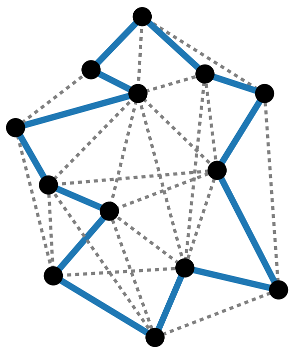
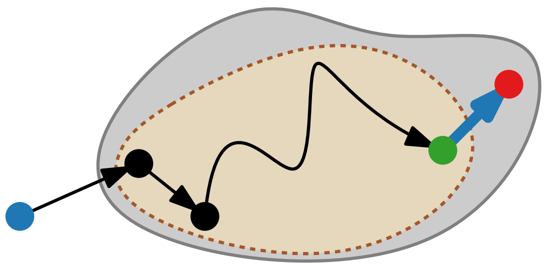
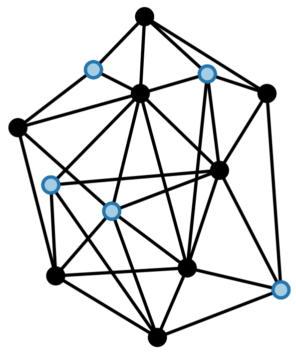
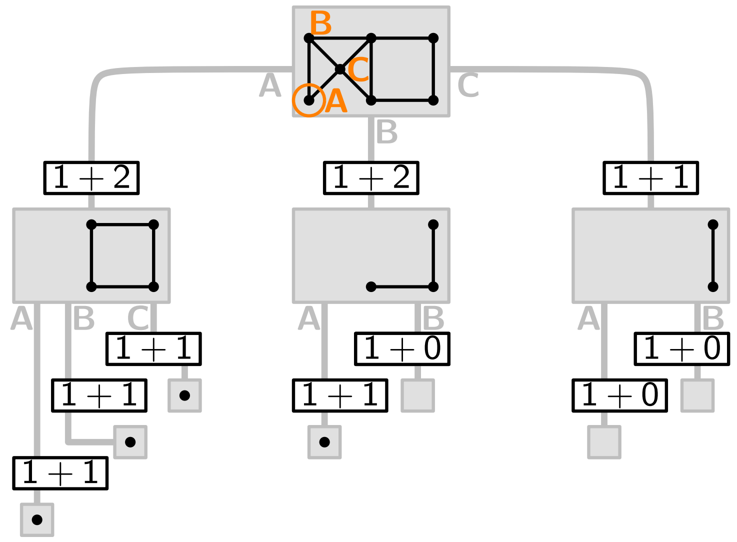
Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

# Advanced Algorithms

## Exact Algorithms for NP-Hard Problems

## Traveling Salesman Problem and Maximal Independent Set

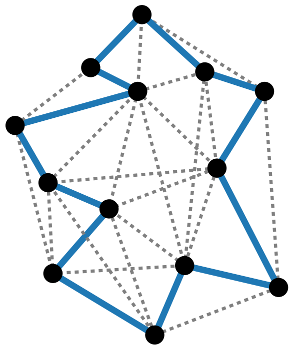






*Alexander Wolff*

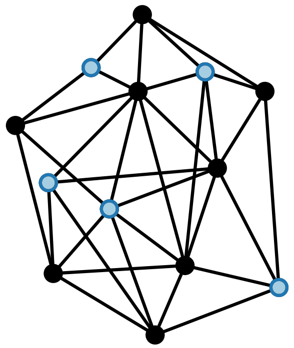
*Summer term 2026*

# Examples of NP-Hard Problems

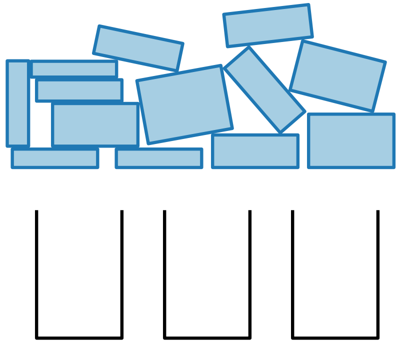
Many important (practical) problems are NP-hard, for example ...



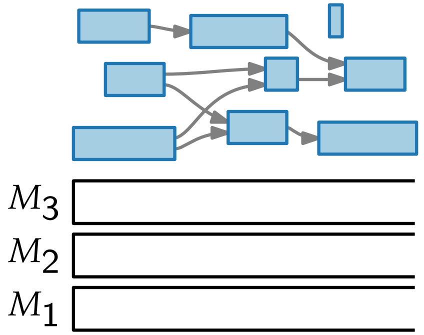
TSP



MIS



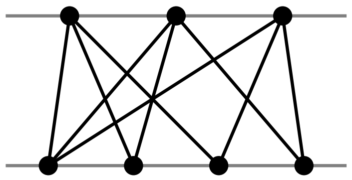
Bin Packing



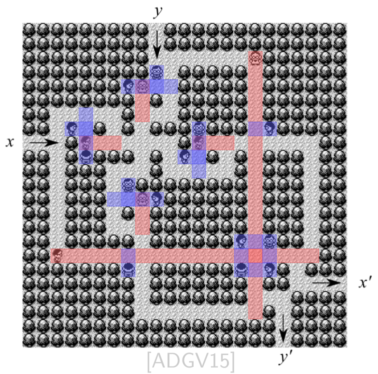

Scheduling

$$(x_1 \vee x_2 \vee \neg x_4) \wedge$$
$$(\neg x_2 \vee x_3 \vee \neg x_4) \wedge$$
$$(x_3 \vee x_7 \vee \neg x_8) \wedge$$
$$\ldots$$

SAT



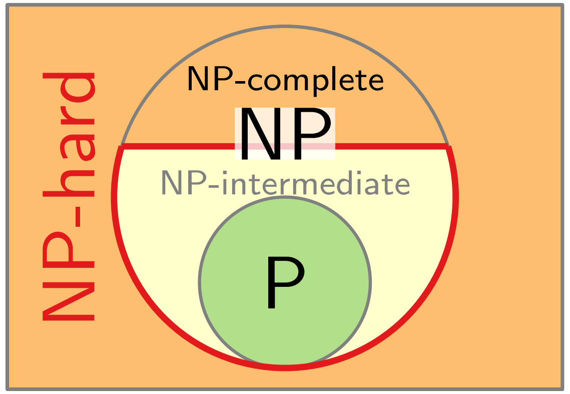
Graph Drawing




Games

...

# What is P, NP, and NP-Hardness?




- P is the complexity class that consists of all problems that can be solved in polynomial time – deterministically.
- NP is the complexity class that consists of all problems that can be solved in polynomial time – *non-deterministically*, i.e., a problem in NP can be solved in polynomial time by a hypothetical machine that can duplicate itself to try different paths to a solution in its computation.
- There is another, more accessible equivalent definition:
  A problem is in NP if the correctness of a solution can be verified in polynomial time.
- It is not proven yet, but all indications suggest that P $\neq$ NP.
- The hardest problems in NP are called *NP-complete*.
- All problems that are at least as hard as any NP-complete problem are called *NP-hard*.
  One can show NP-hardness by a polynomial-time reduction from an NP-hard problem.
- Assuming P $\neq$ NP, NP-hard problems cannot be solved in polynomial time.

# Misconceptions about NP-Hardness

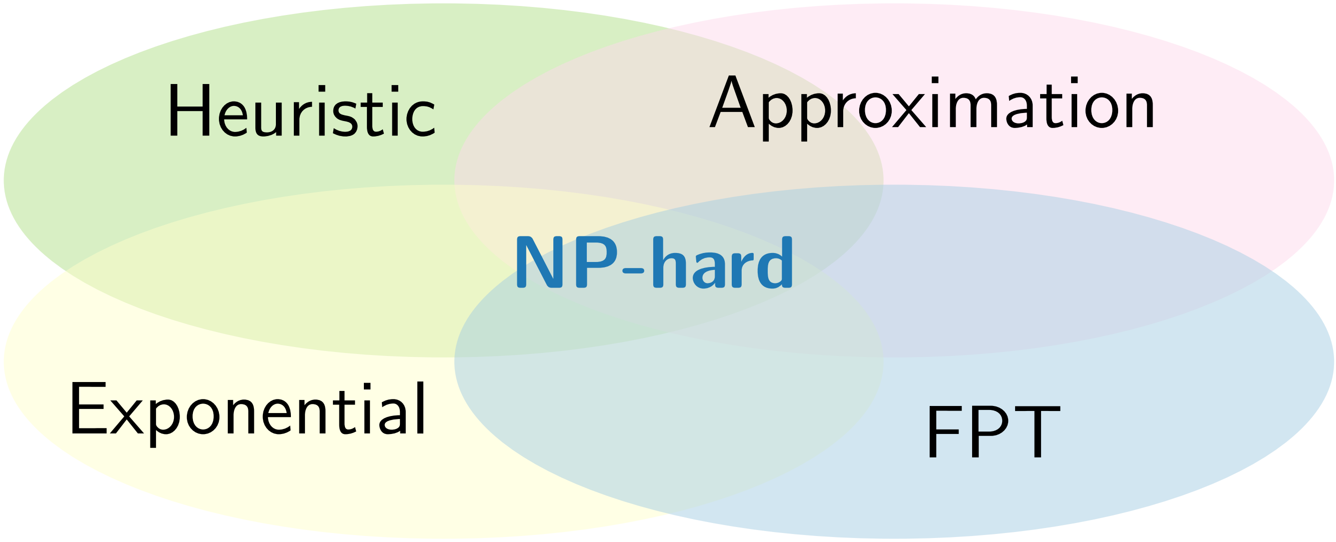
Common misconceptions [Mann '17]

- If similar problems are NP-hard, then the problem at hand is also NP-hard.
- Problems that are hard to solve in practice by an engineer are NP-hard.
- NP-hard problems cannot be solved optimally.
- NP-hard problems cannot be solved more efficiently than by exhaustive search.
- For solving NP-hard problems, the only practical possibility is the use of heuristics.

# Dealing with NP-Hard Problems

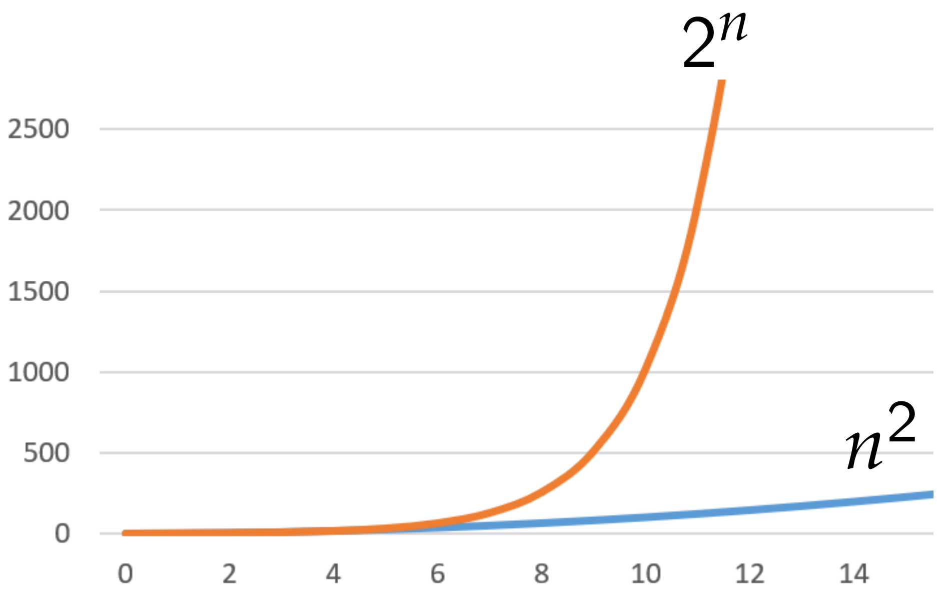
What should we do?

- Sacrifice optimality for speed
  - Heuristics (Simulated Annealing, Tabu-Search)
  - Approximation Algorithms (MST-Edge-Doubling, Christofides-Algorithm)
- Optimal Solutions
  - Exact exponential-time algorithms (with a better running time than just a brute-force algorithm)
  - Fine-grained analysis – parameterized algorithms




this lecture

# Motivation




efficient (polynomial-time)
vs.
inefficient (super-pol.time)

Exponential running time ... should we just give up?

- ... can be *"fast"* for medium-size instances:
  - "hidden" constants in polynomial-time algorithms:
    $2^{100}n > 2^n$ for $n \leq 100$
  - $n^4 > 1.2^n$ for $n \leq 100$
  - TSP solvable exactly for $n \leq 2000$ and specialized instances with $n \leq 85900$

# Motivation

Exponential running time ... maybe we need better hardware?

- Suppose an algorithm uses $a^n$ steps & can solve, within a fixed amount of time $t$, instances up to size $n_0$.
- Improving hardware by a constant factor $c$ only *adds a constant* (relative to $c$) to $n_0$:

$$a^{n_1} = c \cdot a^{n_0} \;\Rightarrow\; n_1 = \log_a c + n_0$$

- On the other hand –
  reducing the base of the runtime to $b < a$ results in a *multiplicative* increase:

$$b^{n_1} = a^{n_0} \;\Rightarrow\; n_1 = n_0 \cdot \log_b a$$

# Motivation

Exponential running time . . . but can we at least find exact algorithms that are faster than **brute-force** (trivial) approaches?

- TSP: The Bellman–Held–Karp algorithm runs in $\mathcal{O}(2^n n^2)$ time compared to an $\mathcal{O}(n! \cdot n)$-time brute-force search.

$\mathcal{O}^*$ hides polynomial factors in $n$ (see next slide)

- MIS: The algorithm of Tarjan & Trojanowski runs in $\mathcal{O}^*(2^{n/3})$ time compared to a trivial $\mathcal{O}(n2^n)$-time approach.
- Coloring: Lawler gave an $\mathcal{O}(n(1+\sqrt[3]{3})^n)$ algorithm compared to $\mathcal{O}(n^{n+1})$-time brute-force search.
- SAT: No better algorithm than trivial brute-force search known.

# $\mathcal{O}^*$-Notation

$$\mathcal{O}(1.4^n \cdot n^2) \subsetneq \mathcal{O}(1.5^n \cdot n) \subsetneq \mathcal{O}(2^n)$$

- Base of exponential part dominates $\rightsquigarrow$ polynomial factors can be neglected.

$$f(n) \in \mathcal{O}^*(g(n)) \Leftrightarrow \exists \text{ polynomial } p(n) \text{ with } f(n) \in \mathcal{O}(g(n)p(n))$$

- typical result

| Approach | Runtime in $\mathcal{O}$-Notation | $\mathcal{O}^*$-Notation |
|---|---|---|
| Brute-Force | $\mathcal{O}(2^n)$ | $\mathcal{O}^*(2^n)$ |
| Algorithm A | $\mathcal{O}(1.5^n \cdot n)$ | $\mathcal{O}^*(1.5^n)$ |
| Algorithm B | $\mathcal{O}(1.4^n \cdot n^2)$ | $\mathcal{O}^*(1.4^n)$ |

# Traveling Salesperson Problem (TSP)

**Input.** Distinct cities $\{v_1, v_2, \ldots, v_n\}$ with distances $d(v_i, v_j) \in \mathbb{Q}_{\geq 0}$;
directed, complete graph $G$ with edge weights $d$

**Output.** Tour of the traveling salesperson of minimum total length that visits all the cities and returns to the starting point;


i.e., a Hamiltonian cycle $(v_{\pi(1)}, \ldots, v_{\pi(n)}, v_{\pi(1)})$ of $G$ of minimum weight

$$\sum_{i=1}^{n-1} d(v_{\pi(i)}, v_{\pi(i+1)}) + d(v_{\pi(n)}, v_{\pi(1)})$$

**Brute-force.**

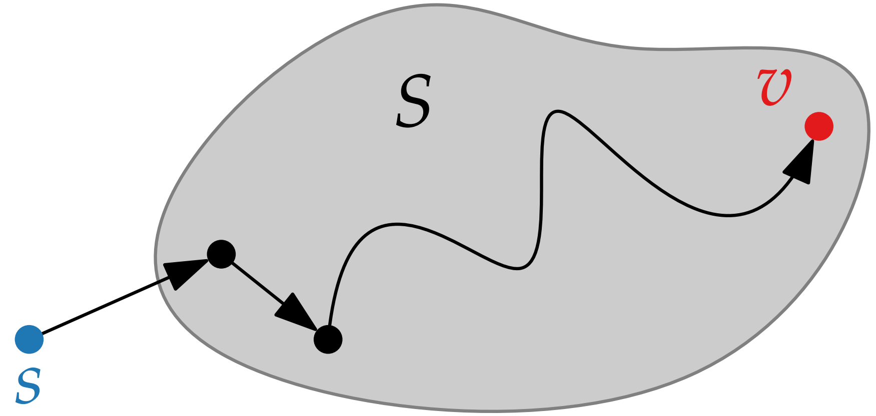
- Try all permutations and pick the one with smallest weight.
- Runtime: $\Theta(n! \cdot n) = n \cdot 2^{\Theta(n \log n)}$

# TSP via Dynamic Programming (Bellman–Held–Karp Algor.)

**Idea.**

- Dynamic programming means re-using optimal substructures (typically stored in a "table"). We store optimal partial tour lengths.
- Select a starting vertex $s \in V(G)$.
- For each $S \subseteq V(G) - s$ and $v \in S$, let:

  $\mathrm{OPT}[S, v]$ = length of a shortest $s$–$v$ path that visits precisely the vertices of $S \cup \{s\}$.




- Use $\mathrm{OPT}[S - v, u]$ to compute $\mathrm{OPT}[S, v]$.

Richard M. Karp

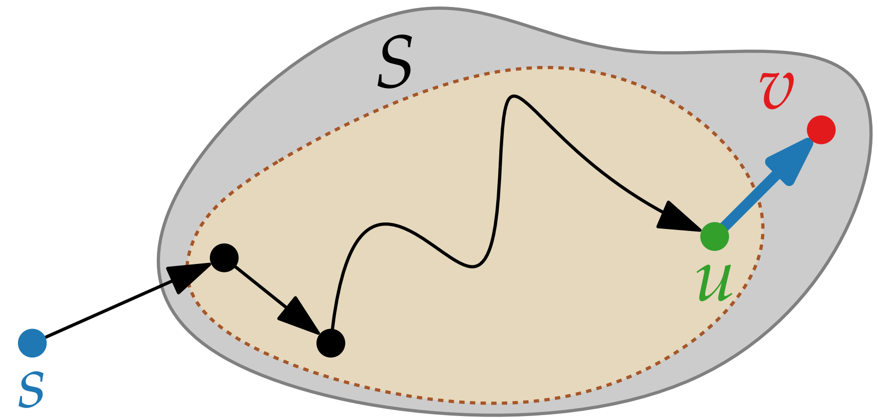
Richard E. Bellman

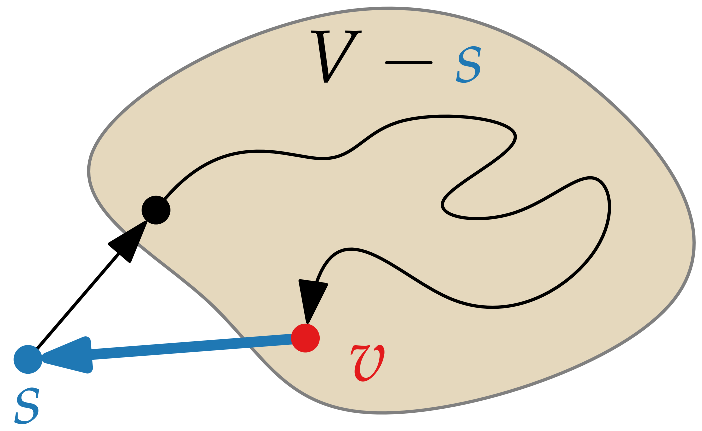
# TSP – Dynamic Programming

**Details.**

- The base case $S = \{v\}$ is easy: $\mathrm{OPT}[\{v\}, v] = d(s, v)$.
- For $|S| \geq 2$, compute $\mathrm{OPT}[S, v]$ recursively:

$$\mathrm{OPT}[S, v] = \min\{\mathrm{OPT}[S - v, u] + d(u, v) \mid u \in S - v\}$$




- After computing $\mathrm{OPT}[S, v]$, for each $S \subseteq V(G) - s$ and each $v \in V(G) - s$, the optimal solution is easily obtained as follows:

$$\mathrm{OPT} = \min\{\mathrm{OPT}[V(G) - s, v]\} + d(v, s) \mid v \in V(G) - s\}$$

# TSP – Dynamic Programming

## Pseudocode.

Bellmann–Held–Karp$(G, d)$:

**foreach** $v \in V(G) \setminus \{s\}$ **do**
  OPT$[\{v\}, v] = d(s, v)$

**for** $j = 2$ **to** $n - 1$ **do** $\Big\}\mathcal{O}(2^n)$
  **foreach** $S \subseteq V(G) \setminus \{s\}$ with $|S| = j$ **do**
    **foreach** $v \in S$ **do** $\}\mathcal{O}(n)$
      OPT$[S, v] = \min\{$ OPT$[S - v, u] + d(u, v) \mid u \in S - v \}$ $\Big\}\mathcal{O}(n)$

**return** $\min\{$ OPT$[V(G) - s, v] + d(v, s) \mid v \in V(G) - s \}$

- A shortest tour can be found by backtracking the DP table (as usual).

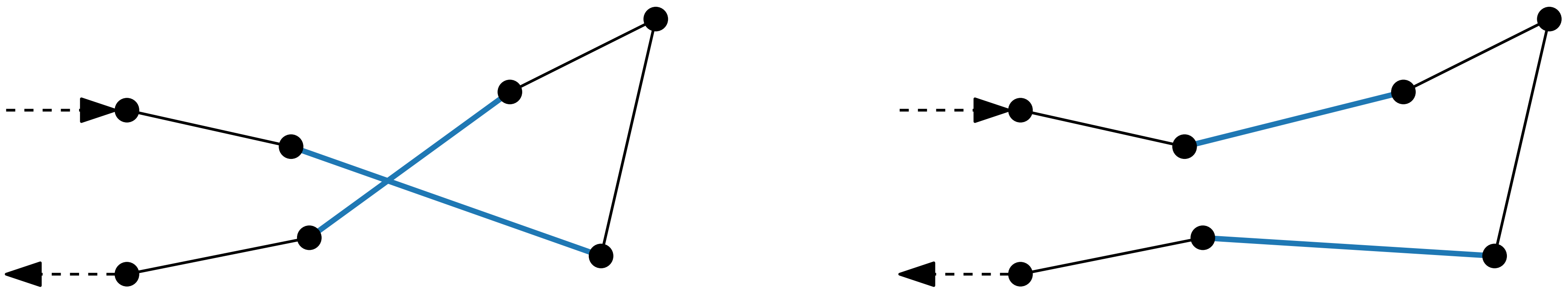
## Analysis.

- running time for the central for-loop is in $\mathcal{O}(2^n n^2) \subseteq \mathcal{O}^*(2^n)$
- Space usage in $\Theta(2^n \cdot n)$
- Or actually better? What table values do we need to store?

# TSP – Discussion

- DP algorithm that runs in $\mathcal{O}^*(2^n)$ time and $\mathcal{O}^*(2^n)$ space.
- Brute-force runs in $2^{\mathcal{O}(n \log n)}$ time and $\mathcal{O}(n^2)$ space.

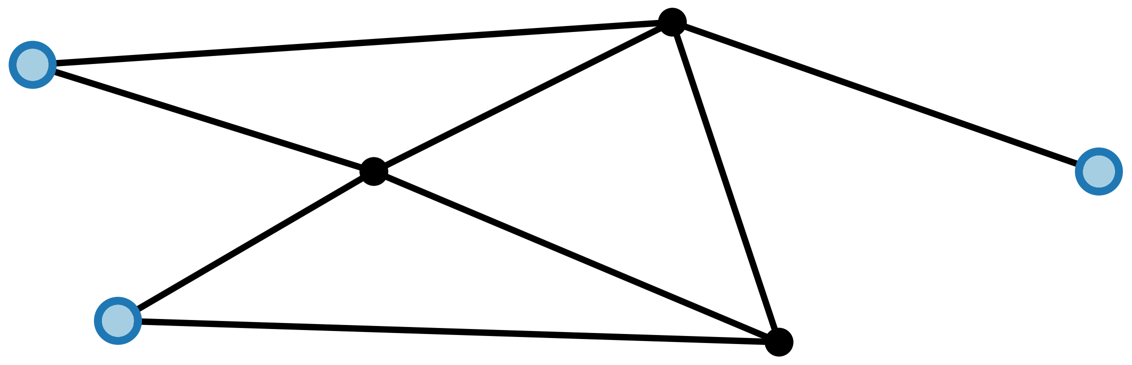
  $\Rightarrow$ Sacrifice space for speedup.
- Many variants of TSP: symmetric, assymetric, metric, vehicle routing problems, ...
- Metric TSP can easily be 2-approximated. (Do you remember how? $\to$ last lecture)
- Euclidian TSP is considered in the lecture *Approximation Algorithms*.
- In practice, one successful approach is to start with a greedily computed Hamiltonian cycle and then use 2-OPT and 3-OPT swaps to improve it.



# Maximum Independent Set (MIS)

**Input.** Graph $G$; let $n$ be the number of vertices of $G$.

**Output.** Maximum size **independent** set, i.e., a largest set $U \subseteq V(G)$ such that no pair of vertices in $U$ is adjacent in $G$.



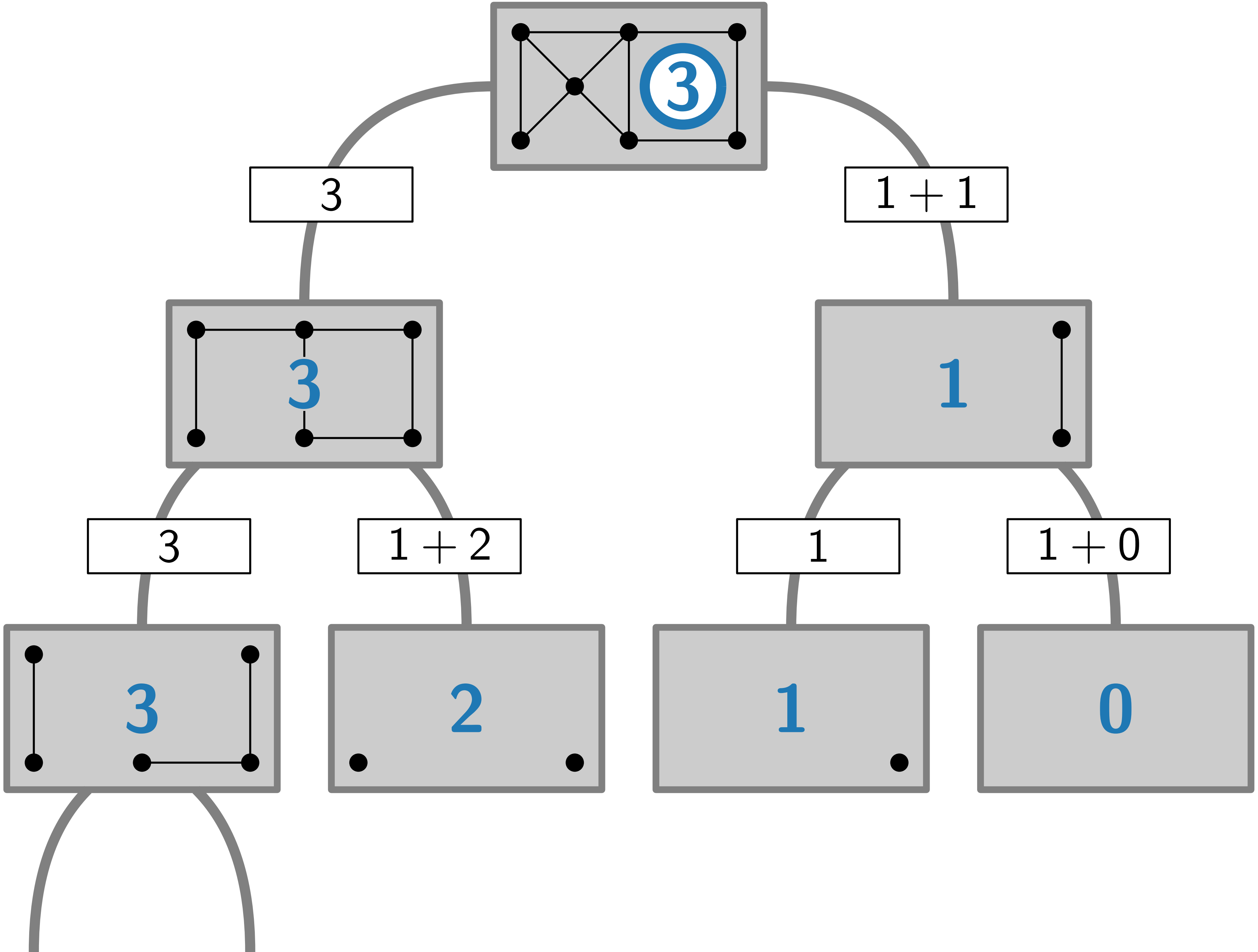
**Naive MIS branching.**

- Take a vertex $v$ – or don't take it!

NaiveMIS$(G)$:
  **if** $V(G) == \emptyset$ **then**
    **return** 0
  $v =$ arbitrary vertex in $V(G)$
  **return** max$\{$ NaiveMIS$(G - \{v\})$,
    $1 +$ NaiveMIS$(G - N(v) - \{v\})$ $\}$

**Brute-force.**

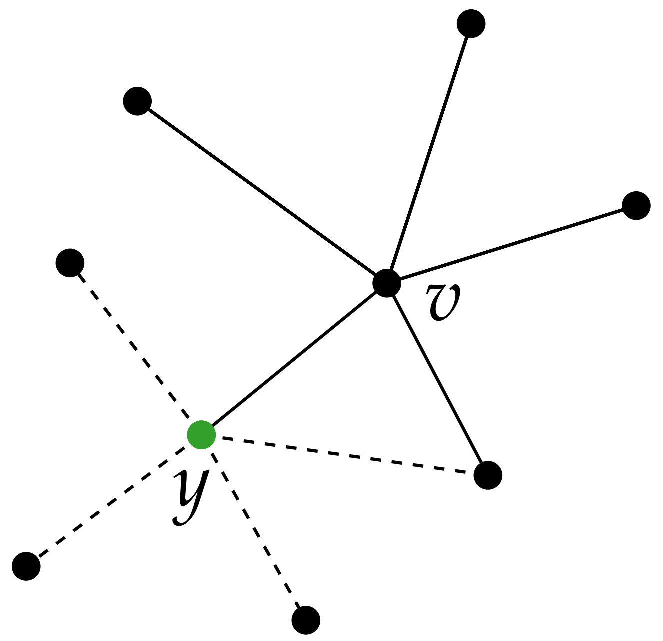
- Try all subets of $V(G)$.
- Runtime: $\mathcal{O}(2^n \cdot n)$


3
3
1 + 1
3
1
3
1 + 2
1
1 + 0
3
2
1
0

# MIS – Smarter Branching

**Lemma.**
Let $U$ be a maximum independent set in $G$.
Then, for every $v \in V(G)$:
1. $v \in U \Rightarrow N(v) \cap U = \emptyset$
2. $v \notin U \Rightarrow |N(v) \cap U| \geq 1$

Thus, $N[v] := N(v) \cup \{v\}$ contains some $y \in U$, and no other vertex of $N[y]$ is in $U$.




## Smarter MIS branching.

- For some vertex $v$, branch on vertices in $N[v]$.

SmarterMIS($G$):
  **if** $V(G) == \emptyset$ **then**
    **return** 0
  $v =$ vertex of minimum degree in $V(G)$
  **return** $1 + \max\{\mathrm{MIS}(G - N[y]) \mid y \in N[v]\}$

- Correctness follows from the lemma.
- We prove a runtime of $\mathcal{O}^*(3^{n/3}) = \mathcal{O}^*(1.4423^n)$.

# MIS – Branching Analysis

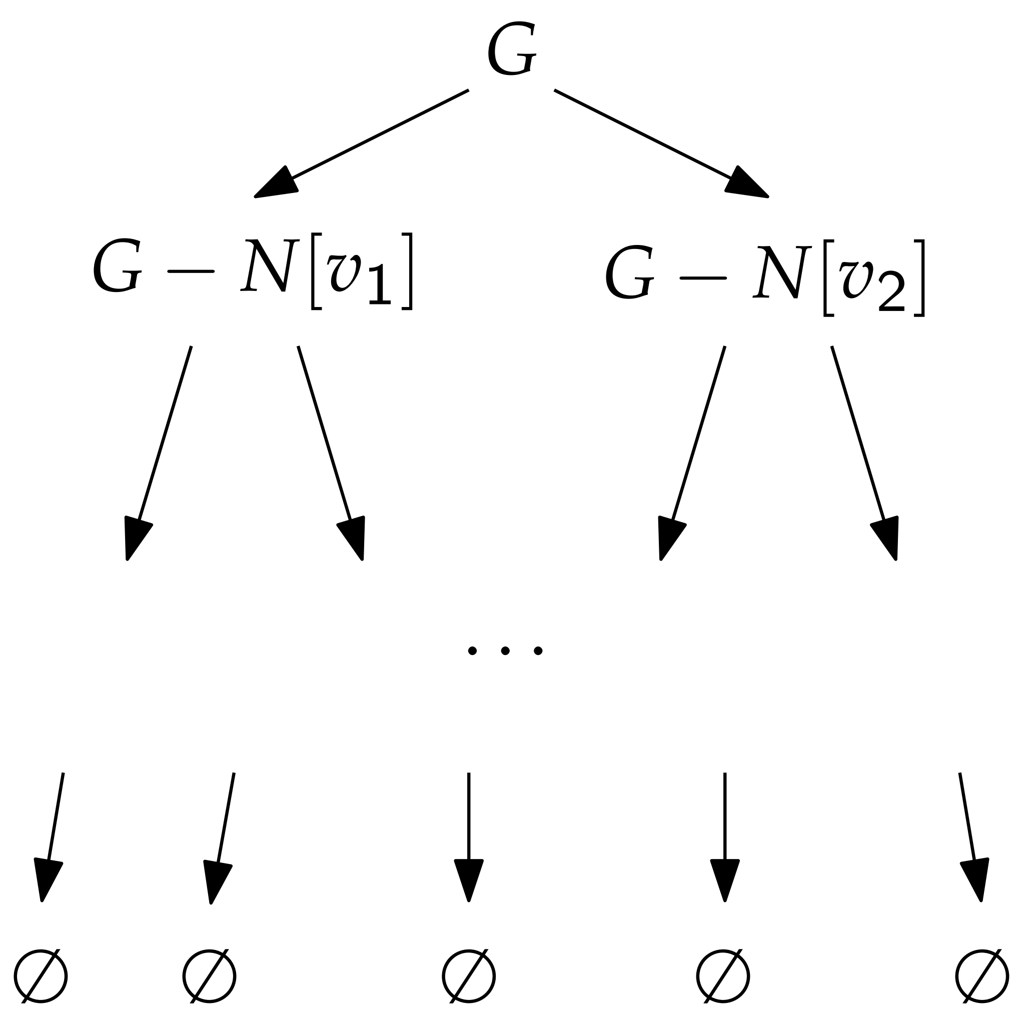
Execution corresponds to a **search tree** whose vertices are labeled with the input of the respective recursive call.

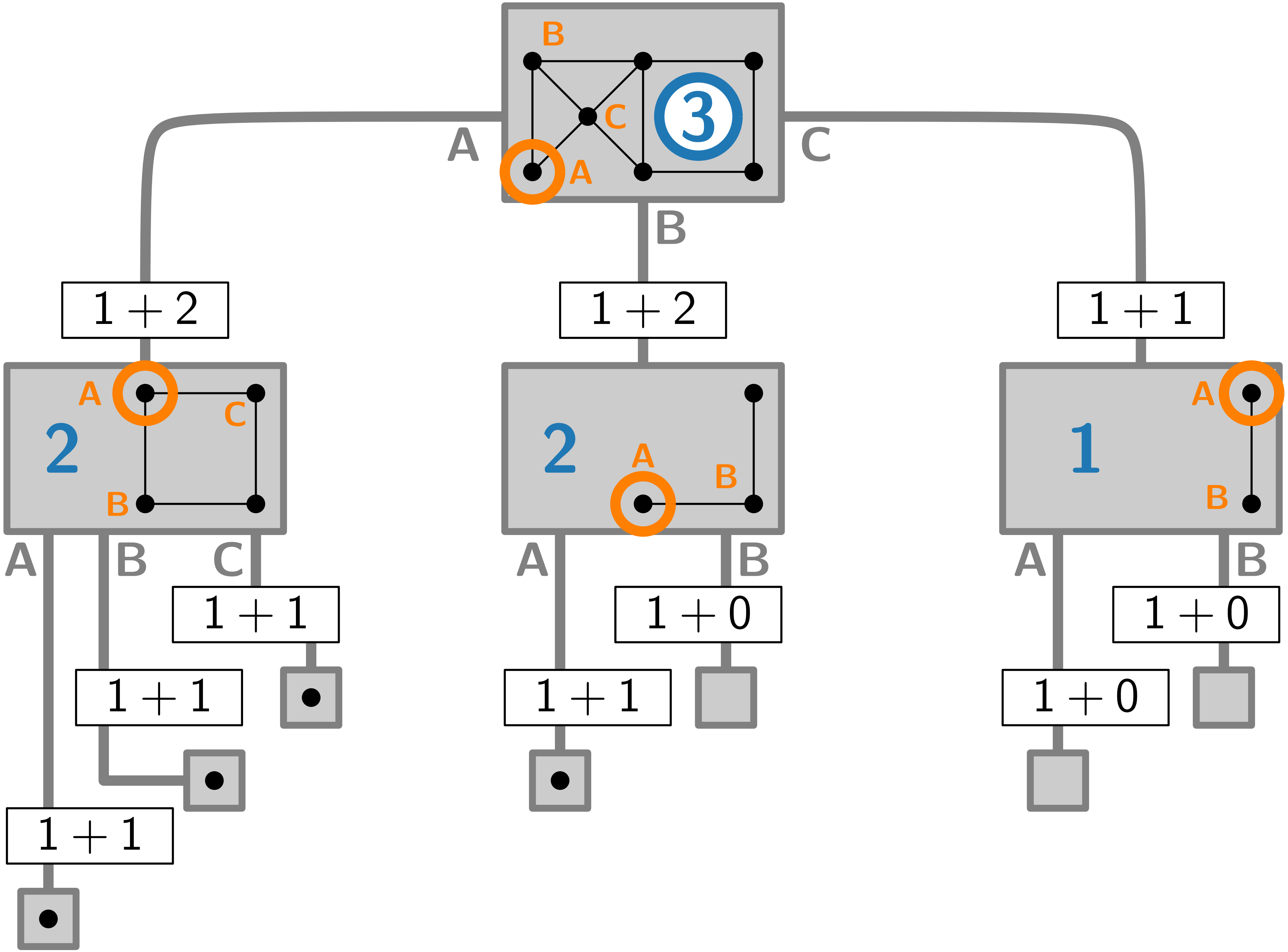
- Let $B(n)$ be the maximum number of leaves of a search tree for an $n$-vertex graph.
- Search tree has height $\leq n$.

$\Rightarrow$ The runtime of the algorithm is

$$T(n) \in \mathcal{O}(nB(n)) = \mathcal{O}^*(B(n)).$$

- Let's consider an example run.




B
C
A
3
A
C
B
1 + 2
1 + 2
1 + 1
A
C
2
B
2
A
B
A
1
B
A
B
C
A
B
A
B
1 + 1
1 + 0
1 + 0
1 + 1
1 + 1
1 + 0
1 + 1

# MIS – Runtime Analysis

For a worst-case $n$-vertex graph $G$ ($n \geq 1$):

$$B(n) \leq \sum_{y \in N[v]} B(n - (\deg(y) + 1)) \leq (\deg(v) + 1) \cdot B(\, n - (\deg(v) + 1) \,)$$

where $v$ is a minimum-degree vertex of $G$, and $B(n') \leq B(n)$ for any $n' \leq n$.

We prove by induction that $B(n) \leq 3^{n/3}$.

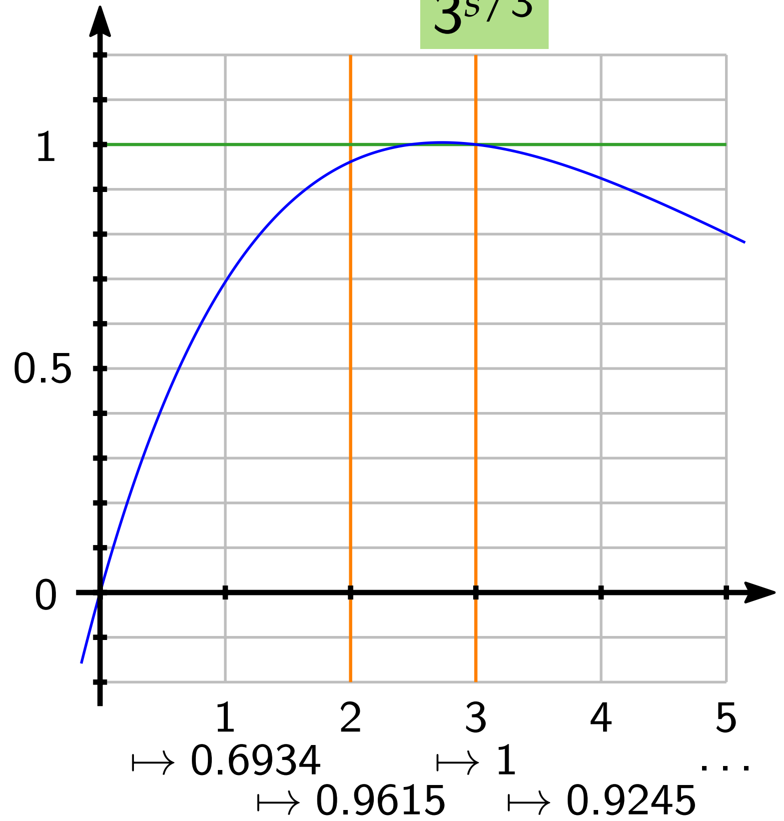
- Base case: $B(0) = 1 \leq 3^{0/3} = 1$
- Induc. hypothesis: for every $n' \leq n$, $B(n') \leq 3^{n'/3}$ holds.
- Induc. step: for $n \geq 1$, set $s = \deg(v) + 1$.

$$B(n) \leq s \cdot B(n-s) \leq s \cdot 3^{(n-s)/3} = \frac{s}{3^{s/3}} \cdot 3^{n/3} \leq 3^{n/3}$$

$\leq 1$ for all natural numbers

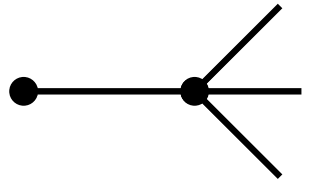
$$B(n) \in \mathcal{O}^*(\sqrt[3]{3}^{\,n}) \subseteq \mathcal{O}^*(1.44225^n)$$

$s \mapsto \frac{s}{3^{s/3}}$



# MIS – Discussion

- Smarter branching leads to an $\mathcal{O}^*(1.44225^n)$-time algorithm.
- In comparison, brute-force runs in $\mathcal{O}^*(2^n)$ time.
- Algorithms for MIS known that run in $\mathcal{O}^*(1.2202^n)$ time and polynomial space,
- and in $\mathcal{O}^*(1.2109^n)$ time and exponential space.
- What vertices are always in a MIS?
- What vertices can we savely assume are in a MIS?



- Advanced case analysis in [Fomin & Kratsch, Ch 2.3] leads to an $\mathcal{O}^*(1.2786^n)$-time algorithm.
- **Exercise**: Edge-branching for MIS

# Literature

Main source:

- [Fomin & Kratsch, Chapter 1] Exact Exponential Algorithms

Referenced papers:

- [ADMV '15] Classic Nintendo Games are (Computationally) Hard
- [Mann '17] The Top Eight Misconceptions about NP-Hardness