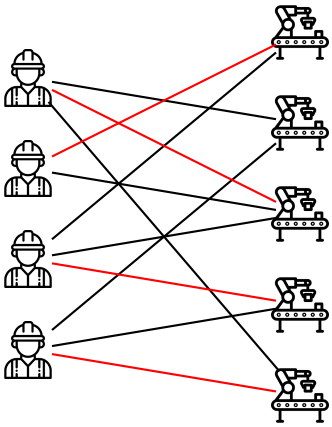
Definition 4 ***Matching*** *in an undirected graph is a set of edges without common vertices.*

Also known as **independent edge set**, this problem goal is to find a subset of the edges as a matching if each node appears in at most one edge of that matching.

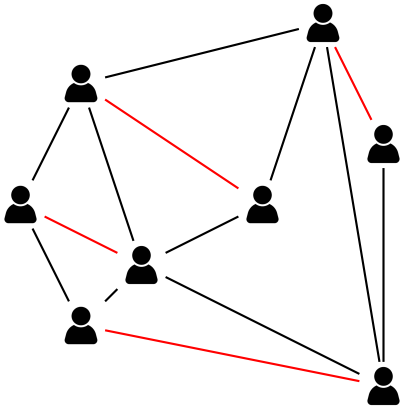
From an undirected graph $G = (V, E)$, $M \subset E$ is called *matching* if all $e \in M$ are pairwise disjoint, i.e., if the endpoints are different. In addition, $M \subset E$ is a *maximum matching* in G if M is a matching with highest cardinality, i.e.,

$$|M'| \leq |M| \quad \text{for all matchings } M'$$

Some illustrations as example:



Assignment different workers to different tasks in order that there is no conflict or overlapping.



Setting pairs for homework assignments.

For this problem, a simple integer linear programming formulation can be calculated:

$$\begin{aligned}
&\text{Maximize} && \sum_{e \in E} x_e \\
&\text{Subject to:} \\
&\sum_{e \in \delta(v)} x_e \leq 1 && \forall v \in V \\
&x_{ij} \in \{0,1\} && \forall e \in E
\end{aligned}$$

where $\delta(v)$ is the set of incident edges of $v \in V$, such that:

$$\delta(v) = \{e \in E : e = \{v, w\}\}$$

Like flow problems, we can also define ***M*-augmenting paths**. Let $G = (V, E)$ be an undirected graph and $M \subseteq E$ matching. A node $v \in V$ is said to be **covered** by M if $v \in e$ for some $e \in M$ and it is **exposed** by M if $v \notin e$ for all $e \in M$.

With those, two types of paths can be defined *M-alternating path P*, where edges $E(P)$ are alternately in M and not in M (or not in M and in M) and *M-augmenting path P* that is a special type of M-alternating path, where the first and last vertex exposed.

Remark: M-augmenting paths have odd number of edges.

According to Berge's Theorem:

Theorem 5 (Petersen (1891), Berge (1957)) *Let G be a graph with some matching M. Then M is the maximum if and only if there is no M-augmenting path.*

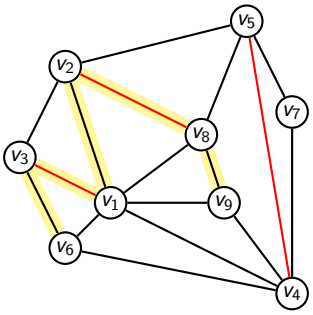
Proof 2 *Proof idea* $\Rightarrow$*: By contraposition: Let* $P = (v_0, e_1, \ldots, e_k, v_k)$ *be an M-augmenting path.*

- *by definition:* v_0, v_k *exposed*

$\Rightarrow$ $|E(P) \setminus M| = |E(P) \cap M| + 1$

$\Rightarrow$ *$M' = (M \setminus E(P)) \cup (E(P) \setminus M)$ is matching with $|M'| = |M| + 1$*

$\Rightarrow$ *M not maximum*



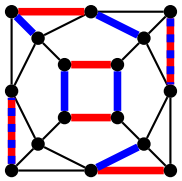
From this theorem, we can derive a few lemmas, such as

Lemma 4 *Let G be a graph with two matchings M, M'. Let $G' = (V, E' = M \blacksquare M')$, with symmetric difference*

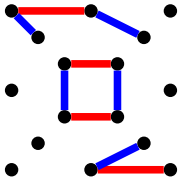
$$M \blacksquare M' = (M \cup M') \setminus (M \cap M').$$

Then, the connected components of G' are

- *isolated vertices*
- *cycles C with $|E(C)| \in 2\mathbb{N}$ where edges in C are alternately in M and M'*
- *paths $P = (v_0, e_1, \ldots, e_k, v_k)$ where edges are alternately in M and M'*



graph G



graph G'

Proof 3 *Proof idea: Let M, M' matchings:*

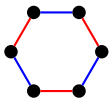
$$\begin{aligned} |\{e \in M \colon v \in e\}| &\leq 1, v \in V \\ |\{e \in M' \colon v \in e\}| &\leq 1, v \in V \\ \Rightarrow |\{e \in E' \colon v \in e\}| &\leq 2, v \in V \end{aligned}$$

If $g_{G'}(v) = |\{e \in E' \colon v \in e\}| = 2$: $\exists! e \in M \colon v \in e$ and $\exists! e \in M' \colon v \in e$.

- isolated vertices $v \rightsquigarrow g_{G'}(v) = 0$

•

- cycles C with $|E(C)| \in 2\mathbb{N} \rightsquigarrow g_{G'}(v) = 2$



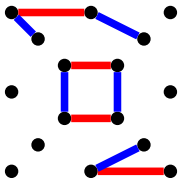
- paths $P = (v_0, e_1, \ldots, e_k, v_k) \rightsquigarrow g_{G'}(v_0) = 0 = g_{G'}(v_k) = 1, g_{G'}(v_i) = 2, 1 \leq i \leq k-1$

Another way to prove the same theorem is listed below:

Theorem 6 (Petersen (1891), Berge (1957)) *Let G be a graph with some matching M. Then M is the maximum if and only if there is no M-augmenting path.*

Proof 4 *Proof idea:*
By contraposition: Let M' be a matching with $|M'| > |M|$.
Construct G'.

$$
\begin{aligned}
|M'| > |M| &\Rightarrow |E' \cap M'| > |E' \cap M| \\
&\Rightarrow \exists P = (v_0, e_1, \ldots, e_k, v_k) \text{ with } e_1 \in M', e_k \in M' \\
&\Rightarrow v_0, v_k \text{ exposed by } M \\
&\Rightarrow P \ M\text{-augmenting path}
\end{aligned}
$$



graph G'

4.1 MAXIMUM MATCHING

With all of this in mind, the **resulting algorithm** can be expressed:

Algorithm: MAXIMUM MATCHING
Input: undirected graph $G = (V, E)$
Output: maximum matching M

```
1 set M = ∅
2 while there exists M-augmenting path in G do
3 |   choose M-augmenting path P
4 |   set M = (M \ E(P)) ∪ (E(P) \ M)
5 return M
```

In this algorithm, up to $\frac{|V|}{2}$ iterations are required. There is no obvious way to find an M-augmenting path. However, for bipartite graphs, the easier way is to find s-t-path in auxiliary graphs, while in general graphs, E**dmond's blossom algorithm** is the best approach. Nevertheless, such an algorithm is highly **complex** and has a **polynomial runtime**.

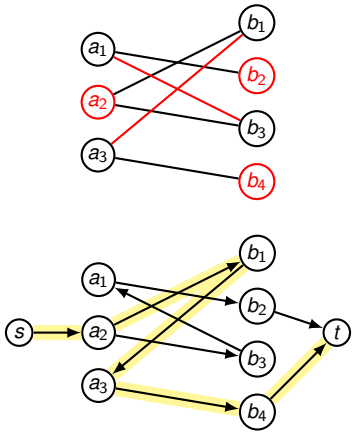
However, the challenge still remains on **finding M-alternating paths**. For bipartite graph $G = (V, E)$ with:

- $V = A \cup B, A \cap B = \emptyset$
- $E \subseteq \{\{a, b\} : a \in A, b \in B\}$

The easier approach is to construct **auxiliary directed graph** $G' = (V', E')$ with:

$$\begin{aligned} V' =& V \cup \{s, t\}, \quad s, t \notin V \\ E' =& \{(b, a) : \{a, b\} \in M, a \in A, b \in B\} \\ & \cup \{(a, b) : \{a, b\} \in E \setminus M, a \in A, b \in B\} \\ & \cup \{(s, a) : a \text{ exposed}, a \in A\} \\ & \cup \{(b, t) : b \text{ exposed}, b \in B\} \end{aligned}$$

Then, $\exists$ M-augmenting path in G if and only if $\exists$ s-t-path in G'.



The resulting **algorithm** encapsulates this procedure:

Algorithm: MAXIMUM MATCHING BIPARTITE GRAPHS
Input: undirected bipartite graph $G = (V, E)$
Output: maximum matching M

```
1 set M = ∅
2 construct G'
3 while there exists s-t-path in G' do
4     choose s-t-path P
5     set M = (M \ E(P)) ∪ (E(P) \ M)
6     update G'
7 return M
```

In order to construct G', it takes up to $O(n + m)$, where $n = |V|$ and $m = |E|$, due to no isolated nodes in G . The remaining $\frac{n}{2}$ iterations are divided into:

- finding P: $O(m)$
- updating M: $O(n)$
- updating G': $O(n)$

The final runtime is $O(nm)$.

4.1.1 Connection to MaxFlow

Solving matching can also be formulated as solving maximum flow. By constructing an auxiliary directed graph $G'' = (V'', E'')$ with:

$$\begin{aligned} V'' =& V \cup \{s, t\}, \quad s, t \notin V \\ E'' =& \{(a, b) \colon \{a, b\} \in E, a \in A, b \in B\} \\ & \cup \{(s, a) \colon a \in A\} \\ & \cup \{(b, t) \colon b \in B\} \end{aligned}$$

and capacity $u(e) = 1$ for all $e \in E''$. With that, G'' has maximal flow with value k if and only if G has a maximum matching of cardinality k.