Advanced algorithms and data structures

Lecture 9: Exact exponential algorithms and parameterized complexity

Jacob Holm (jaho@di.ku.dk)

December 19th 2022

Today's Lecture

Introduction

We usually want algorithms that

1) in polynomial time,
2) for all instances,
3) find an exact solution.

Unfortunately some problems are hard, and we may have to settle for (at best) 2 out of 3. We call such algorithms

Exact exponential algorithms
if we relax 1) to allow using exponential time.

Parameterized algorithms
if we relax 2) to instances with small fixed values of some parameter.

Approximation algorithms
if we relax 3) to allow approximate solutions (next 2 lectures).

Introduction

We usually want algorithms that

1) in polynomial time,
2) for all instances,
3) find an exact solution.

Unfortunately some problems are hard, and we may have to settle for (at best) 2 out of 3. We call such algorithms

Exact exponential algorithms
  if we relax 1) to allow using exponential time.

Parameterized algorithms
  if we relax 2) to instances with small fixed values of some parameter.

Approximation algorithms
  if we relax 3) to allow approximate solutions (next 2 lectures).

Introduction

We usually want algorithms that

1) in polynomial time,
2) for all instances,
3) find an exact solution.

Unfortunately some problems are hard, and we may have to settle for (at best) 2 out of 3. We call such algorithms

Exact exponential algorithms
: if we relax 1) to allow using exponential time. 当我们就想要正确答案时.

Parameterized algorithms
: if we relax 2) to instances with small fixed values of some parameter.

Approximation algorithms
: if we relax 3) to allow approximate solutions (next 2 lectures).

Introduction

We usually want algorithms that

1) in polynomial time,
2) for all instances,
3) find an exact solution.

Unfortunately some problems are hard, and we may have to settle for (at best) 2 out of 3. We call such algorithms

Exact exponential algorithms
: if we relax 1) to allow using exponential time.

Parameterized algorithms
: if we relax 2) to instances with small fixed values of some parameter.

Approximation algorithms
: if we relax 3) to allow approximate solutions (next 2 lectures).

Introduction

We usually want algorithms that

1) in polynomial time,
2) for all instances,
3) find an exact solution.

Unfortunately some problems are hard, and we may have to settle for (at best) 2 out of 3. We call such algorithms

Exact exponential algorithms
: if we relax 1) to allow using exponential time.

Parameterized algorithms
: if we relax 2) to instances with small fixed values of some parameter.

Approximation algorithms
: if we relax 3) to allow approximate solutions (next 2 lectures).

↓ 想要一个大概的 solution

AADS Lecture 9, Part 2

Exact exponential algorithms

Exact exponential algorithms

Recall that a *decision problem* is in NP if and only if there exists:

- A polynomial-time verifier $R(x, y)$; and
- a function $m(x) \in \mathcal{O}(\text{poly}|x|)$; such that
- for every problem instance x: x is a yes-instance if and only if there exists a certificate y of size $|y| \leq m(x)$ such that $R(x, y)$ is true.

Note: A certificate is a proof that a solution exists, but does not have to *be* a solution. However, a solution is often the most natural certificate.

Note: Every optimization problem has a decision version. What is it?

Every problem in NP has a simple brute-force algorithm of the following form: Given problem instance x, try all potential certificates y with $|y| \leq m(x)$ and check if $R(x, y)$ for any of them.

Since a potential certificate is just a bit string of length at most $m(x)$ there are at most $\mathcal{O}(2^{m(x)})$ potential certificates to check, and each check takes $\mathcal{O}(\text{poly}|x|)$ time. Thus, if we assume $m(x)$ can be computed in $\mathcal{O}(\text{poly}|x|)$ time, the brute force running time is $\mathcal{O}(2^{m(x)} \text{poly}|x|)$.

Exact exponential algorithms

Recall that a *decision problem* is in NP if and only if there exists:

- A polynomial-time verifier $R(x, y)$; and
- a function $m(x) \in \mathcal{O}(\text{poly}|x|)$; such that
- for every problem instance x: x is a yes-instance if and only if there exists a certificate y of size $|y| \leq m(x)$ such that $R(x, y)$ is true.

Note: A certificate is a proof that a solution exists, but does not have to *be* a solution. However, a solution is often the most natural certificate.

Note: Every optimization problem has a decision version. What is it?

Every problem in NP has a simple brute-force algorithm of the following form: Given problem instance x, try all potential certificates y with $|y| \leq m(x)$ and check if $R(x, y)$ for any of them.

Since a potential certificate is just a bit string of length at most $m(x)$ there are at most $\mathcal{O}(2^{m(x)})$ potential certificates to check, and each check takes $\mathcal{O}(\text{poly}|x|)$ time. Thus, if we assume $m(x)$ can be computed in $\mathcal{O}(\text{poly}|x|)$ time, the brute force running time is $\mathcal{O}(2^{m(x)} \text{poly}|x|)$.

Exact exponential algorithms

Recall that a *decision problem* is in NP if and only if there exists:

- A polynomial-time verifier $R(x, y)$; and
- a function $m(x) \in \mathcal{O}(\text{poly}|x|)$; such that
- for every problem instance x: x is a yes-instance if and only if there exists a certificate y of size $|y| \leq m(x)$ such that $R(x, y)$ is true.

Note: A certificate is a proof that a solution exists, but does not have to *be* a solution. However, a solution is often the most natural certificate.

Note: Every optimization problem has a decision version. What is it?
Instead of asking for the "best" value z with some property, ask whether a value z given as part of the input has that property.

Every problem in NP has a simple brute-force algorithm of the following form: Given problem instance x, try all potential certificates y with $|y| \leq m(x)$ and check if $R(x, y)$ for any of them.

Since a potential certificate is just a bit string of length at most $m(x)$ there are at most $\mathcal{O}(2^{m(x)})$ potential certificates to check, and each check takes $\mathcal{O}(\text{poly}|x|)$ time. Thus, if we assume $m(x)$ can be computed in $\mathcal{O}(\text{poly}|x|)$ time, the brute force running time is $\mathcal{O}(2^{m(x)} \text{poly}|x|)$.

Exact exponential algorithms

Recall that a *decision problem* is in NP if and only if there exists:

- A polynomial-time verifier $R(x, y)$; and
- a function $m(x) \in \mathcal{O}(\mathrm{poly}|x|)$; such that
- for every problem instance x: x is a yes-instance if and only if there exists a certificate y of size $|y| \leq m(x)$ such that $R(x, y)$ is true.

Note: A certificate is a proof that a solution exists, but does not have to *be* a solution. However, a solution is often the most natural certificate.

Note: Every optimization problem has a decision version. What is it?
Instead of asking for the "best" value z with some property, ask whether a value z given as part of the input has that property.

Every problem in NP has a simple brute-force algorithm of the following form: Given problem instance x, try all potential certificates y with $|y| \leq m(x)$ and check if $R(x, y)$ for any of them.

Since a potential certificate is just a bit string of length at most $m(x)$ there are at most $\mathcal{O}(2^{m(x)})$ potential certificates to check, and each check takes $\mathcal{O}(\mathrm{poly}|x|)$ time. Thus, if we assume $m(x)$ can be computed in $\mathcal{O}(\mathrm{poly}|x|)$ time, the brute force running time is $\mathcal{O}(2^{m(x)}\,\mathrm{poly}|x|)$.

每个NP都可以通过brute-force算法。给定问题实例x，尝试所有输入为poly大小的y并且有R(x,y)

Exact exponential algorithms

Recall that a *decision problem* is in NP if and only if there exists:

- A polynomial-time verifier $R(x, y)$; and
- a function $m(x) \in \mathcal{O}(\text{poly}|x|)$; such that
- for every problem instance x: x is a yes-instance if and only if there exists a certificate y of size $|y| \leq m(x)$ such that $R(x, y)$ is true.

Note: A certificate is a proof that a solution exists, but does not have to *be* a solution. However, a solution is often the most natural certificate.

Note: Every optimization problem has a decision version. What is it? Instead of asking for the "best" value z with some property, ask whether a value z given as part of the input has that property.

Every problem in NP has a simple brute-force algorithm of the following form: Given problem instance x, try all potential certificates y with $|y| \leq m(x)$ and check if $R(x, y)$ for any of them.

每一位都有2种可能

Since a potential certificate is just a bit string of length at most $m(x)$ there are at most $\mathcal{O}(2^{m(x)})$ potential certificates to check, and each check takes $\mathcal{O}(\text{poly}|x|)$ time. Thus, if we assume $m(x)$ can be computed in $\mathcal{O}(\text{poly}|x|)$ time, the brute force running time is $\mathcal{O}(2^{m(x)} \text{poly}|x|)$.

Notation: $\mathcal{O}^\star(\cdot)$

For any fixed $b > a \geq 1$, and $c \in \mathbb{R}$, we have $\mathcal{O}(a^n \cdot n^c) \subset \mathcal{O}(b^n)$.

So when comparing exact exponential algorithms, the polynomial factors are mostly irrelevant.

Define

$$f(n) \in \mathcal{O}^\star(g(n)) \iff \exists c \in \mathbb{R} : f(n) \in \mathcal{O}(n^c \cdot g(n))$$

In other words, $\mathcal{O}^\star(\cdot)$ is the same as $\mathcal{O}(\cdot)$ but ignores polynomial factors.

Notice that for all $b > a \geq 1$: $\mathcal{O}(a^n) \subset \mathcal{O}^\star(a^n) \subset \mathcal{O}(b^n)$.

Using this notation, what is the running time for the simple brute-force algorithm?

Notation: $\mathcal{O}^\star(\cdot)$

grows faster ,

For any fixed $b > a \geq 1$, and $c \in \mathbb{R}$, we have $\mathcal{O}(a^n \cdot n^c) \subset \mathcal{O}(b^n)$.

只要底数稍大一点，它都比左边大。

So when comparing exact exponential algorithms, the polynomial factors are mostly irrelevant.

在比较精准指数时，多项式因素大多无关。

Define

$$f(n) \in \mathcal{O}^\star(g(n)) \iff \exists c \in \mathbb{R} : f(n) \in \mathcal{O}(n^c \cdot g(n))$$

In other words, $\mathcal{O}^\star(\cdot)$ is the same as $\mathcal{O}(\cdot)$ but ignores polynomial factors.

Notice that for all $b > a \geq 1$: $\mathcal{O}(a^n) \subset \mathcal{O}^\star(a^n) \subset \mathcal{O}(b^n)$.

Using this notation, what is the running time for the simple brute-force algorithm?

Notation: $\mathcal{O}^\star(\cdot)$

For any fixed $b > a \geq 1$, and $c \in \mathbb{R}$, we have $\mathcal{O}(a^n \cdot n^c) \subset \mathcal{O}(b^n)$.

So when comparing exact exponential algorithms, the polynomial factors are mostly irrelevant.

Define

$$f(n) \in \mathcal{O}^\star(g(n)) \iff \exists c \in \mathbb{R} : f(n) \in \mathcal{O}(n^c \cdot g(n))$$

In other words, $\mathcal{O}^\star(\cdot)$ is the same as $\mathcal{O}(\cdot)$ but ignores polynomial factors.

Notice that for all $b > a \geq 1$: $\mathcal{O}(a^n) \subset \mathcal{O}^\star(a^n) \subset \mathcal{O}(b^n)$.

Using this notation, what is the running time for the simple brute-force algorithm?

Notation: $\mathcal{O}^\star(\cdot)$

For any fixed $b > a \geq 1$, and $c \in \mathbb{R}$, we have $\mathcal{O}(a^n \cdot n^c) \subset \mathcal{O}(b^n)$.

So when comparing exact exponential algorithms, the polynomial factors are mostly irrelevant.

Define

$$f(n) \in \mathcal{O}^\star(g(n)) \iff \exists c \in \mathbb{R} : f(n) \in \mathcal{O}(n^c \cdot g(n))$$

In other words, $\mathcal{O}^\star(\cdot)$ is the same as $\mathcal{O}(\cdot)$ but ignores polynomial factors.

Notice that for all $b > a \geq 1$: $\mathcal{O}(a^n) \subset \mathcal{O}^\star(a^n) \subset \mathcal{O}(b^n)$.

Using this notation, what is the running time for the simple brute-force algorithm?

Notation: $\mathcal{O}^\star(\cdot)$

For any fixed $b > a \geq 1$, and $c \in \mathbb{R}$, we have $\mathcal{O}(a^n \cdot n^c) \subset \mathcal{O}(b^n)$.

So when comparing exact exponential algorithms, the polynomial factors are mostly irrelevant.

Define

$$f(n) \in \mathcal{O}^\star(g(n)) \iff \exists c \in \mathbb{R} : f(n) \in \mathcal{O}(n^c \cdot g(n))$$

In other words, $\mathcal{O}^\star(\cdot)$ is the same as $\mathcal{O}(\cdot)$ but ignores polynomial factors.

Notice that for all $b > a \geq 1$: $\mathcal{O}(a^n) \subset \mathcal{O}^\star(a^n) \subset \mathcal{O}(b^n)$.

Using this notation, what is the running time for the simple brute-force algorithm?

$O(2^{m(x)} \mathrm{poly}|x|) \Rightarrow O^*(2^{m(x)})$

Notation: $\mathcal{O}^\star(\cdot)$

For any fixed $b > a \geq 1$, and $c \in \mathbb{R}$, we have $\mathcal{O}(a^n \cdot n^c) \subset \mathcal{O}(b^n)$.

So when comparing exact exponential algorithms, the polynomial factors are mostly irrelevant.

Define

$$f(n) \in \mathcal{O}^\star(g(n)) \iff \exists c \in \mathbb{R} : f(n) \in \mathcal{O}(n^c \cdot g(n))$$

In other words, $\mathcal{O}^\star(\cdot)$ is the same as $\mathcal{O}(\cdot)$ but ignores polynomial factors.

Notice that for all $b > a \geq 1$: $\mathcal{O}(a^n) \subset \mathcal{O}^\star(a^n) \subset \mathcal{O}(b^n)$.

Using this notation, what is the running time for the simple brute-force algorithm? $\mathcal{O}^\star(2^{m(x)})$

polynomial factor is ~~the~~ small to matter

Size of a problem

What do we mean by the "size" n of a problem? Typically:

n, or $m+n$	for graphs with n vertices and m edges.
$\|S\|$	for problems involving some set S.
#variables	for SAT-type problems.

This measure of "size" is usually sufficient to describe the running time of the natural brute-force algorithm and to show improvements in better algorithms.

Problem	**certificate size**	**brute-force time**
SAT, MIS	$m(x) = n$	$T(n) \in \mathcal{O}^\star(2^n)$
TSP	$m(x) = \log_2(n!)$	$T(n) \in \mathcal{O}^\star(n!)$
k-Vertex Cover	$m(x) = k \log_2(n)$	$T(n) \in \mathcal{O}(n^k \cdot \mathrm{poly}\|x\|)$
Vertex k-coloring	$m(x) = \log_2(k^n)$	$T(n) \in \mathcal{O}^\star(k^n)$

Size of a problem

What do we mean by the "size" n of a problem? Typically:

n, or $m+n$	for graphs with n vertices and m edges.
$\lvert S \rvert$	for problems involving some set S.
#variables	for SAT-type problems.

This measure of "size" is usually sufficient to describe the running time of the natural brute-force algorithm and to show improvements in better algorithms.

Problem	**certificate size**	**brute-force time**
SAT, MIS	$m(x) = n$	$T(n) \in \mathcal{O}^\star(2^n)$
TSP	$m(x) = \log_2(n!)$	$T(n) \in \mathcal{O}^\star(n!)$
k-Vertex Cover	$m(x) = k \log_2(n)$	$T(n) \in \mathcal{O}(n^k \cdot \text{poly}\lvert x \rvert)$
Vertex k-coloring	$m(x) = \log_2(k^n)$	$T(n) \in \mathcal{O}^\star(k^n)$

Size of a problem

What do we mean by the "size" n of a problem? Typically:

n, or $m+n$	for graphs with n vertices and m edges.
$\lvert S\rvert$	for problems involving some set S.
#variables	for SAT-type problems.

This measure of "size" is usually sufficient to describe the running time of the natural brute-force algorithm and to show improvements in better algorithms.

Problem	**certificate size**	**brute-force time**
SAT, MIS	$m(x) = n$	$T(n) \in \mathcal{O}^\star(2^n)$
TSP	$m(x) = \log_2(n!)$	$T(n) \in \mathcal{O}^\star(n!)$
k-Vertex Cover	$m(x) = k \log_2(n)$	$T(n) \in \mathcal{O}(n^k \cdot \text{poly}\lvert x\rvert)$
Vertex k-coloring	$m(x) = \log_2(k^n)$	$T(n) \in \mathcal{O}^\star(k^n)$

Size of a problem

What do we mean by the "size" n of a problem? Typically:

n, or $m+n$	for graphs with n vertices and m edges.
$\lvert S\rvert$	for problems involving some set S.
#variables	for SAT-type problems.

This measure of "size" is usually sufficient to describe the running time of the natural brute-force algorithm and to show improvements in better algorithms.

Problem	**certificate size**	**brute-force time**
SAT, MIS	$m(x) = n$	$T(n) \in \mathcal{O}^\star(2^n)$
TSP	$m(x) = \log_2(n!)$	$T(n) \in \mathcal{O}^\star(n!)$
k-Vertex Cover	$m(x) = k \log_2(n)$	$T(n) \in \mathcal{O}(n^k \cdot \mathrm{poly}\lvert x\rvert)$
Vertex k-coloring	$m(x) = \log_2(k^n)$	$T(n) \in \mathcal{O}^\star(k^n)$

Size of a problem

What do we mean by the "size" n of a problem? Typically:

n, or $m+n$	for graphs with n vertices and m edges.
$\lvert S \rvert$	for problems involving some set S.
#variables	for SAT-type problems.

This measure of "size" is usually sufficient to describe the running time of the natural brute-force algorithm and to show improvements in better algorithms.

Problem	**certificate size**	**brute-force time**
SAT, MIS	$m(x) = n$	$T(n) \in \mathcal{O}^\star(2^n)$
TSP	$m(x) = \log_2(n!)$	$T(n) \in \mathcal{O}^\star(n!)$
k-Vertex Cover	$m(x) = k \log_2(n)$	$T(n) \in \mathcal{O}(n^k \cdot \text{poly}\lvert x\rvert)$
Vertex k-coloring	$m(x) = \log_2(k^n)$	$T(n) \in \mathcal{O}^\star(k^n)$

Size of a problem

What do we mean by the "size" n of a problem? Typically:

n, or $m + n$	for graphs with n vertices and m edges.
$\lvert S \rvert$	for problems involving some set S.
#variables	for SAT-type problems.

This measure of "size" is usually sufficient to describe the running time of the natural brute-force algorithm and to show improvements in better algorithms.

maximum independent set.

每1位都有 true / false

Problem	**certificate size**	**brute-force time**
SAT, MIS	$m(x) = n$	$T(n) \in \mathcal{O}^\star(2^n)$
TSP	$m(x) = \log_2(n!)$	$T(n) \in \mathcal{O}^\star(n!)$
k-Vertex Cover	$m(x) = k \log_2(n)$	$T(n) \in \mathcal{O}(n^k \cdot \mathrm{poly}\lvert x \rvert)$
Vertex k-coloring	$m(x) = \log_2(k^n)$	$T(n) \in \mathcal{O}^\star(k^n)$

Size of a problem

What do we mean by the "size" n of a problem? Typically:

n, or $m+n$	for graphs with n vertices and m edges.
$\lvert S\rvert$	for problems involving some set S.
#variables	for SAT-type problems.

This measure of "size" is usually sufficient to describe the running time of the natural brute-force algorithm and to show improvements in better algorithms.

Problem	**certificate size**	**brute-force time**
SAT, MIS	$m(x) = n$	$T(n) \in \mathcal{O}^\star(2^n)$
TSP	$m(x) = \log_2(n!)$	$T(n) \in \mathcal{O}^\star(n!)$
k-Vertex Cover	$m(x) = k\log_2(n)$	$T(n) \in \mathcal{O}(n^k \cdot \mathrm{poly}\lvert x\rvert)$
Vertex k-coloring	$m(x) = \log_2(k^n)$	$T(n) \in \mathcal{O}^\star(k^n)$

Size of a problem

What do we mean by the "size" n of a problem? Typically:

n, or $m+n$	for graphs with n vertices and m edges.
$\|S\|$	for problems involving some set S.
#variables	for SAT-type problems.

This measure of "size" is usually sufficient to describe the running time of the natural brute-force algorithm and to show improvements in better algorithms.

Problem	**certificate size**	**brute-force time**
SAT, MIS	$m(x) = n$	$T(n) \in \mathcal{O}^\star(2^n)$
TSP	$m(x) = \log_2(n!)$	$T(n) \in \mathcal{O}^\star(n!)$
k-Vertex Cover	$m(x) = k \log_2(n)$	$T(n) \in \mathcal{O}(n^k \cdot \text{poly}\|x\|)$
Vertex k-coloring	$m(x) = \log_2(k^n)$	$T(n) \in \mathcal{O}^\star(k^n)$

Size of a problem

What do we mean by the "size" n of a problem? Typically:

n, or $m+n$	for graphs with n vertices and m edges.
$\|S\|$	for problems involving some set S.
#variables	for SAT-type problems.

This measure of "size" is usually sufficient to describe the running time of the natural brute-force algorithm and to show improvements in better algorithms.

Problem	**certificate size**	**brute-force time**
SAT, MIS	$m(x) = n$	$T(n) \in \mathcal{O}^\star(2^n)$
TSP	$m(x) = \log_2(n!)$	$T(n) \in \mathcal{O}^\star(n!)$
k-Vertex Cover	$m(x) = k \log_2(n)$	$T(n) \in \mathcal{O}(n^k \cdot \mathrm{poly}\|x\|)$
Vertex k-coloring	$m(x) = \log_2(k^n)$	$T(n) \in \mathcal{O}^\star(k^n)$

Size of a problem

What do we mean by the "size" n of a problem? Typically:

n, or $m+n$	for graphs with n vertices and m edges.
$\|S\|$	for problems involving some set S.
#variables	for SAT-type problems.

This measure of "size" is usually sufficient to describe the running time of the natural brute-force algorithm and to show improvements in better algorithms.

Problem	**certificate size**	**brute-force time**	**this lecture**
SAT, MIS	$m(x) = n$	$T(n) \in \mathcal{O}^\star(2^n)$ $\longrightarrow$	$\mathcal{O}^\star(3^{n/3})$
TSP	$m(x) = \log_2(n!)$	$T(n) \in \mathcal{O}^\star(n!)$ $\longrightarrow$	$\mathcal{O}^\star(2^n)$
k-Vertex Cover	$m(x) = k \log_2(n)$	$T(n) \in \mathcal{O}(n^k \cdot \text{poly}\|x\|)$	$\mathcal{O}_k(m+n)$
Vertex k-coloring	$m(x) = \log_2(k^n)$	$T(n) \in \mathcal{O}^\star(k^n)$	?

TSP via Dynamic Programming (Bellman-Held-Karp)

Problem: Given cities $c_1, \ldots, c_n$, and distances $d_{ij} = d(c_i, c_j)$, find tour of minimal length, visiting all cities exactly once. Equivalently, find permutation π minimizing $d(c_{\pi(n)}, c_{\pi(1)}) + \sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)})$.

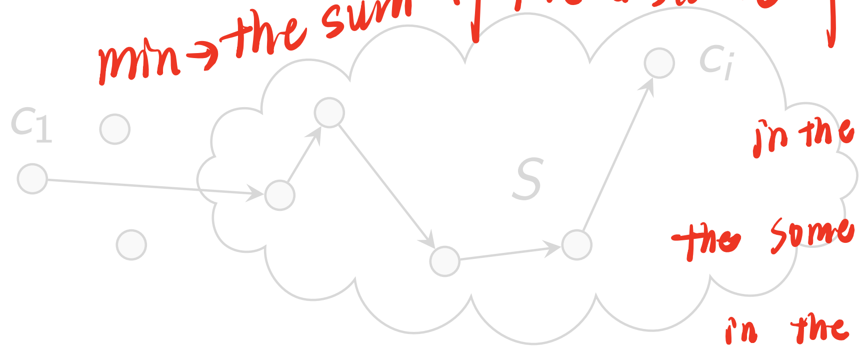
排列组合

使这个排列中从最后一个到第一个的距离加上序列中从每2个到下一个的距离之和最小化

Idea: For all $S \subseteq \{c_2, \ldots, c_n\}$ and $c_i \in S$ define $\mathrm{OPT}[S, c_i] :=$ minimum length of all paths in $S \cup \{c_1\}$ that starts in c_1, visits all of S once, and ends in c_i. Then $\min\{\mathrm{OPT}[\{c_2, \ldots, c_n\}, c_i] + d(c_i, c_1) \mid c_i \in \{c_2, \ldots, c_n\}\}$ is the length of the minimal tour.

min → the sum of the distance from the last to the first in the permutation plus the some from each two to the next in the sequence

$\mathrm{OPT}[S, c_i]$:



Lemma

$$\mathrm{OPT}[S, c_i] = \begin{cases} d(c_1, c_i) & \text{if } S = \{c_i\} \\ \min\left\{\mathrm{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i) \;\middle|\; c_k \in S \setminus \{c_i\}\right\} & \text{if } \{c_i\} \subset S \end{cases}$$

Proof.

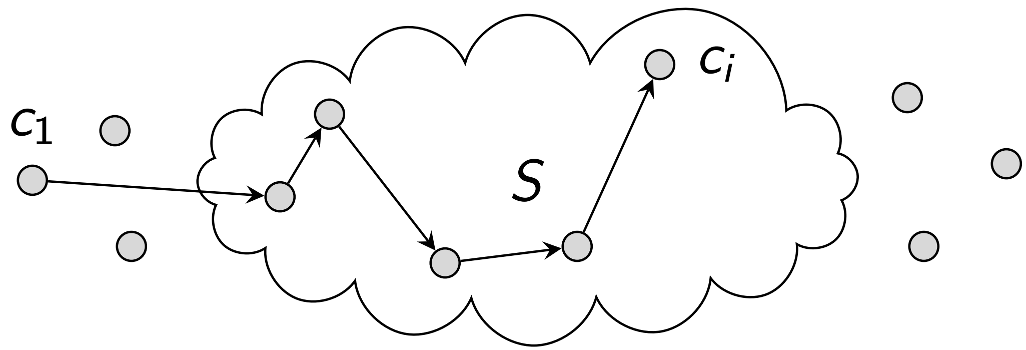
Let $e = (c_k, c_i)$ be the last edge on such a path. If $k = 1$ we are done. If $k \neq 1$ the shortest length through e must be $\mathrm{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i)$. The shortest such path must use the minimum over all $c_k \in S \setminus \{c_i\}$. □

TSP via Dynamic Programming (Bellman-Held-Karp)

Problem: Given cities $c_1, \ldots, c_n$, and distances $d_{ij} = d(c_i, c_j)$, find tour of minimal length, visiting all cities exactly once. Equivalently, find permutation π minimizing $d(c_{\pi(n)}, c_{\pi(1)}) + \sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)})$.

Idea: For all $S \subseteq \{c_2, \ldots, c_n\}$ and $c_i \in S$ define $\mathrm{OPT}[S, c_i] :=$ minimum length of all paths in $S \cup \{c_1\}$ that starts in c_1, visits all of S once, and ends in c_i. Then $\min\{\mathrm{OPT}[\{c_2, \ldots c_n\}, c_i] + d(c_i, c_1) \mid c_i \in \{c_2, \ldots, c_n\}\}$ is the length of the minimal tour.

$\mathrm{OPT}[S, c_i]$:



Lemma

$$\mathrm{OPT}[S, c_i] = \begin{cases} d(c_1, c_i) & \text{if } S = \{c_i\} \\ \min\Big\{\mathrm{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i) \;\Big|\; c_k \in S \setminus \{c_i\}\Big\} & \text{if } \{c_i\} \subset S \end{cases}$$

Proof.

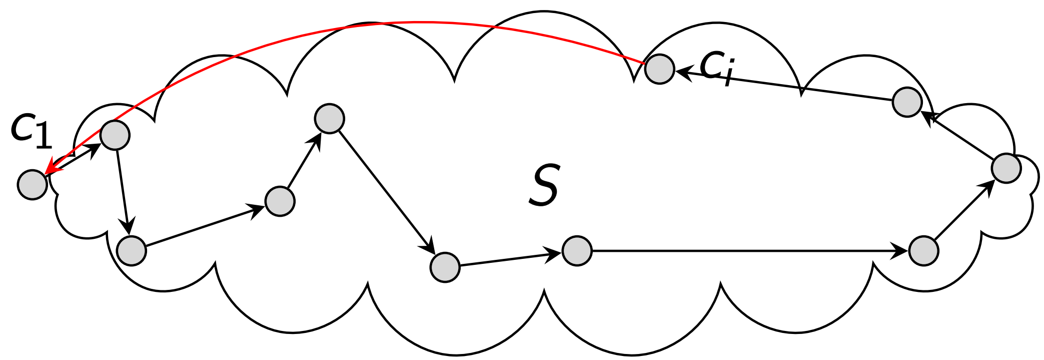
Let $e = (c_k, c_i)$ be the last edge on such a path. If $k = 1$ we are done. If $k \neq 1$ the shortest length through e must be $\mathrm{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i)$. The shortest such path must use the minimum over all $c_k \in S \setminus \{c_i\}$. $\square$

TSP via Dynamic Programming (Bellman-Held-Karp)

Problem: Given cities $c_1, \dots, c_n$, and distances $d_{ij} = d(c_i, c_j)$, find tour of minimal length, visiting all cities exactly once. Equivalently, find permutation π minimizing $d(c_{\pi(n)}, c_{\pi(1)}) + \sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)})$.

Idea: For all $S \subseteq \{c_2, \dots, c_n\}$ and $c_i \in S$ define $\mathrm{OPT}[S, c_i] :=$ minimum length of all paths in $S \cup \{c_1\}$ that starts in c_1, visits all of S once, and ends in c_i. Then $\min\Big\{\mathrm{OPT}[\{c_2, \dots c_n\}, c_i] + d(c_i, c_1) \;\Big|\; c_i \in \{c_2, \dots, c_n\}\Big\}$ is the length of the minimal tour.

$\mathrm{OPT}[S, c_i]$:



Lemma

$$\mathrm{OPT}[S, c_i] = \begin{cases} d(c_1, c_i) & \text{if } S = \{c_i\} \\ \min\Big\{\mathrm{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i) \;\Big|\; c_k \in S \setminus \{c_i\}\Big\} & \text{if } \{c_i\} \subset S \end{cases}$$

Proof.

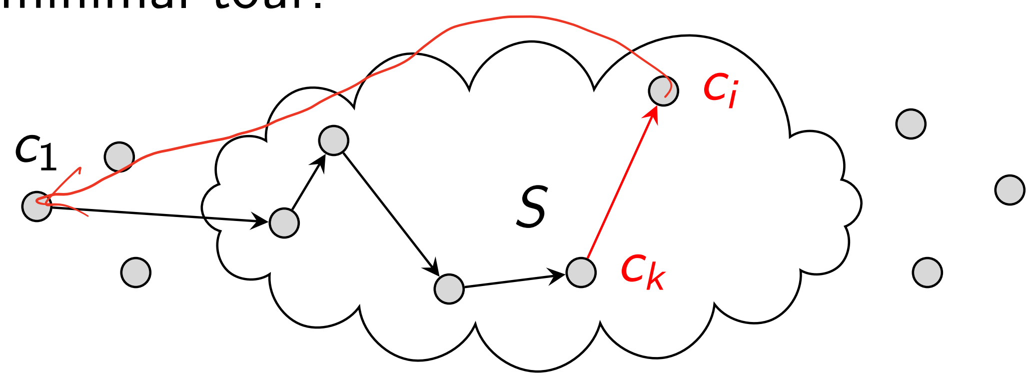
Let $e = (c_k, c_i)$ be the last edge on such a path. If $k = 1$ we are done. If $k \neq 1$ the shortest length through e must be $\mathrm{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i)$. The shortest such path must use the minimum over all $c_k \in S \setminus \{c_i\}$. $\square$

TSP via Dynamic Programming (Bellman-Held-Karp)

Problem: Given cities $c_1, \dots, c_n$, and distances $d_{ij} = d(c_i, c_j)$, find tour of minimal length, visiting all cities exactly once. Equivalently, find permutation π minimizing $d(c_{\pi(n)}, c_{\pi(1)}) + \sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)})$.

Idea: For all $S \subseteq \{c_2, \dots, c_n\}$ and $c_i \in S$ define $\mathrm{OPT}[S, c_i] :=$ minimum length of all paths in $S \cup \{c_1\}$ that starts in c_1, visits all of S once, and ends in c_i. Then $\min\left\{\mathrm{OPT}[\{c_2, \dots c_n\}, c_i] + d(c_i, c_1) \;\middle|\; c_i \in \{c_2, \dots, c_n\}\right\}$ is the length of the minimal tour.

$\mathrm{OPT}[S, c_i]$:



Lemma

$$\mathrm{OPT}[S, c_i] = \begin{cases} d(c_1, c_i) & \text{if } S = \{c_i\} \\ \min\left\{\mathrm{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i) \;\middle|\; c_k \in S \setminus \{c_i\}\right\} & \text{if } \{c_i\} \subset S \end{cases}$$

Proof.

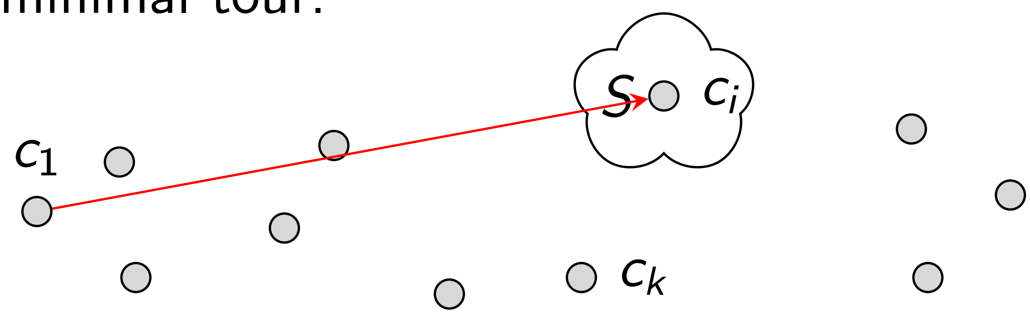
Let $e = (c_k, c_i)$ be the last edge on such a path. If $k = 1$ we are done. If $k \neq 1$ the shortest length through e must be $\mathrm{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i)$. The shortest such path must use the minimum over all $c_k \in S \setminus \{c_i\}$. □

TSP via Dynamic Programming (Bellman-Held-Karp)

Problem: Given cities $c_1, \dots, c_n$, and distances $d_{ij} = d(c_i, c_j)$, find tour of minimal length, visiting all cities exactly once. Equivalently, find permutation π minimizing $d(c_{\pi(n)}, c_{\pi(1)}) + \sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)})$.

Idea: For all $S \subseteq \{c_2, \dots, c_n\}$ and $c_i \in S$ define $\mathrm{OPT}[S, c_i] :=$ minimum length of all paths in $S \cup \{c_1\}$ that starts in c_1, visits all of S once, and ends in c_i. Then $\min\left\{\mathrm{OPT}[\{c_2, \dots c_n\}, c_i] + d(c_i, c_1) \;\middle|\; c_i \in \{c_2, \dots, c_n\}\right\}$ is the length of the minimal tour.

$\mathrm{OPT}[S, c_i]$:



Lemma

$$\mathrm{OPT}[S, c_i] = \begin{cases} d(c_1, c_i) & \text{if } S = \{c_i\} \\ \min\left\{\mathrm{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i) \;\middle|\; c_k \in S \setminus \{c_i\}\right\} & \text{if } \{c_i\} \subset S \end{cases}$$

Proof.

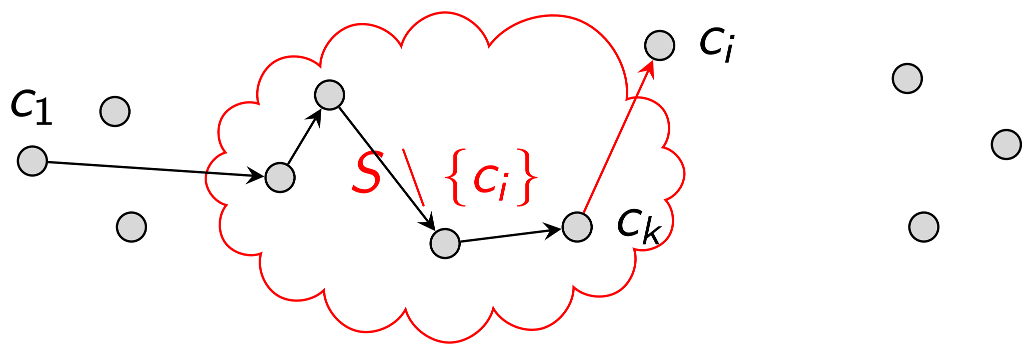
Let $e = (c_k, c_i)$ be the last edge on such a path. If $k = 1$ we are done. If $k \neq 1$ the shortest length through e must be $\mathrm{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i)$. The shortest such path must use the minimum over all $c_k \in S \setminus \{c_i\}$. $\square$

TSP via Dynamic Programming (Bellman-Held-Karp)

Problem: Given cities $c_1, \dots, c_n$, and distances $d_{ij} = d(c_i, c_j)$, find tour of minimal length, visiting all cities exactly once. Equivalently, find permutation π minimizing $d(c_{\pi(n)}, c_{\pi(1)}) + \sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)})$.

Idea: For all $S \subseteq \{c_2, \dots, c_n\}$ and $c_i \in S$ define $\mathrm{OPT}[S, c_i] :=$ minimum length of all paths in $S \cup \{c_1\}$ that starts in c_1, visits all of S once, and ends in c_i. Then $\min\left\{\mathrm{OPT}[\{c_2, \dots c_n\}, c_i] + d(c_i, c_1) \;\middle|\; c_i \in \{c_2, \dots, c_n\}\right\}$ is the length of the minimal tour.

$\mathrm{OPT}[S, c_i]$:



Lemma

$$\mathrm{OPT}[S, c_i] = \begin{cases} d(c_1, c_i) & \textit{if } S = \{c_i\} \\ \min\left\{\mathrm{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i) \;\middle|\; c_k \in S \setminus \{c_i\}\right\} & \textit{if } \{c_i\} \subset S \end{cases}$$

Proof.

Let $e = (c_k, c_i)$ be the last edge on such a path. If $k = 1$ we are done. If $k \neq 1$ the shortest length through e must be $\mathrm{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i)$.

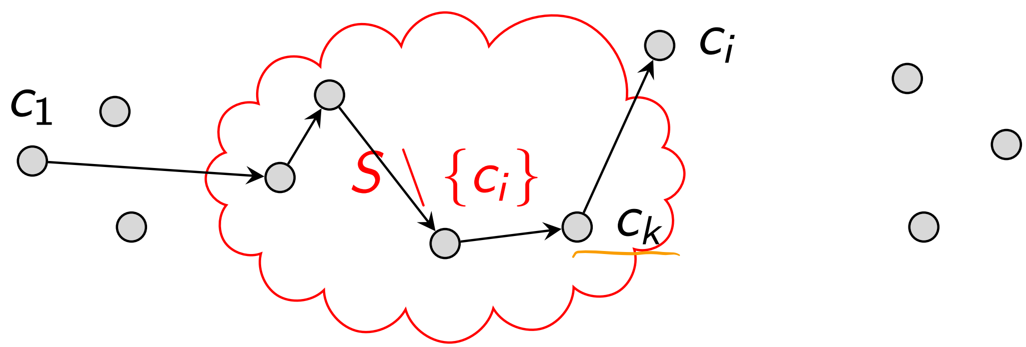
The shortest such path must use the minimum over all $c_k \in S \setminus \{c_i\}$. $\square$

TSP via Dynamic Programming (Bellman-Held-Karp)

Problem: Given cities $c_1, \ldots, c_n$, and distances $d_{ij} = d(c_i, c_j)$, find tour of minimal length, visiting all cities exactly once. Equivalently, find permutation π minimizing $d(c_{\pi(n)}, c_{\pi(1)}) + \sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)})$.

Idea: For all $S \subseteq \{c_2, \ldots, c_n\}$ and $c_i \in S$ define $\mathrm{OPT}[S, c_i] :=$ minimum length of all paths in $S \cup \{c_1\}$ that starts in c_1, visits all of S once, and ends in c_i. Then $\min\left\{\mathrm{OPT}[\{c_2, \ldots c_n\}, c_i] + d(c_i, c_1) \;\middle|\; c_i \in \{c_2, \ldots, c_n\}\right\}$ is the length of the minimal tour.

$\mathrm{OPT}[S, c_i]$:



Lemma

$$\mathrm{OPT}[S, c_i] = \begin{cases} d(c_1, c_i) & \text{if } S = \{c_i\} \\ \min\left\{\mathrm{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i) \;\middle|\; c_k \in S \setminus \{c_i\}\right\} & \text{if } \{c_i\} \subset S \end{cases}$$

Proof.

Let $e = (c_k, c_i)$ be the last edge on such a path. If $k = 1$ we are done. If $k \neq 1$ the shortest length through e must be $\mathrm{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i)$. The shortest such path must use the minimum over all $c_k \in S \setminus \{c_i\}$. $\square$

TSP via Dynamic Programming (Bellman-Held-Karp)

We can compute all OPT$[S, c_i]$ values in order of increasing size of S.

$S = \{c_i\}$

Lemma

$$\text{OPT}[S, c_i] = \begin{cases} d(c_1, c_i) & \text{if } S = \{c_i\} \text{ ①} \\ \min\{\text{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i) \mid c_k \in S \setminus \{c_i\}\} & \text{if } \{c_i\} \subset S \text{ ②} \end{cases}$$

```
function TSP({c_1, ..., c_n}, d)
    for i ← 2 ... n do
        OPT[{c_i}, c_i] ← d(c_1, c_i)                                   ①
    for j ← 2 ... n − 1 do   → 扩大S，返回c1~ci的最短路径，且每步的上一步可复用.
        for S ⊆ {c_2, ..., c_n} with |S| = j do                         ②
            for c_i ∈ S do
                OPT[S, c_i] ← min{OPT[S \ {c_i}, c_k] + d(c_k, c_i) | c_k ∈ S \ {c_i}}
    return min{OPT[{c_2, ... c_n}, c_i] + d(c_i, c_1) | c_i ∈ {c_2, ..., c_n}}
```

$2, c_3\}, c_2] + d(c_2, c_4) \mid \ldots$

加上返回的 $c_i \to c_1$

$c_1 \sim c_i$

$c_k = c_2$

$c_i = c_3$

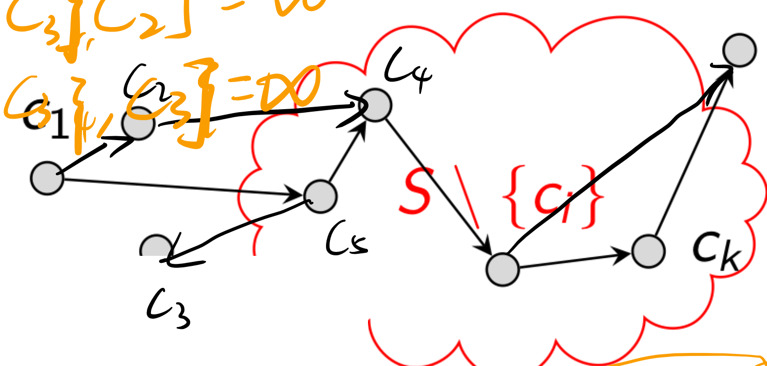
$S = \{c_2, c_3\}$

$S = \{c_3, c_4\}$

OPT $[\{c_2, c_3\}, c_2] = \infty$

OPT $[\{c_2, c_3\}, c_3] = \infty$

OPT$[S, c_i]$:



$S = \{c_4, c_5\}$

$S = \{c_2, c_3, c_4\}$

$S = \{c_3, c_4, c_5\}$

c_1 c_2 c_3 c_i

c_1 c_3 c_4

$c_2 \xrightarrow{\infty} c_4$

TSP via Dynamic Programming (Bellman-Held-Karp)

We can compute all $\mathrm{OPT}[S, c_i]$ values in order of increasing size of S.

1: **function** TSP($\{c_1, \ldots, c_n\}$,d)
2:     **for** $i \leftarrow 2 \ldots n$ **do**
3:         $\mathrm{OPT}[\{c_i\}, c_i] \leftarrow d(c_1, c_i)$
4:     **for** $j \leftarrow 2 \ldots n-1$ **do**
5:         **for** $S \subseteq \{c_2, \ldots, c_n\}$ with $|S| = j$ **do**
6:             **for** $c_i \in S$ **do**
7:                 $\mathrm{OPT}[S, c_i] \leftarrow \min\{\mathrm{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i) \mid c_k \in S \setminus \{c_i\}\}$
8:     **return** $\min\{\mathrm{OPT}[\{c_2, \ldots c_n\}, c_i] + d(c_i, c_1) \mid c_i \in \{c_2, \ldots, c_n\}\}$

Lemma

The above procedure solves TSP by computing $\mathcal{O}(n^2 \cdot 2^n)$ shortest paths.

Proof.

The number of path lengths computed in line 7 is

$$\sum_{j=2}^{n-1} \binom{n-1}{j} \sum_{i=1}^{j} (j-1) \quad \leq \quad n^2 \sum_{j=1}^{n} \binom{n}{j} \quad = \quad n^2 \cdot 2^n$$

And lines 3 and 8 compute only $n-1$ more path lengths each. $\square$

What is the running time of the algorithm?

TSP via Dynamic Programming (Bellman-Held-Karp)

We can compute all $\mathrm{OPT}[S, c_i]$ values in order of increasing size of S.

```
1: function TSP({c_1, ..., c_n}, d)
2:     for i ← 2 ... n do
3:         OPT[{c_i}, c_i] ← d(c_1, c_i)
4:     for j ← 2 ... n − 1 do
5:         for S ⊆ {c_2, ..., c_n} with |S| = j do
6:             for c_i ∈ S do
7:                 OPT[S, c_i] ← min{OPT[S \ {c_i}, c_k] + d(c_k, c_i) | c_k ∈ S \ {c_i}}
8:     return min{OPT[{c_2, ... c_n}, c_i] + d(c_i, c_1) | c_i ∈ {c_2, ..., c_n}}
```

Lemma

The above procedure solves TSP by computing $\mathcal{O}(n^2 \cdot 2^n)$ shortest paths.

Proof.

The number of path lengths computed in line 7 is

$$\sum_{j=2}^{n-1} \binom{n-1}{j} \sum_{i=1}^{j} (j-1) \quad \leq \quad n^2 \sum_{j=1}^{n} \binom{n}{j} \quad = \quad n^2 \cdot 2^n$$

And lines 3 and 8 compute only $n-1$ more path lengths each. □

What is the running time of the algorithm?

TSP via Dynamic Programming (Bellman-Held-Karp)

We can compute all OPT$[S, c_i]$ values in order of increasing size of S.

1: **function** TSP($\{c_1, \dots, c_n\}$,d)
2: **for** $i \leftarrow 2 \dots n$ **do**
3: OPT$[\{c_i\}, c_i] \leftarrow d(c_1, c_i)$
4: **for** $j \leftarrow 2 \dots n-1$ **do**
5: **for** $S \subseteq \{c_2, \dots, c_n\}$ with $|S| = j$ **do**
6: **for** $c_i \in S$ **do**
7: OPT$[S, c_i] \leftarrow \min\{\text{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i) \mid c_k \in S \setminus \{c_i\}\}$
8: **return** $\min\{\text{OPT}[\{c_2, \dots c_n\}, c_i] + d(c_i, c_1) \mid c_i \in \{c_2, \dots, c_n\}\}$

Lemma

The above procedure solves TSP by computing $\mathcal{O}(n^2 \cdot 2^n)$ shortest paths.

Proof.

line 4.

The number of path lengths computed in line 7 is

$$\sum_{j=2}^{n-1} \binom{n-1}{j} \sum_{i=1}^{j} (j-1) \quad \leq \quad n^2 \sum_{j=1}^{n} \binom{n}{j} \quad = \quad n^2 \cdot 2^n$$

And lines 3 and 8 compute only $n-1$ more path lengths each. $\square$

What is the running time of the algorithm?

TSP via Dynamic Programming (Bellman-Held-Karp)

We can compute all $\mathrm{OPT}[S, c_i]$ values in order of increasing size of S.

1: **function** TSP($\{c_1, \ldots, c_n\}$,d)
2:     **for** $i \leftarrow 2 \ldots n$ **do**
3:         $\mathrm{OPT}[\{c_i\}, c_i] \leftarrow d(c_1, c_i)$
4:     **for** $j \leftarrow 2 \ldots n-1$ **do**
5:         **for** $S \subseteq \{c_2, \ldots, c_n\}$ with $|S| = j$ **do**
6:             **for** $c_i \in S$ **do**
7:                 $\mathrm{OPT}[S, c_i] \leftarrow \min\{\mathrm{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i) \mid c_k \in S \setminus \{c_i\}\}$
8:     **return** $\min\{\mathrm{OPT}[\{c_2, \ldots c_n\}, c_i] + d(c_i, c_1) \mid c_i \in \{c_2, \ldots, c_n\}\}$

Lemma

The above procedure solves TSP by computing $\mathcal{O}(n^2 \cdot 2^n)$ shortest paths.

Proof.

共有n个城市-1个， take any subset of j of them

The number of path lengths computed in line 7 is

$$\sum_{j=2}^{n-1} \binom{n-1}{j} \sum_{i=1}^{j} (j-1) \quad \leq \quad n^2 \sum_{j=1}^{n} \binom{n}{j} \quad = \quad n^2 \cdot 2^n$$

And lines 3 and 8 compute only $n-1$ more path lengths each. $\square$

What is the running time of the algorithm?

TSP via Dynamic Programming (Bellman-Held-Karp)

We can compute all OPT$[S, c_i]$ values in order of increasing size of S.

```
1: function TSP({c_1, ..., c_n}, d)
2:     for i ← 2 ... n do
3:         OPT[{c_i}, c_i] ← d(c_1, c_i)
4:     for j ← 2 ... n − 1 do
5:         for S ⊆ {c_2, ..., c_n} with |S| = j do
6:             for c_i ∈ S do
7:                 OPT[S, c_i] ← min{OPT[S \ {c_i}, c_k] + d(c_k, c_i) | c_k ∈ S \ {c_i}}
8:     return min{OPT[{c_2, ... c_n}, c_i] + d(c_i, c_1) | c_i ∈ {c_2, ..., c_n}}
```

Lemma

The above procedure solves TSP by computing $\mathcal{O}(n^2 \cdot 2^n)$ shortest paths.

Proof.

for each subset, we pick one city to be the last

The number of path lengths computed in line 7 is

$$\sum_{j=2}^{n-1} \binom{n-1}{j} \sum_{i=1}^{j} (j-1) \quad \leq \quad n^2 \sum_{j=1}^{n} \binom{n}{j} \quad = \quad n^2 \cdot 2^n$$

And lines 3 and 8 compute only $n-1$ more path lengths each. □

What is the running time of the algorithm?

TSP via Dynamic Programming (Bellman-Held-Karp)

We can compute all $\mathrm{OPT}[S, c_i]$ values in order of increasing size of S.

```
1: function TSP({c_1, ..., c_n}, d)
2:     for i ← 2 ... n do
3:         OPT[{c_i}, c_i] ← d(c_1, c_i)
4:     for j ← 2 ... n − 1 do
5:         for S ⊆ {c_2, ..., c_n} with |S| = j do
6:             for c_i ∈ S do
7:                 OPT[S, c_i] ← min{OPT[S \ {c_i}, c_k] + d(c_k, c_i) | c_k ∈ S \ {c_i}}
8:     return min{OPT[{c_2, ... c_n}, c_i] + d(c_i, c_1) | c_i ∈ {c_2, ..., c_n}}
```

Lemma

The above procedure solves TSP by computing $\mathcal{O}(n^2 \cdot 2^n)$ shortest paths.

Proof.

The number of path lengths computed in line 7 is

these can both be upper bounded by n so that's n^2.

$$\sum_{j=2}^{n-1} \binom{n-1}{j} \boxed{\sum_{i=1}^{j} (j-1)} \quad \leq \quad n^2 \sum_{j=1}^{n} \binom{n}{j} \quad = \quad n^2 \cdot 2^n$$

And lines 3 and 8 compute only $n - 1$ more path lengths each. □

What is the running time of the algorithm?

TSP via Dynamic Programming (Bellman-Held-Karp)

We can compute all $\mathrm{OPT}[S, c_i]$ values in order of increasing size of S.

1: **function** $\mathrm{TSP}(\{c_1, \dots, c_n\}, d)$
2:     **for** $i \leftarrow 2 \dots n$ **do**
3:         $\mathrm{OPT}[\{c_i\}, c_i] \leftarrow d(c_1, c_i)$
4:     **for** $j \leftarrow 2 \dots n-1$ **do**
5:         **for** $S \subseteq \{c_2, \dots, c_n\}$ with $|S| = j$ **do**
6:             **for** $c_i \in S$ **do**
7:                 $\mathrm{OPT}[S, c_i] \leftarrow \min\{\mathrm{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i) \mid c_k \in S \setminus \{c_i\}\}$
8:     **return** $\min\{\mathrm{OPT}[\{c_2, \dots c_n\}, c_i] + d(c_i, c_1) \mid c_i \in \{c_2, \dots, c_n\}\}$

Lemma

The above procedure solves TSP by computing $\mathcal{O}(n^2 \cdot 2^n)$ shortest paths.

Proof.

The number of path lengths computed in line 7 is

total number of subsets of n size

$$\sum_{j=2}^{n-1} \binom{n-1}{j} \sum_{i=1}^{j} (j-1) \quad \leq \quad n^2 \sum_{j=1}^{n} \binom{n}{j} \quad = \quad n^2 \cdot 2^n$$

And lines 3 and 8 compute only $n-1$ more path lengths each. □

What is the running time of the algorithm?

TSP via Dynamic Programming (Bellman-Held-Karp)

We can compute all $\mathrm{OPT}[S, c_i]$ values in order of increasing size of S.

1: **function** $\mathrm{TSP}(\{c_1, \ldots, c_n\}, d)$
2:     **for** $i \leftarrow 2 \ldots n$ **do**
3:         $\mathrm{OPT}[\{c_i\}, c_i] \leftarrow d(c_1, c_i)$
4:     **for** $j \leftarrow 2 \ldots n-1$ **do**
5:         **for** $S \subseteq \{c_2, \ldots, c_n\}$ with $|S| = j$ **do**
6:             **for** $c_i \in S$ **do**
7:                 $\mathrm{OPT}[S, c_i] \leftarrow \min\{\mathrm{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i) \mid c_k \in S \setminus \{c_i\}\}$
8:     **return** $\min\{\mathrm{OPT}[\{c_2, \ldots c_n\}, c_i] + d(c_i, c_1) \mid c_i \in \{c_2, \ldots, c_n\}\}$

Lemma

The above procedure solves TSP by computing $\mathcal{O}(n^2 \cdot 2^n)$ shortest paths.

Proof.

The number of path lengths computed in line 7 is

$$\sum_{j=2}^{n-1} \binom{n-1}{j} \sum_{i=1}^{j} (j-1) \quad \leq \quad n^2 \sum_{j=1}^{n} \binom{n}{j} \quad = \quad n^2 \cdot 2^n$$

And lines 3 and 8 compute only $n-1$ more path lengths each. $\square$

What is the running time of the algorithm?

TSP via Dynamic Programming (Bellman-Held-Karp)

We can compute all OPT$[S, c_i]$ values in order of increasing size of S.

1: **function** TSP($\{c_1, \ldots, c_n\}$,d)
2:     **for** $i \leftarrow 2 \ldots n$ **do**
3:         $\mathrm{OPT}[\{c_i\}, c_i] \leftarrow d(c_1, c_i)$
4:     **for** $j \leftarrow 2 \ldots n-1$ **do**
5:         **for** $S \subseteq \{c_2, \ldots, c_n\}$ with $|S| = j$ **do**
6:             **for** $c_i \in S$ **do**
7:                 $\mathrm{OPT}[S, c_i] \leftarrow \min\{\mathrm{OPT}[S \setminus \{c_i\}, c_k] + d(c_k, c_i) \mid c_k \in S \setminus \{c_i\}\}$
8:     **return** $\min\{\mathrm{OPT}[\{c_2, \ldots c_n\}, c_i] + d(c_i, c_1) \mid c_i \in \{c_2, \ldots, c_n\}\}$

Lemma

The above procedure solves TSP by computing $\mathcal{O}(n^2 \cdot 2^n)$ shortest paths.

Proof.

The number of path lengths computed in line 7 is

$$\sum_{j=2}^{n-1} \binom{n-1}{j} \sum_{i=1}^{j} (j-1) \quad \leq \quad n^2 \sum_{j=1}^{n} \binom{n}{j} \quad = \quad n^2 \cdot 2^n$$

$\mathcal{O}^*\binom{2^n}{\square}$

And lines 3 and 8 compute only $n-1$ more path lengths each.

What is the running time of the algorithm? $\mathcal{O}^\star(2^n)$ if we assume additions take at most polynomial time in n. Much better than $\mathcal{O}^\star(n!)$.

最快的TSP

Dynamic Programming in general

Similar to "Divide and Conquer" in that it requires "Optimal Substructure" but subproblems may be overlapping. 要求子问题最佳

Instead of recursively solving smaller disjoint subproblems, "Dynamic Programming" solves all smaller subproblems in order of increasing size.

A hybrid idea called "Memoization" (not "Memorization"!) does the same by using recursion, but caching results so each subproblem is only solved once.

In our TSP example the original problem does not have the optimal substructure property (A piece of an optimal tour does not have to be an optimal tour of some subgraph). The trick is to notice that the problem of computing $\mathrm{OPT}[\{c_2, \ldots, c_n\}, c_i]$ *does* have the property, and that TSP can be solved once we know that for all c_i.

Dynamic Programming in general

Similar to "Divide and Conquer" in that it requires "Optimal Substructure" but subproblems may be overlapping.

Instead of recursively solving smaller disjoint subproblems, "Dynamic Programming" solves all smaller subproblems in order of increasing size.

按大小顺序解决所有的子问题

A hybrid idea called "Memoization" (not "Memorization"!) does the same by using recursion, but caching results so each subproblem is only solved once.

In our TSP example the original problem does not have the optimal substructure property (A piece of an optimal tour does not have to be an optimal tour of some subgraph). The trick is to notice that the problem of computing $\mathrm{OPT}[\{c_2, \ldots, c_n\}, c_i]$ *does* have the property, and that TSP can be solved once we know that for all c_i.

Dynamic Programming in general

Similar to "Divide and Conquer" in that it requires "Optimal Substructure" but subproblems may be overlapping.

Instead of recursively solving smaller disjoint subproblems, "Dynamic Programming" solves all smaller subproblems in order of increasing size.

A hybrid idea called "Memoization" (not "Memo**r**ization"!) does the same by using recursion, but caching results so each subproblem is only solved once.

类似TSP中OPT记住了前面计算的结果，就跟cache一样，无需重复计算，所有子问题只算一次。

In our TSP example the original problem does not have the optimal substructure property (A piece of an optimal tour does not have to be an optimal tour of some subgraph). The trick is to notice that the problem of computing $OPT[\{c_2, \ldots, c_n\}, c_i]$ *does* have the property, and that TSP can be solved once we know that for all c_i.

Dynamic Programming in general

Similar to "Divide and Conquer" in that it requires "Optimal Substructure" but subproblems may be overlapping.

Instead of recursively solving smaller disjoint subproblems, "Dynamic Programming" solves all smaller subproblems in order of increasing size.

A hybrid idea called "Memoization" (not "Memo**r**ization"!) does the same by using recursion, but caching results so each subproblem is only solved once.

In our TSP example the original problem does not have the optimal substructure property (A piece of an optimal tour does not have to be an optimal tour of some subgraph). The trick is to notice that the problem of computing $\text{OPT}[\{c_2, \ldots, c_n\}, c_i]$ *does* have the property, and that TSP can be solved once we know that for all c_i.

MIS via Branching

Problem: Given undirected graph (V, E), find the maximum cardinality of $I \subseteq V$ so each edge has at most one endpoint in I.

Such a set I is called a *Maximum Independent Set (MIS)* for the graph.

Naive: Try all 2^n subsets (where $n = |V|$). This takes $\mathcal{O}^\star(2^n)$ time.

For $v \in V$ define $N[v] := \{v\} \cup \{w \in V \mid (v, w) \in E\}$. This is called the *closed neighborhood* of v.

Observation: $N[v] \cap I \neq \emptyset$ for all $v \in V$ and all MIS I.
Why?

function MISsize($G = (V, E)$)
  if $V = \emptyset$ **then return** 0
  $v \leftarrow$ vertex in V of minimum degree.
  return $1 + \max\{\text{MISsize}(G \setminus N[w]) \mid w \in N[v]\}$

MIS via Branching

Problem: Given undirected graph (V, E), find the maximum cardinality of $I \subseteq V$ so each edge has at most one endpoint in I.

Such a set I is called a *Maximum Independent Set (MIS)* for the graph.

Naive: Try all 2^n subsets (where $n = |V|$). This takes $\mathcal{O}^\star(2^n)$ time.

For $v \in V$ define $N[v] := \{v\} \cup \{w \in V \mid (v, w) \in E\}$. This is called the *closed neighborhood* of v.

Observation: $N[v] \cap I \neq \emptyset$ for all $v \in V$ and all MIS I.
Why?

function MISsize($G = (V, E)$)
  if $V = \emptyset$ **then return** 0
  $v \leftarrow$ vertex in V of minimum degree.
  return $1 + \max\{\text{MISsize}(G \setminus N[w]) \mid w \in N[v]\}$

MIS via Branching

Problem: Given undirected graph (V, E), find the maximum cardinality of $I \subseteq V$ so each edge has at most one endpoint in I.

Such a set I is called a *Maximum Independent Set (MIS)* for the graph.

Naive: Try all 2^n subsets (where $n = |V|$). This takes $\mathcal{O}^\star(2^n)$ time.

For $v \in V$ define $N[v] := \{v\} \cup \{w \in V \mid (v, w) \in E\}$. This is called the *closed neighborhood* of v.

Observation: $N[v] \cap I \neq \emptyset$ for all $v \in V$ and all MIS I.
Why?

function MISsize($G = (V, E)$)
    if $V = \emptyset$ **then return** 0
    $v \leftarrow$ vertex in V of minimum degree.
    return $1 + \max\{\text{MISsize}(G \setminus N[w]) \mid w \in N[v]\}$

MIS via Branching

Problem: Given undirected graph (V, E), find the maximum cardinality of $I \subseteq V$ so each edge has at most one endpoint in I.

Such a set I is called a *Maximum Independent Set (MIS)* for the graph.

Naive: Try all 2^n subsets (where $n = |V|$). This takes $\mathcal{O}^\star(2^n)$ time.

For $v \in V$ define $N[v] := \{v\} \cup \{w \in V \mid (v, w) \in E\}$. This is called the *closed neighborhood* of v.

Observation: $N[v] \cap I \neq \emptyset$ for all $v \in V$ and all MIS I. 反证

Why? If $N[v] \cap I = \emptyset$ for some $v \in V$, $I \cup \{v\}$ would be a larger solution.

```
function MISsize(G = (V, E))
    if V = ∅ then return 0
    v ← vertex in V of minimum degree.
    return 1 + max{MISsize(G \ N[w]) | w ∈ N[v]}
```

MIS via Branching

Problem: Given undirected graph (V, E), find the maximum cardinality of $I \subseteq V$ so each edge has at most one endpoint in I.

Such a set I is called a *Maximum Independent Set (MIS)* for the graph.

Naive: Try all 2^n subsets (where $n = |V|$). This takes $\mathcal{O}^\star(2^n)$ time.

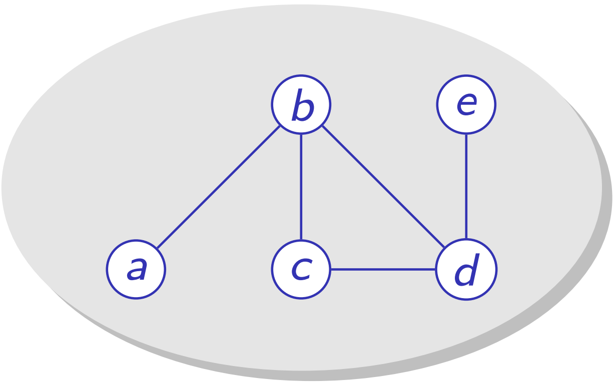
For $v \in V$ define $N[v] := \{v\} \cup \{w \in V \mid (v, w) \in E\}$. This is called the *closed neighborhood* of v.

因为MVC是MIS的补
MVC到每个边有一端在，而另一个则在MIS里，而邻域是一个点及全部相关点，所以不可能出

Observation: $N[v] \cap I \neq \emptyset$ for all $v \in V$ and all MIS I.
Why? If $N[v] \cap I = \emptyset$ for some $v \in V$, $I \cup \{v\}$ would be a larger solution.

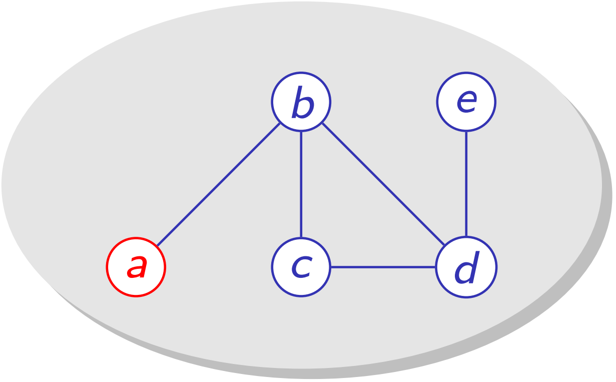
function MISsize($G = (V, E)$)
    if $V = \emptyset$ **then return** 0
    $v \leftarrow$ vertex in V of minimum degree.
    return $1 + \max\{\text{MISsize}(G \setminus N[w]) \mid w \in N[v]\}$

MIS via Branching



MIS via Branching

$v = a$

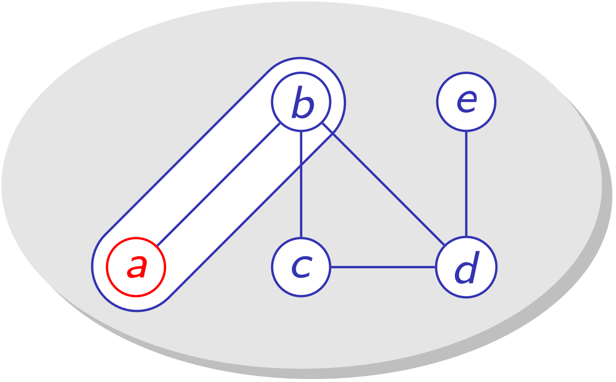



```
function MISsize(G = (V, E))
    if V = ∅ then return 0
    v ← vertex in V of minimum degree.
    return 1 + max{MISsize(G \ N[w]) | w ∈ N[v]}
```

MIS via Branching

$v = a$

$N[v] = \{a, b\}$

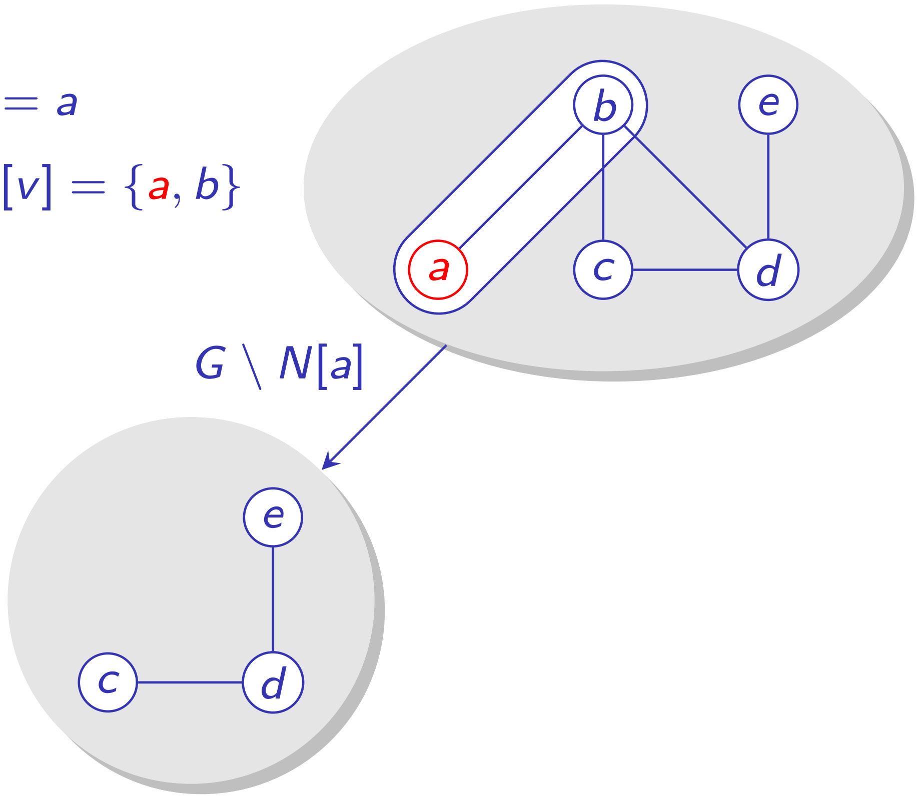


1: **function** MISsize($G = (V, E)$)
2: **if** $V = \emptyset$ **then return** 0 ✓
3: $v \leftarrow$ vertex in V of minimum degree.
4: **return** $1 + \max\{\text{MISsize}(G \setminus N[w]) \mid w \in N[v]\}$

MIS via Branching

$v = a$

$N[v] = \{a, b\}$

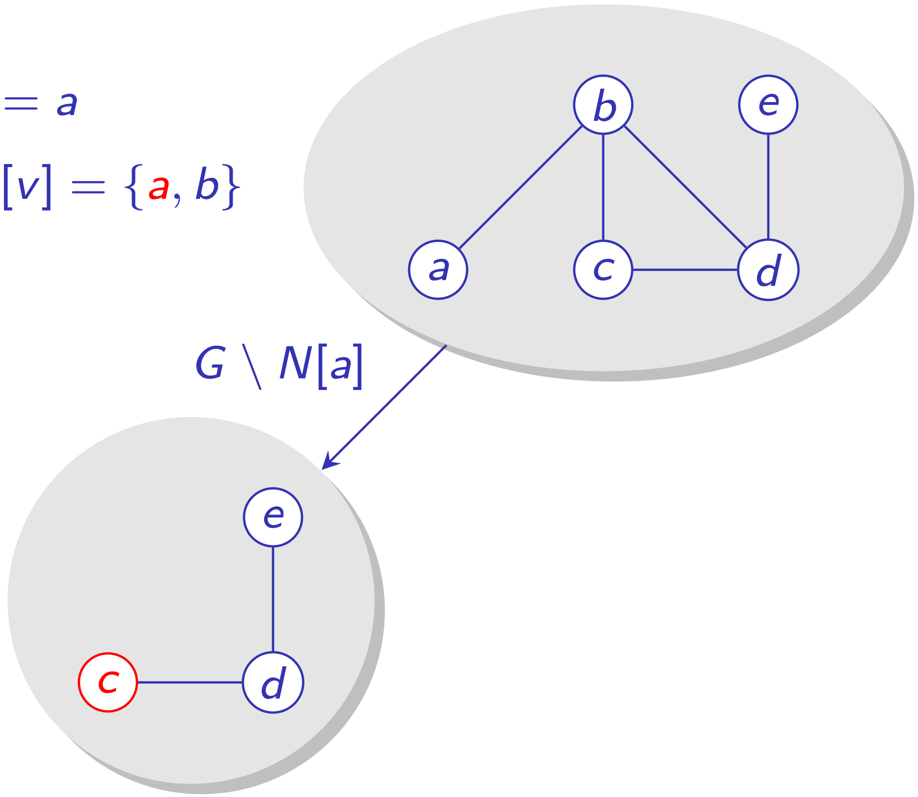


MIS via Branching

$v = a$

$N[v] = \{a, b\}$

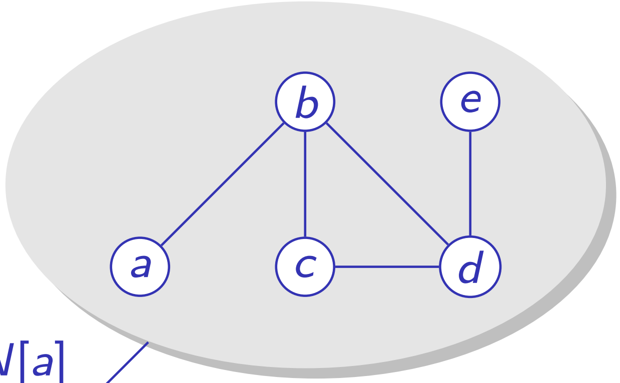
$v = c$



MIS via Branching

$v = a$

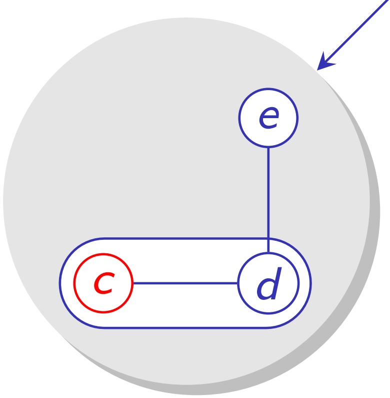
$N[v] = \{a, b\}$



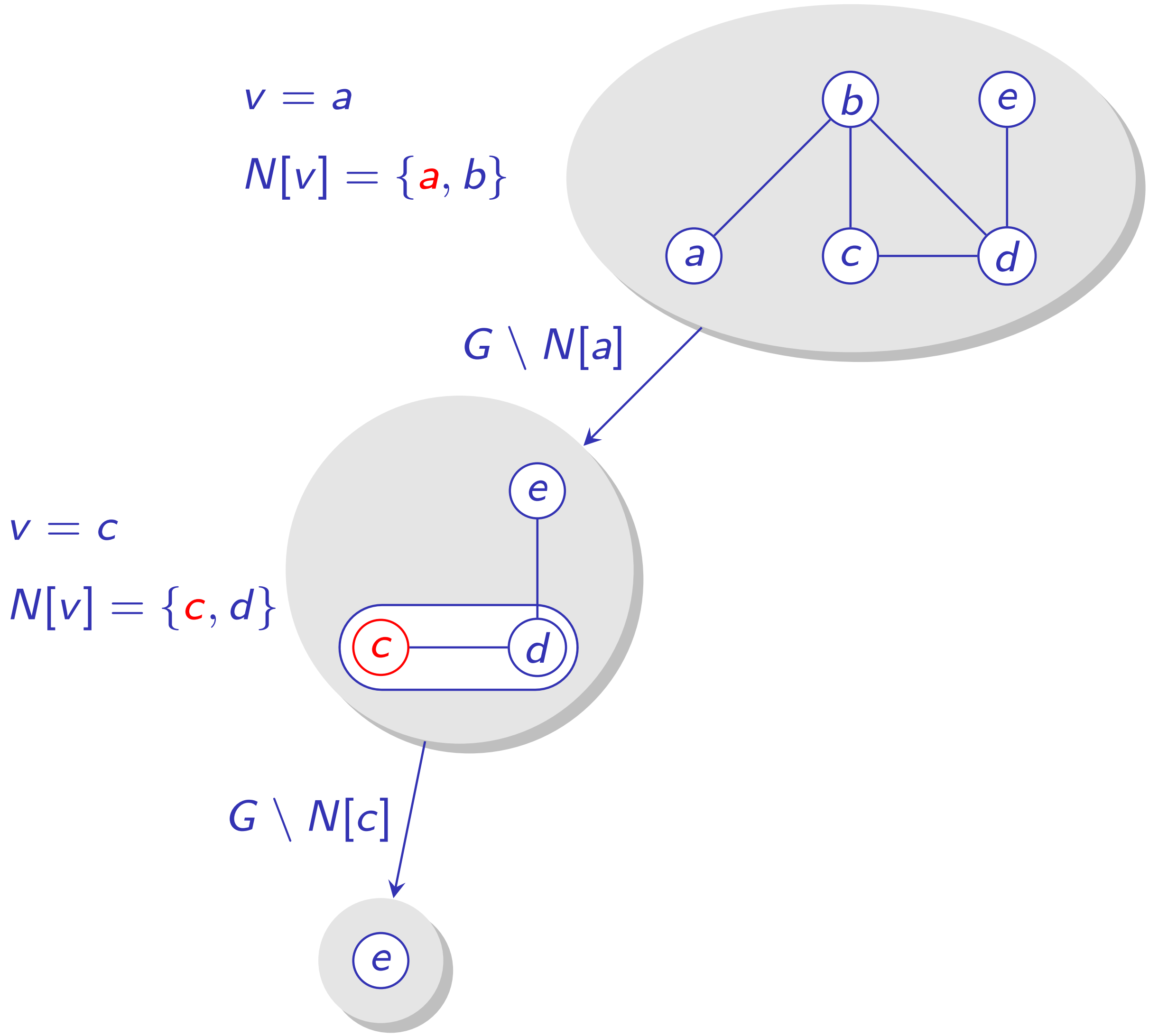
$G \setminus N[a]$

$v = c$

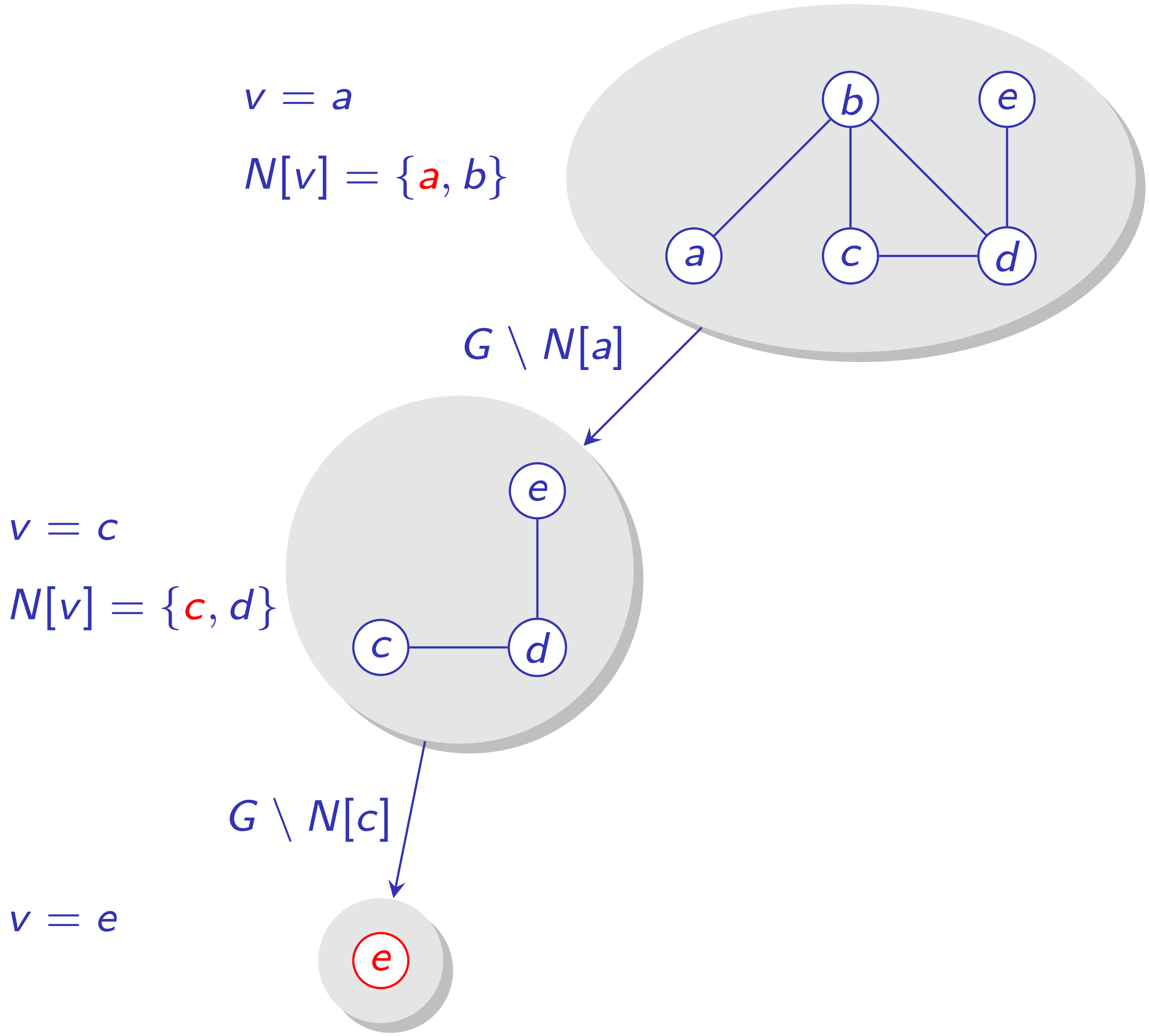
$N[v] = \{c, d\}$



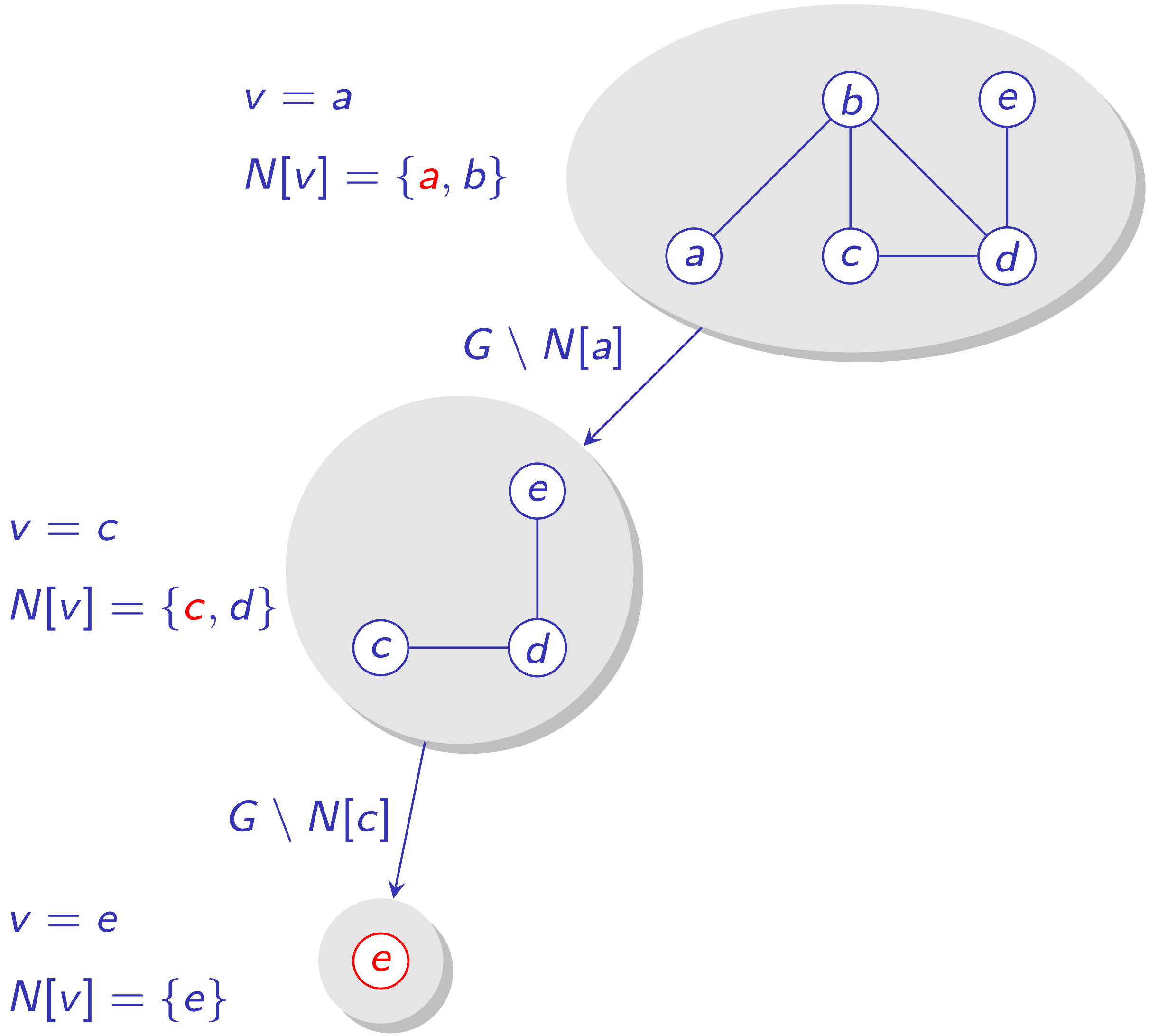
MIS via Branching



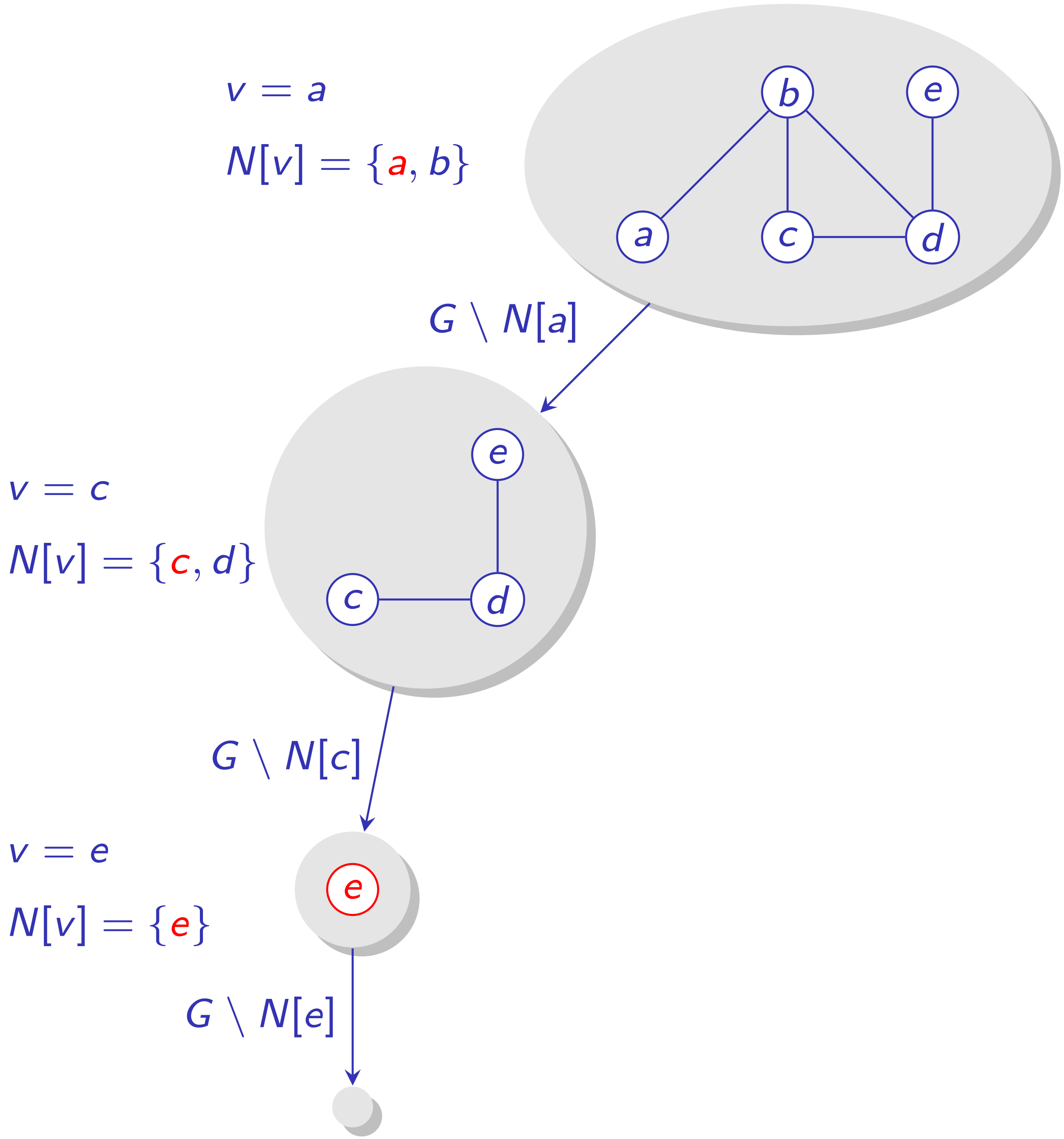
MIS via Branching



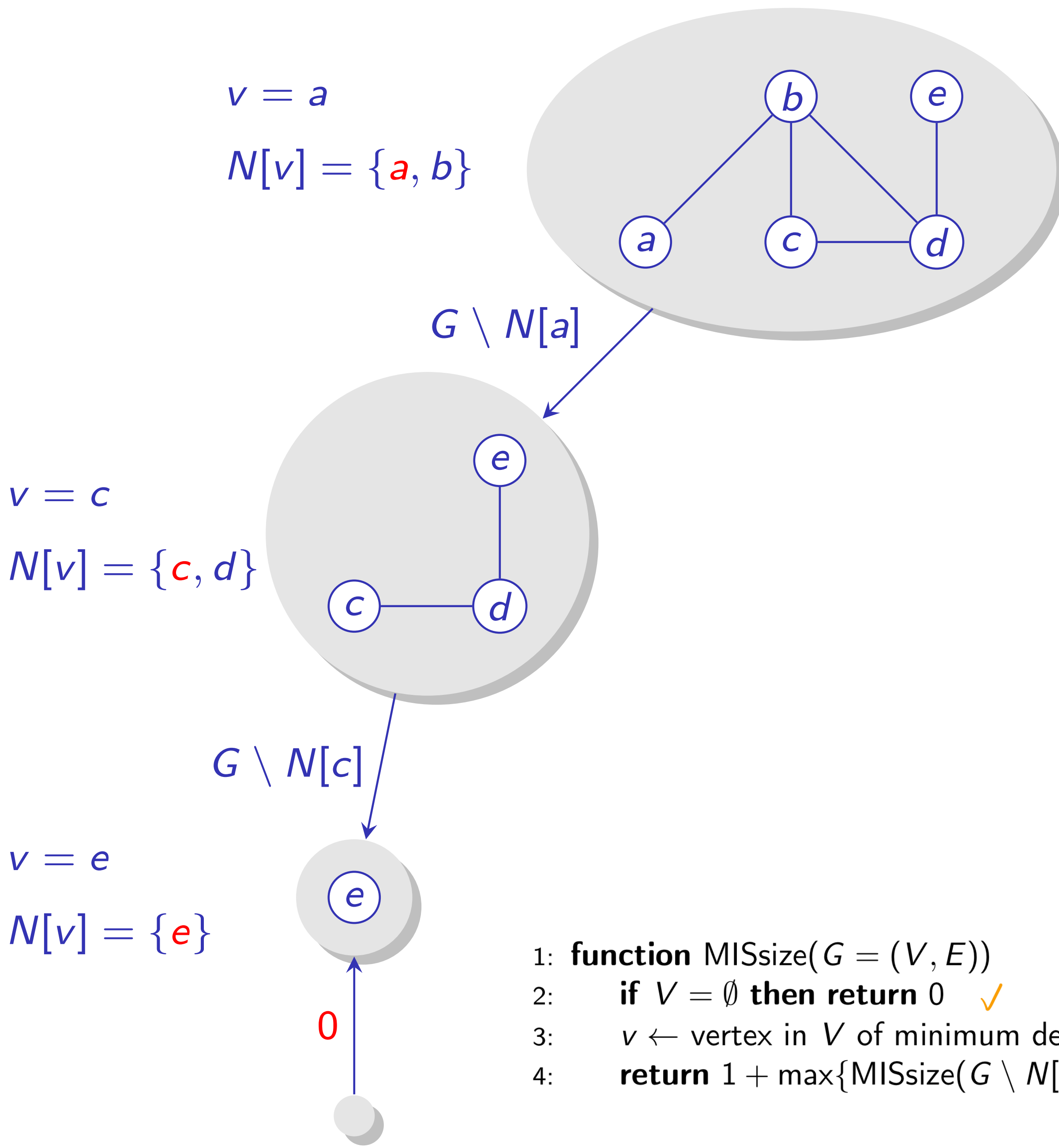
MIS via Branching



MIS via Branching



MIS via Branching



function MISsize($G = (V, E)$)
  if $V = \emptyset$ **then return** 0 ✓
  $v \leftarrow$ vertex in V of minimum degree.
  return $1 + \max\{\text{MISsize}(G \setminus N[w]) \mid w \in N[v]\}$

MIS via Branching

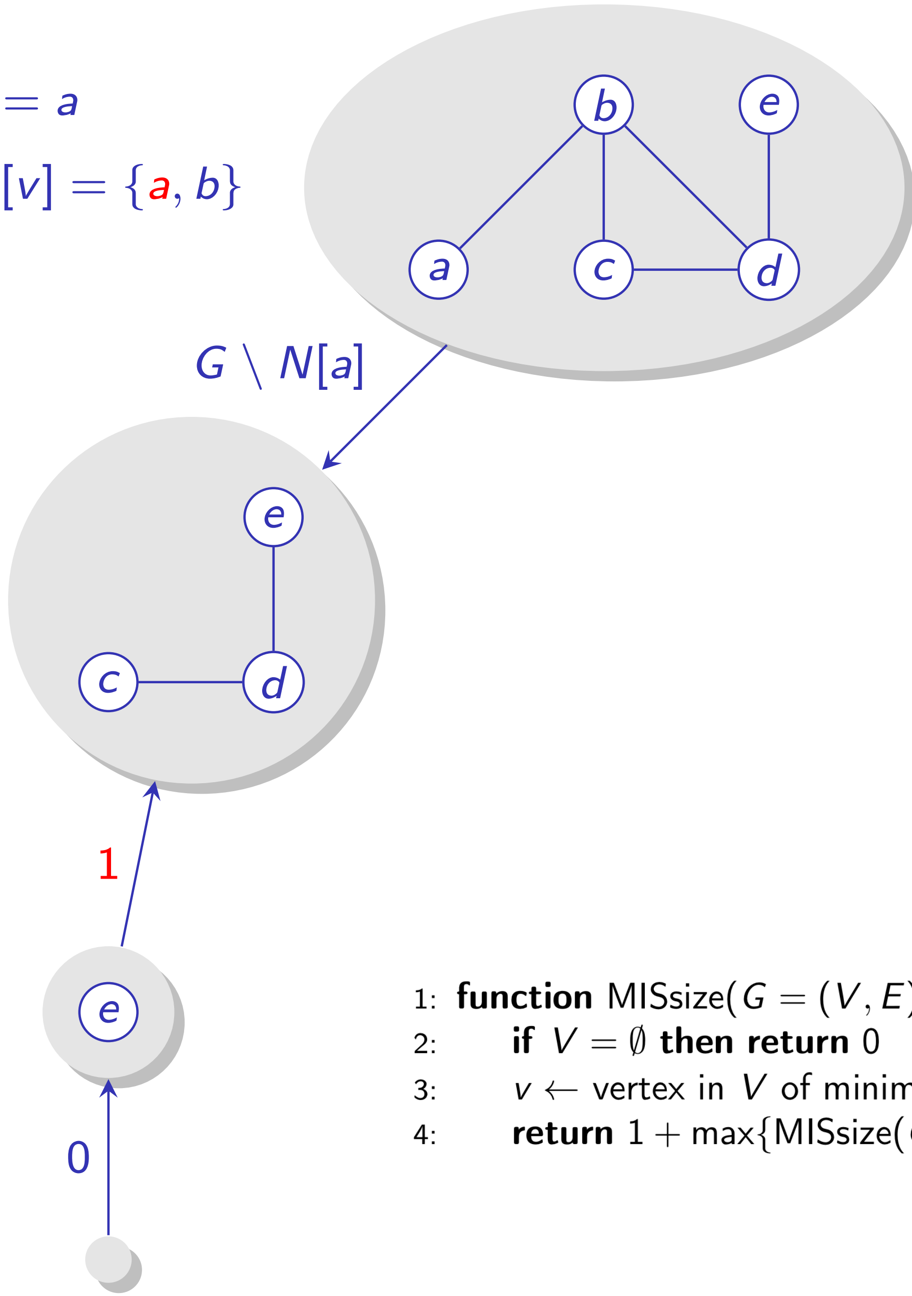
$v = a$

$N[v] = \{a, b\}$

$G \setminus N[a]$

$v = c$

$N[v] = \{c, d\}$

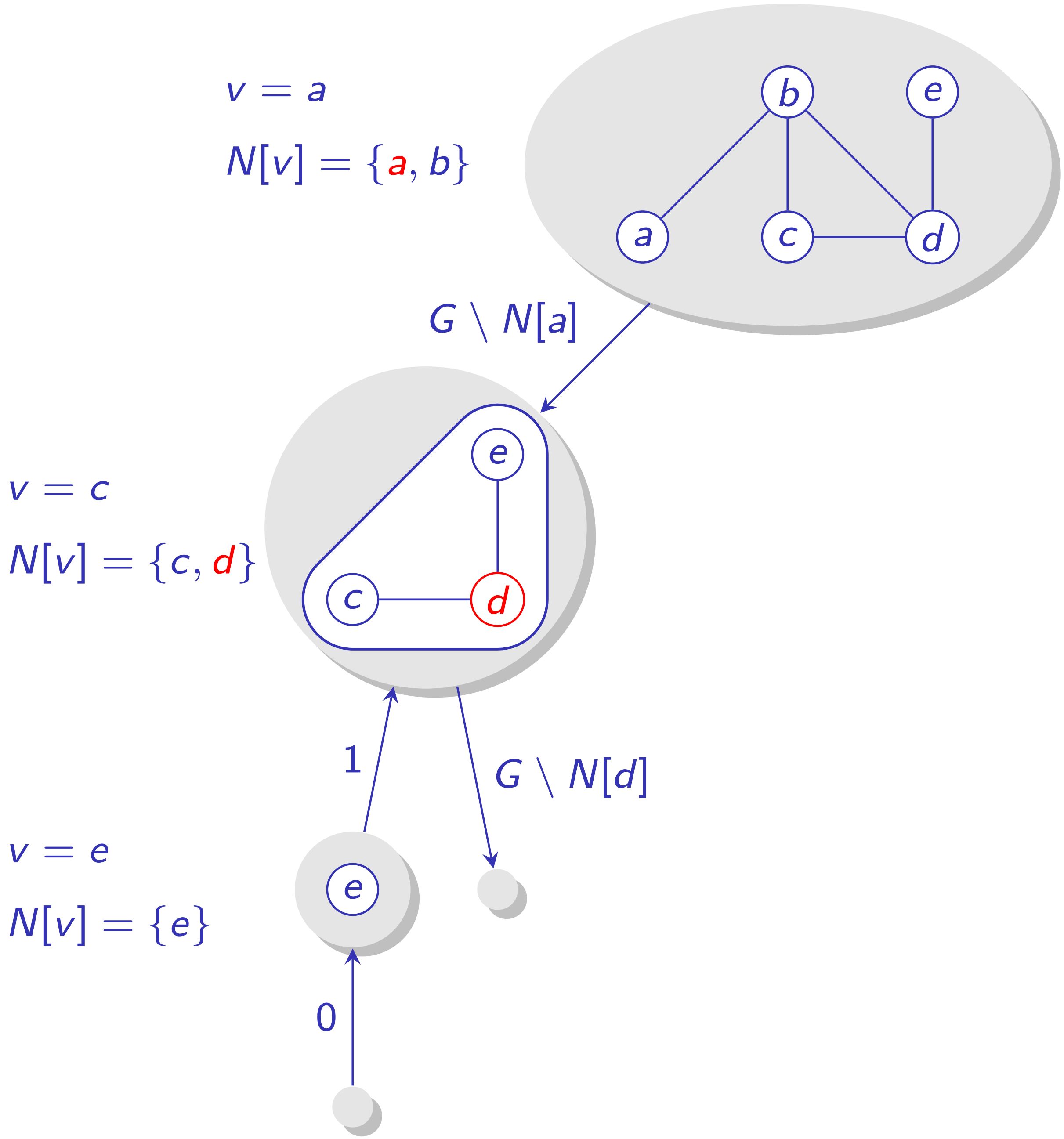


$v = e$

$N[v] = \{e\}$

function MISsize$(G = (V, E))$
  if $V = \emptyset$ **then return** 0 ✓
  $v \leftarrow$ vertex in V of minimum degree.
  return $1 + \max\{\text{MISsize}(G \setminus N[w]) \mid w \in N[v]\}$

MIS via Branching



MIS via Branching

$v = a$

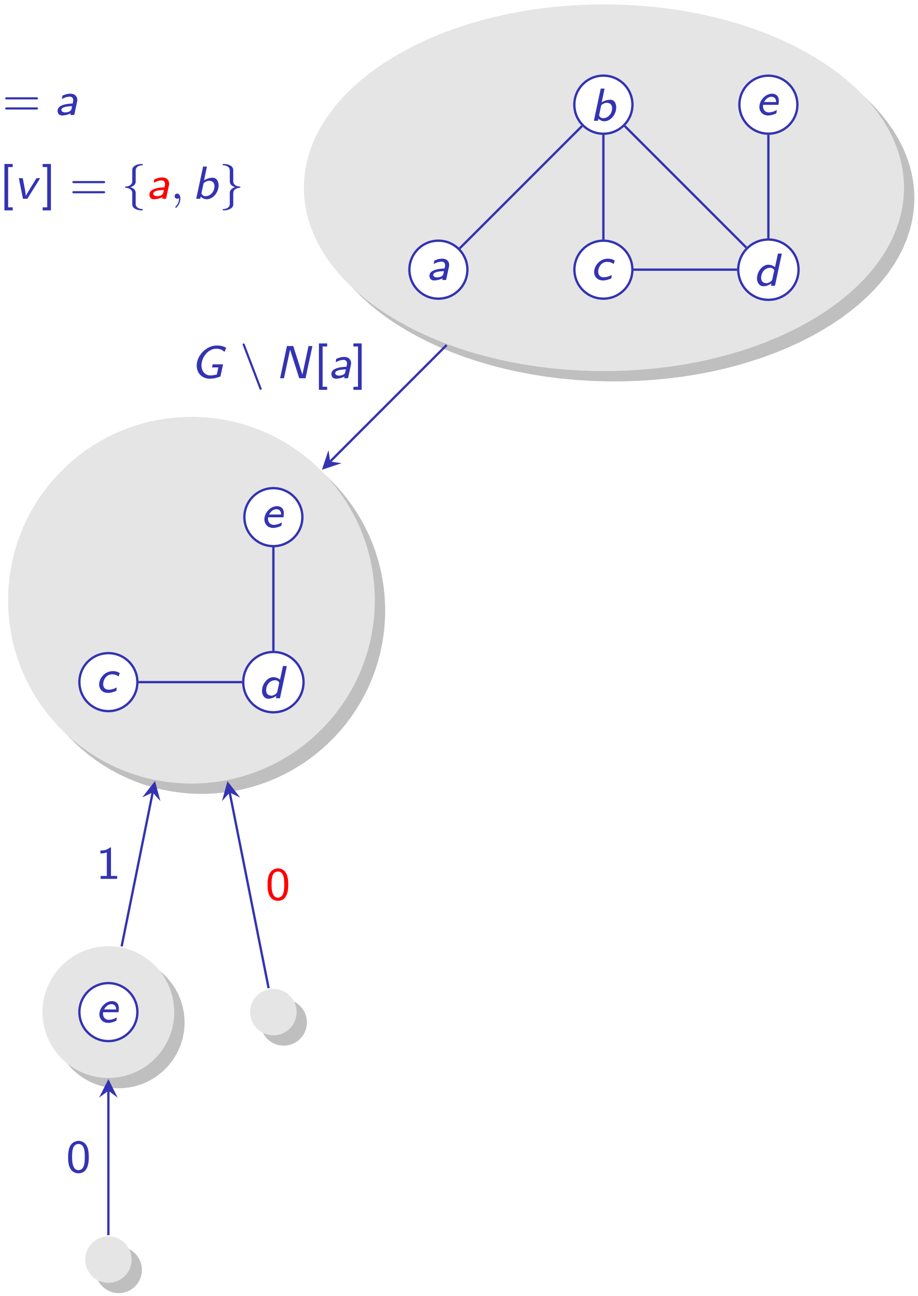
$N[v] = \{a, b\}$

$v = c$

$N[v] = \{c, d\}$

$v = e$

$N[v] = \{e\}$



MIS via Branching

$v = a$

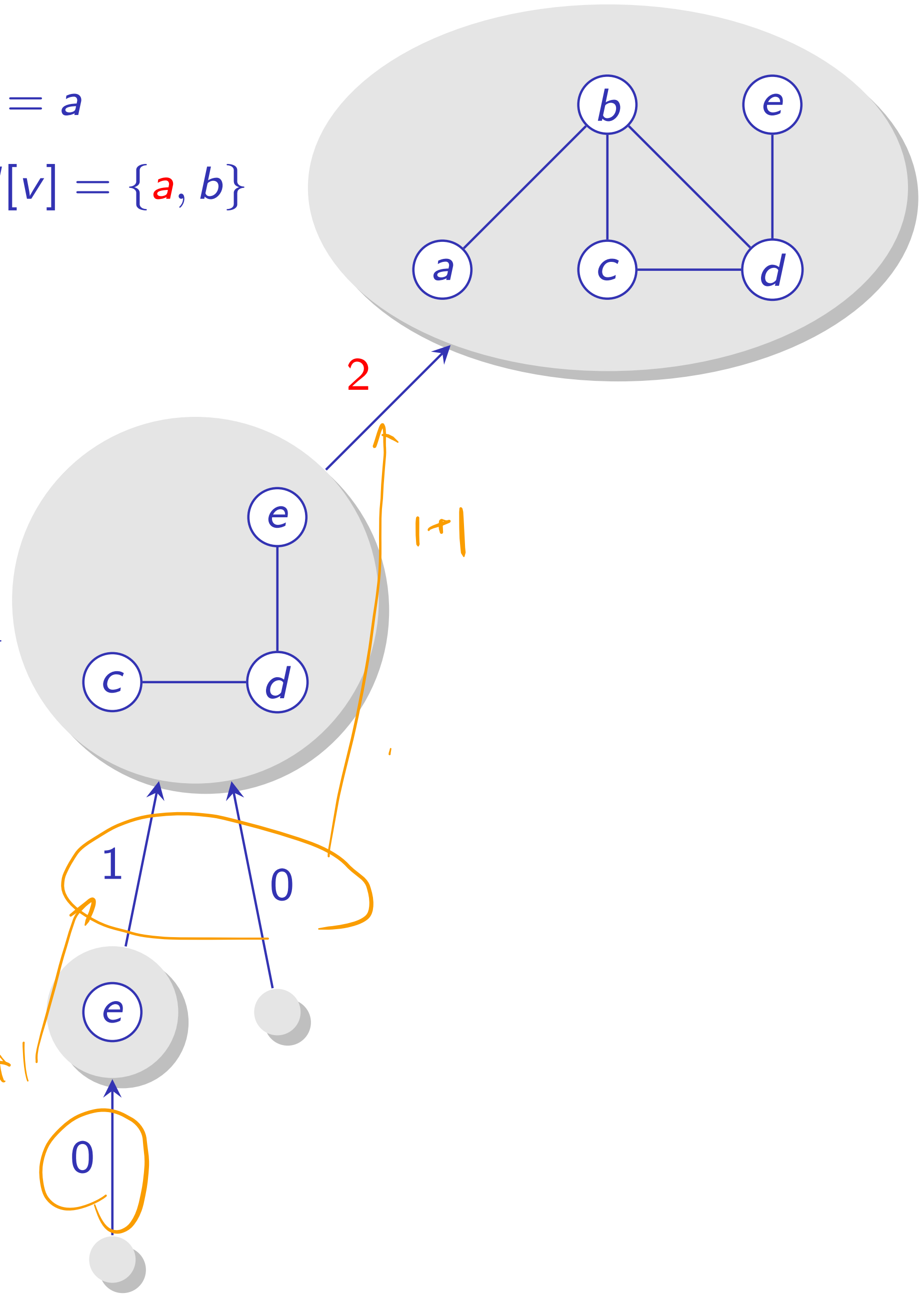
$N[v] = \{a, b\}$

$v = c$

$N[v] = \{c, d\}$

$v = e$

$N[v] = \{e\}$



MIS via Branching

$v = a$

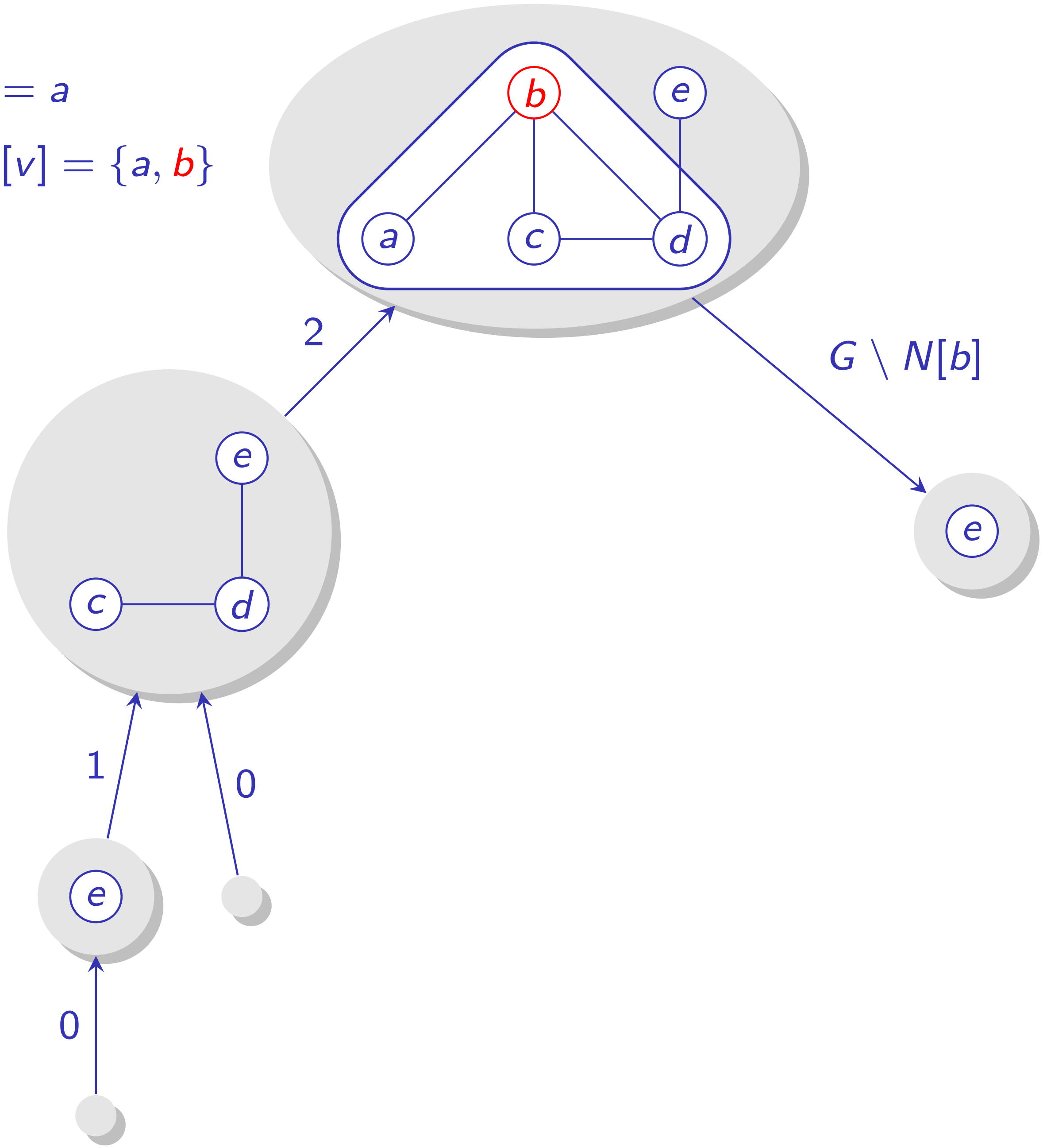
$N[v] = \{a, b\}$

$v = c$

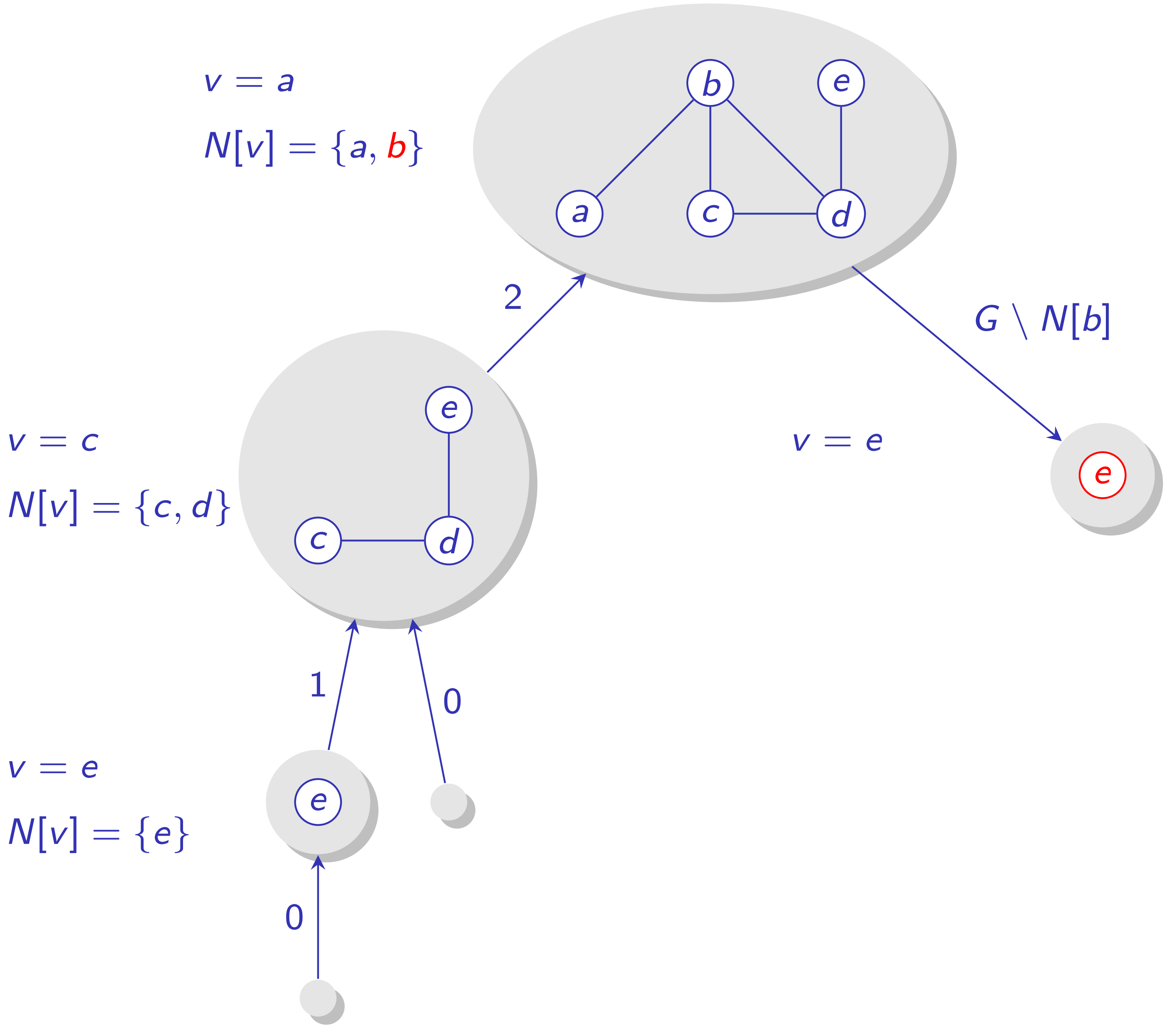
$N[v] = \{c, d\}$

$v = e$

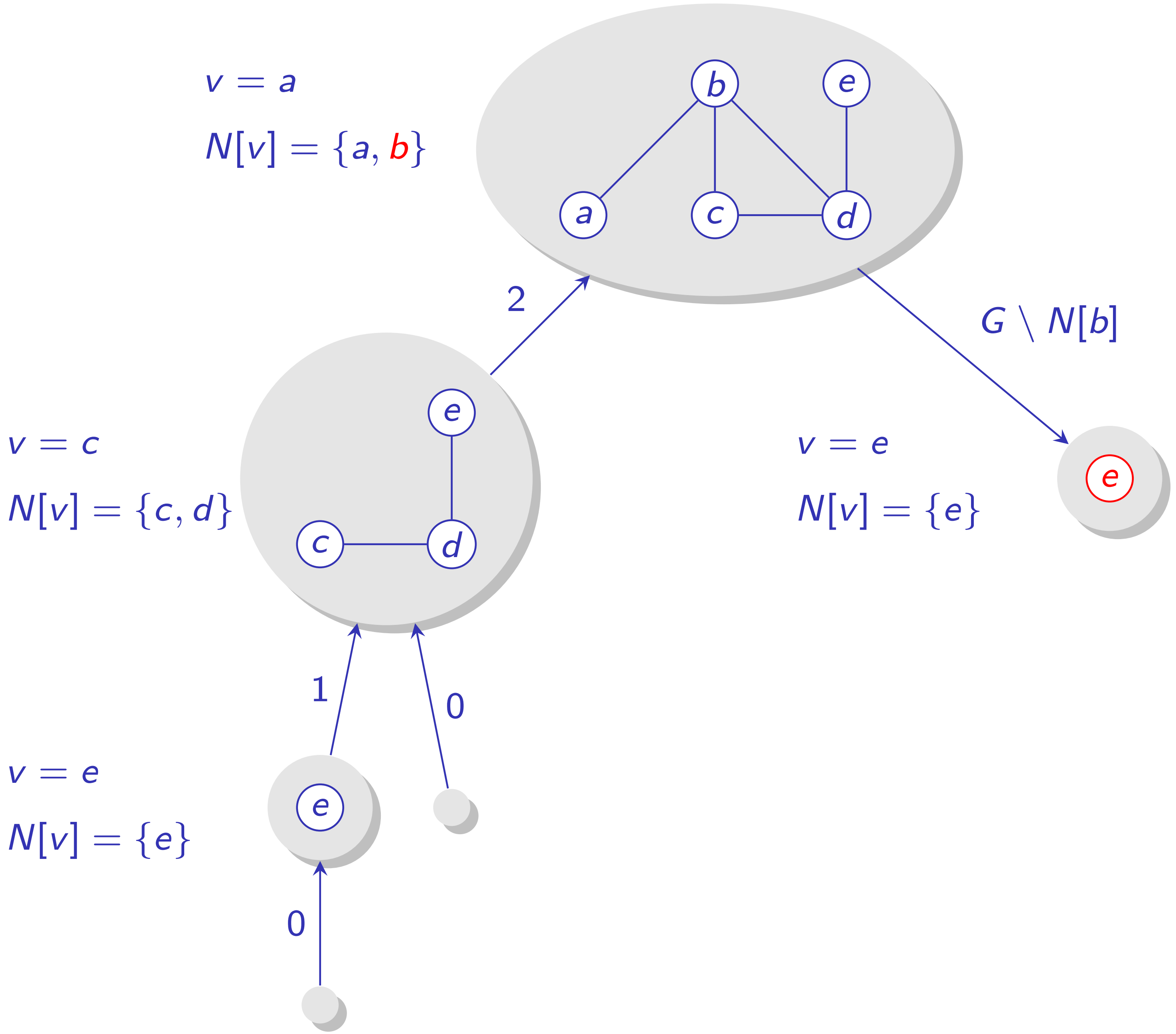
$N[v] = \{e\}$



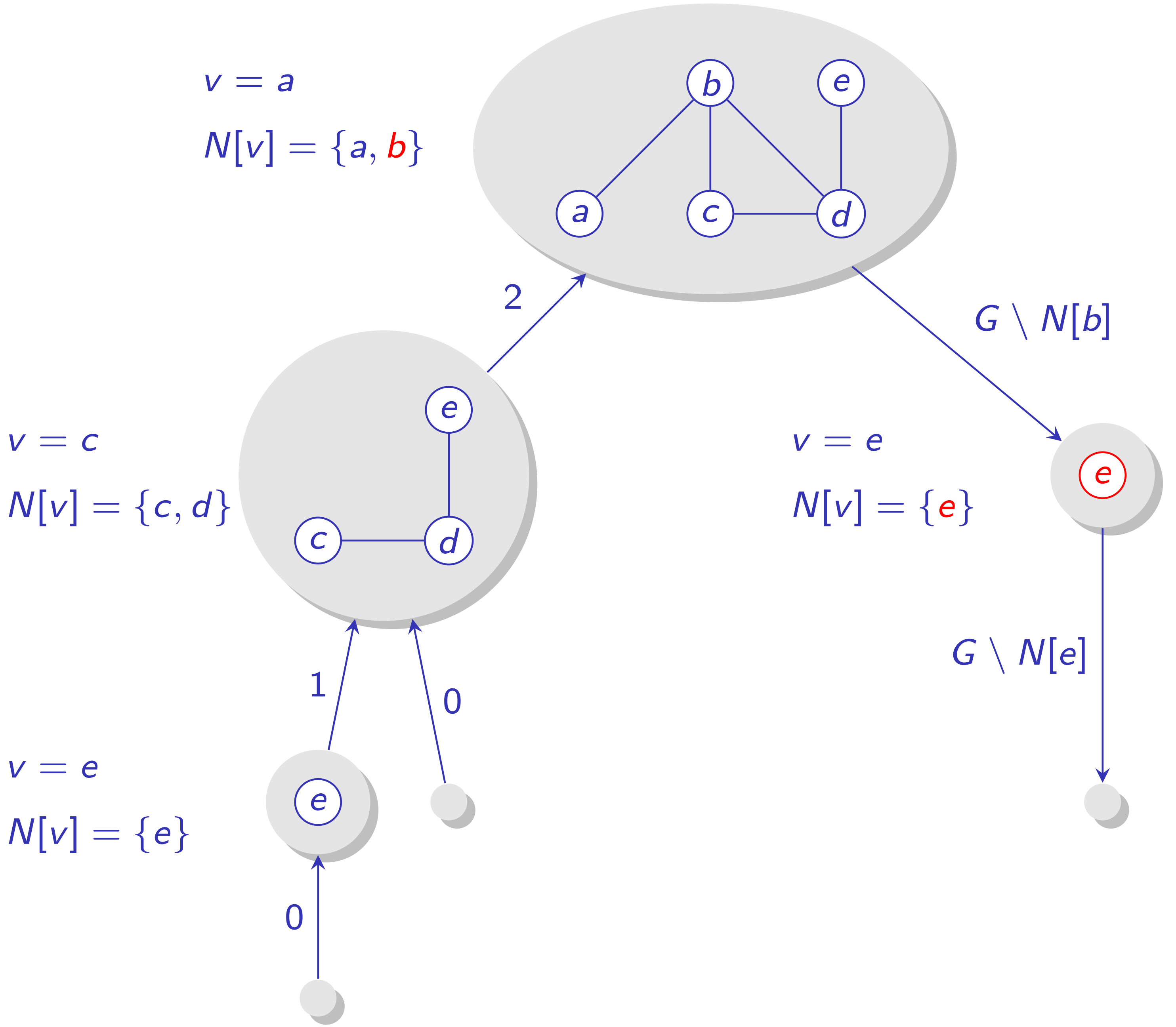
MIS via Branching



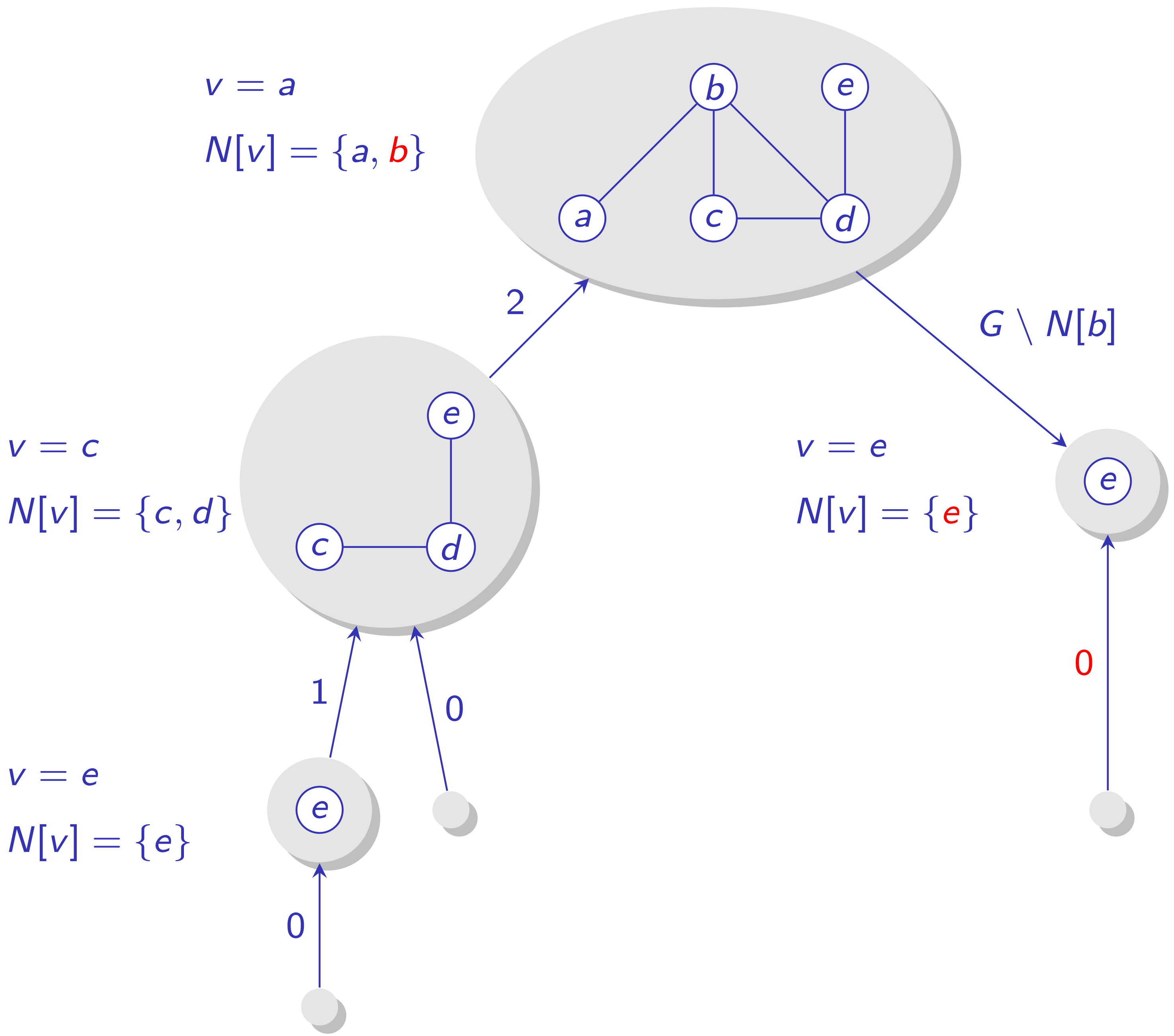
MIS via Branching



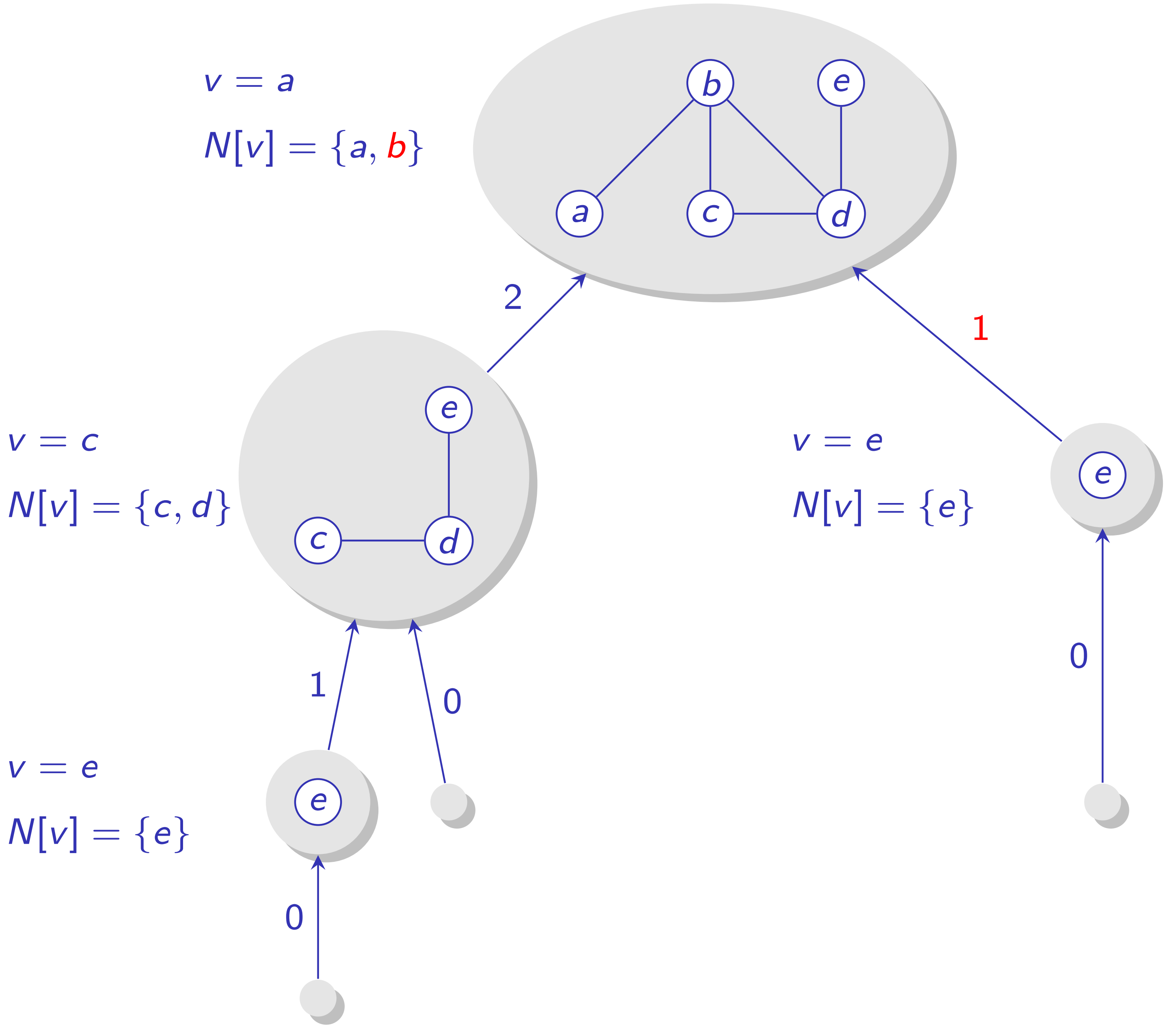
MIS via Branching



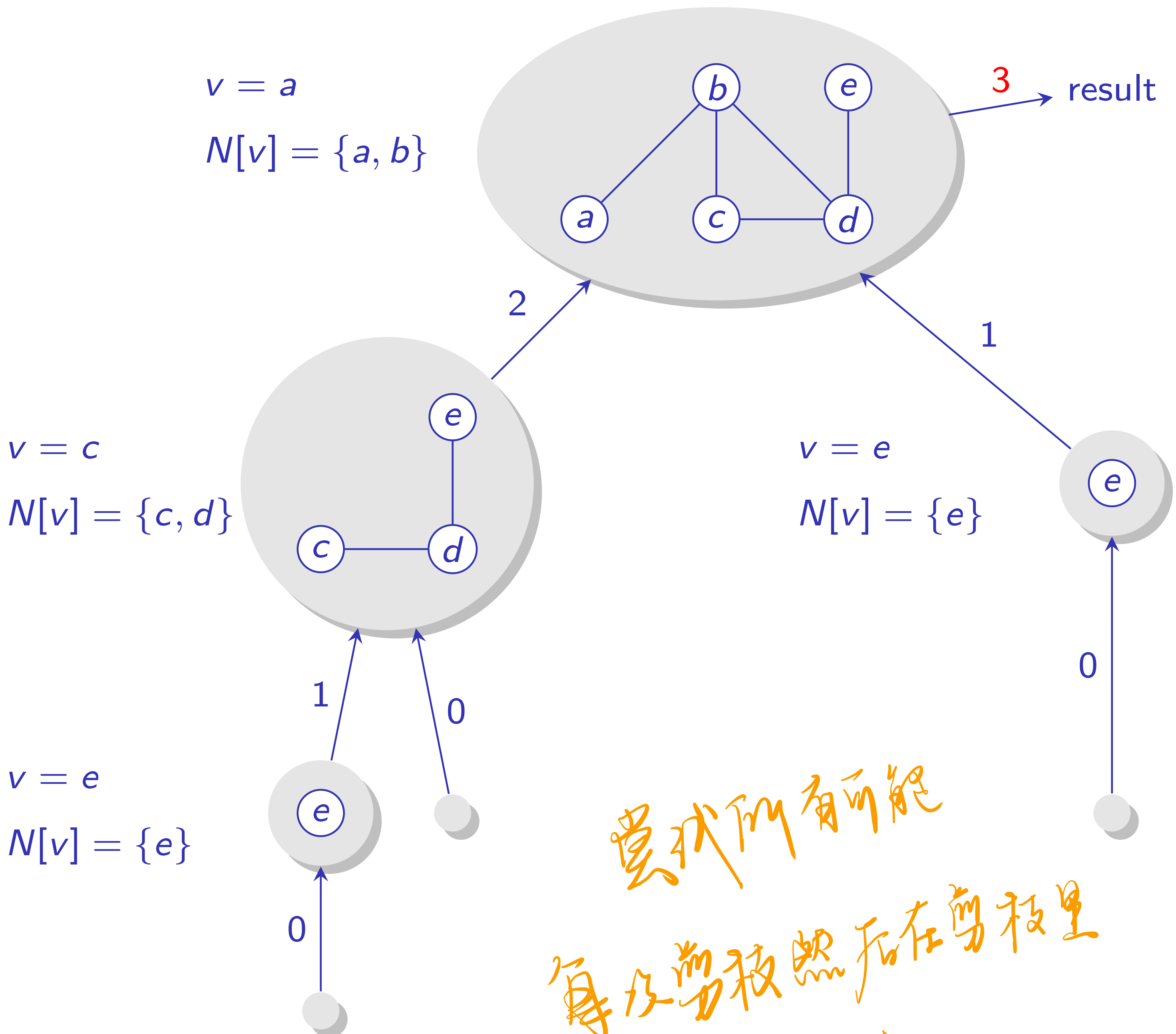
MIS via Branching



MIS via Branching



MIS via Branching



要找所有可能

每次剪枝然后在剪枝里

挑最大的+1

MIS via Branching   做写

Let $T(n)$ be the maximum number of subproblems considered by the branching algorithm on a graph with n vertices, then (very loosely):

$$T(0) = 1$$

$$T(n) \leq 1 + \sum_{w \in N[v]} T\Big(n - (d(w) + 1)\Big)$$ for some $v \in V$ of minimum degree

$$\leq 1 + (d(v) + 1) \cdot T\Big(n - (d(v) + 1)\Big)$$ since $T(\cdot)$ is nondecreasing

$$= 1 + s \cdot T(n - s)$$ where $s = d(v) + 1$ and thus $s \in \{1, \ldots, n\}$

Lemma

$T(n) \in \mathcal{O}(3^{n/3}) \subset \mathcal{O}(1.44225^n)$

"Proof" (spot the error).

$$T(n) \leq 1 + s \cdot T(n - s)$$

$$\leq 1 + s + s^2 + \cdots + s^{n/s}$$

$$= \frac{s^{1+n/s} - 1}{s - 1} < 2s^{n/s} \qquad (\text{for } s \geq 2)$$

$$\in \mathcal{O}(s^{n/s}) \subseteq \mathcal{O}(e^{n/e}) \qquad \square$$

MIS via Branching

Let $T(n)$ be the maximum number of subproblems considered by the branching algorithm on a graph with n vertices, then (very loosely):

$$T(0) = 1$$

$$T(n) \leq 1 + \sum_{w \in N[v]} T\Big(n - (d(w)+1)\Big)$$ for some $v \in V$ of minimum degree

$$\leq 1 + (d(v)+1) \cdot T\Big(n - (d(v)+1)\Big)$$ since $T(\cdot)$ is nondecreasing

$$= 1 + s \cdot T(n-s)$$ where $s = d(v)+1$ and thus $s \in \{1, \ldots, n\}$

Lemma

$T(n) \in \mathcal{O}(3^{n/3}) \subset \mathcal{O}(1.44225^n)$

"Proof" (spot the error).

$$T(n) \leq 1 + s \cdot T(n-s)$$

$$\leq 1 + s + s^2 + \cdots + s^{n/s}$$

$$= \frac{s^{1+n/s}-1}{s-1} < 2s^{n/s} \qquad \text{(for } s \geq 2\text{)}$$

$$\in \mathcal{O}(s^{n/s}) \subseteq \mathcal{O}(e^{n/e}) \qquad \square$$

MIS via Branching

Let $T(n)$ be the maximum number of subproblems considered by the branching algorithm on a graph with n vertices, then (very loosely):

$$T(0) = 1$$

$$T(n) \le 1 + \sum_{w \in N[v]} T\Big(n - (d(w) + 1)\Big) \qquad \text{for some } v \in V \text{ of minimum degree}$$

$$\le 1 + (d(v) + 1) \cdot T\Big(n - (d(v) + 1)\Big) \qquad \text{since } T(\cdot) \text{ is nondecreasing}$$

$$= 1 + s \cdot T(n - s) \qquad \text{where } s = d(v) + 1 \text{ and thus } s \in \{1, \dots, n\}$$

不是很懂

Lemma

$T(n) \in \mathcal{O}(3^{n/3}) \subset \mathcal{O}(1.44225^n)$

"Proof" (spot the error).

$$T(n) \le 1 + s \cdot T(n - s)$$

$$\le 1 + s + s^2 + \cdots + s^{n/s}$$

$$= \frac{s^{1+n/s} - 1}{s - 1} < 2s^{n/s} \qquad (\text{for } s \ge 2)$$

$$\in \mathcal{O}(s^{n/s}) \subseteq \mathcal{O}(e^{n/e}) \qquad \square$$

MIS via Branching

Let $T(n)$ be the maximum number of subproblems considered by the branching algorithm on a graph with n vertices, then (very loosely):

$$T(0) = 1$$

$$\begin{aligned} T(n) &\leq 1 + \sum_{w \in N[v]} T\Big(n - (d(w)+1)\Big) && \text{for some } v \in V \text{ of minimum degree} \\ &\leq 1 + (d(v)+1) \cdot T\Big(n - (d(v)+1)\Big) && \text{since } T(\cdot) \text{ is nondecreasing} \\ &= 1 + s \cdot T(n-s) && \text{where } s = d(v)+1 \text{ and thus } s \in \{1, \dots, n\} \end{aligned}$$

Lemma

$T(n) \in \mathcal{O}(3^{n/3}) \subset \mathcal{O}(1.44225^n)$

"Proof" (spot the error).

$$\begin{aligned} T(n) &\leq 1 + s \cdot T(n-s) \\ &\leq 1 + s + s^2 + \cdots + s^{n/s} \\ &= \frac{s^{1+n/s}-1}{s-1} < 2s^{n/s} \qquad (\text{for } s \geq 2) \\ &\in \mathcal{O}(s^{n/s}) \subseteq \mathcal{O}(e^{n/e}) \qquad \square \end{aligned}$$

MIS via Branching

Let $T(n)$ be the maximum number of subproblems considered by the branching algorithm on a graph with n vertices, then (very loosely):

$$T(0) = 1$$

$$\begin{aligned}
T(n) &\leq 1 + \sum_{w \in N[v]} T\Big(n - (d(w)+1)\Big) && \text{for some } v \in V \text{ of minimum degree} \\
&\leq 1 + (d(v)+1) \cdot T\Big(n - (d(v)+1)\Big) && \text{since } T(\cdot) \text{ is nondecreasing} \\
&= 1 + s \cdot T(n-s) && \text{where } s = d(v)+1 \text{ and thus } s \in \{1, \ldots, n\}
\end{aligned}$$

Lemma

$T(n) \in \mathcal{O}(3^{n/3}) \subset \mathcal{O}(1.44225^n)$

"Proof" (spot the error).

$$\begin{aligned}
T(n) &\leq 1 + s \cdot T(n-s) \\
&\leq 1 + s + s^2 + \cdots + s^{n/s} \\
&= \frac{s^{1+n/s}-1}{s-1} < 2s^{n/s} \qquad (\text{for } s \geq 2) \\
&\in \mathcal{O}(s^{n/s}) \subseteq \mathcal{O}(e^{n/e}) \qquad \square
\end{aligned}$$

MIS via Branching

Let $T(n)$ be the maximum number of subproblems considered by the branching algorithm on a graph with n vertices, then (very loosely):

$$T(0) = 1$$

$$T(n) \leq 1 + \sum_{w \in N[v]} T\Big(n - (d(w) + 1)\Big) \qquad \text{for some } v \in V \text{ of minimum degree}$$

$$\leq 1 + (d(v) + 1) \cdot T\Big(n - (d(v) + 1)\Big) \qquad \text{since } T(\cdot) \text{ is nondecreasing}$$

$$= 1 + s \cdot T(n - s) \qquad \text{where } s = d(v) + 1 \text{ and thus } s \in \{1, \dots, n\}$$

Lemma

$T(n) \in \mathcal{O}(3^{n/3}) \subset \mathcal{O}(1.44225^n)$

"Proof" (spot the error).

$$T(n) \leq 1 + s \cdot T(n - s)$$

$$\leq 1 + s + s^2 + \cdots + s^{n/s}$$

$$= \frac{s^{\mathbf{1}+n/s} - 1}{s - 1} < 2s^{n/s} \qquad (\text{for } s \geq 2)$$

$$\in \mathcal{O}(s^{n/s}) \subseteq \mathcal{O}(e^{n/e}) \qquad \square$$

MIS via Branching

Let $T(n)$ be the maximum number of subproblems considered by the branching algorithm on a graph with n vertices, then (very loosely):

$$T(0) = 1$$

$$T(n) \leq 1 + \sum_{w \in N[v]} T\Big(n - (d(w)+1)\Big) \qquad \text{for some } v \in V \text{ of minimum degree}$$

$$\leq 1 + (d(v)+1) \cdot T\Big(n - (d(v)+1)\Big) \qquad \text{since } T(\cdot) \text{ is nondecreasing}$$

$$= 1 + s \cdot T(n-s) \qquad \text{where } s = d(v)+1 \text{ and thus } s \in \{1, \ldots, n\}$$

Lemma

$T(n) \in \mathcal{O}(3^{n/3}) \subset \mathcal{O}(1.44225^n)$

“Proof” (spot the error).

$$T(n) \leq 1 + s \cdot T(n-s)$$

$$\leq 1 + s + s^2 + \cdots + s^{n/s}$$

$$= \frac{s^{1+n/s}-1}{s-1} < 2s^{n/s} \qquad (\text{for } s \geq 2)$$

$$\in \mathcal{O}(s^{n/s}) \subseteq \mathcal{O}(e^{n/e}) \qquad \square$$

MIS via Branching

Let $T(n)$ be the maximum number of subproblems considered by the branching algorithm on a graph with n vertices, then (very loosely):

$$T(0) = 1$$

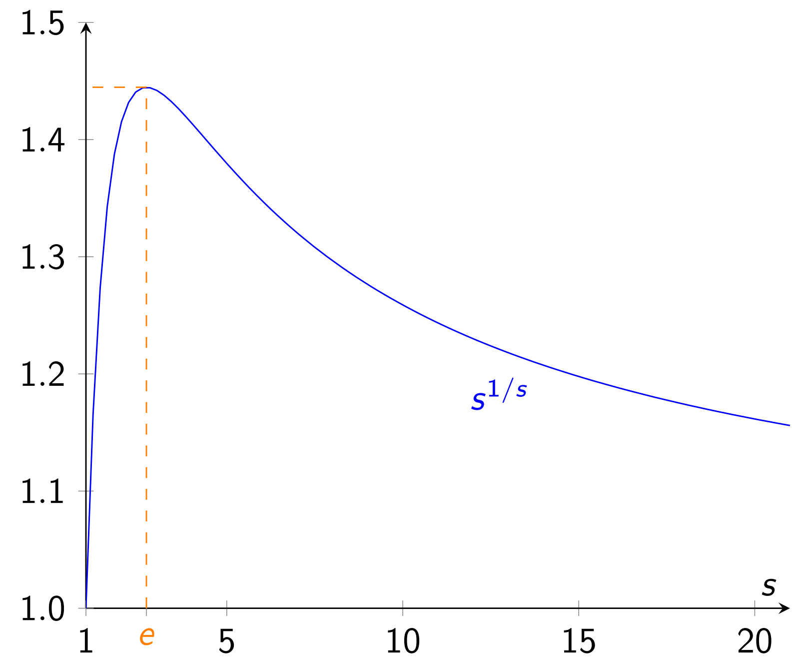
$$T(n) \le 1 + \sum_{w \in N[v]} T\Big(n - (d(w)+1)\Big)$$ for some $v \in V$ of minimum degree

$$\le 1 + (d(v)+1) \cdot T\Big(n - (d(v)+1)\Big)$$ since $T(\cdot)$ is nondecreasing

$$= 1 + s \cdot T(n-s)$$ where $s = d(v)+1$ and thus $s \in \{1, \ldots, n\}$

Lemma

$T(n) \in \mathcal{O}(3^{n/3}) \subset \mathcal{O}(1.44225^n)$



"Proof" (spot the error):

$$T(n) \le 1 + s \cdot T(n-s)$$

$$\le 1 + s + s^2 + \cdots + s^{n/s}$$

$$= \frac{s^{1+n/s}-1}{s-1} < 2s^{n/s} \qquad \text{(for } s \ge 2\text{)}$$

$$\in \mathcal{O}(s^{n/s}) \subseteq \mathcal{O}(e^{n/e}) \qquad \square$$

MIS via Branching

Let $T(n)$ be the maximum number of subproblems considered by the branching algorithm on a graph with n vertices, then (very loosely):

$$T(0) = 1$$

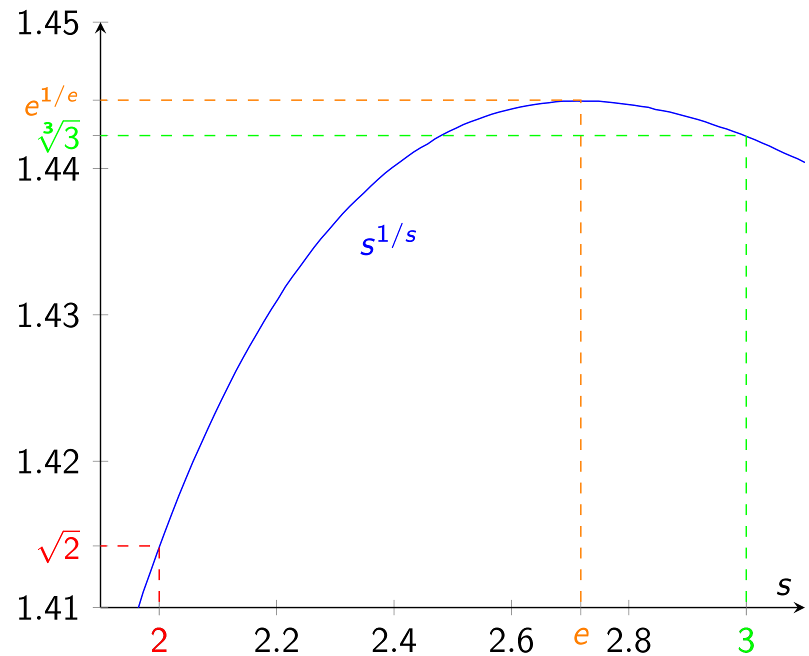
$$\begin{aligned}
T(n) &\le 1 + \sum_{w \in N[v]} T\Big(n - (d(w)+1)\Big) && \text{for some } v \in V \text{ of minimum degree} \\
&\le 1 + (d(v)+1) \cdot T\Big(n - (d(v)+1)\Big) && \text{since } T(\cdot) \text{ is nondecreasing} \\
&= 1 + s \cdot T(n-s) && \text{where } s = d(v)+1 \text{ and thus } s \in \{1, \dots, n\}
\end{aligned}$$

Lemma

$T(n) \in \mathcal{O}(3^{n/3}) \subset \mathcal{O}(1.44225^n)$

"Proof" (spot the error).

$$\begin{aligned}
T(n) &\le 1 + s \cdot T(n-s) \\
&\le 1 + s + s^2 + \dots + s^{n/s} \\
&= \frac{s^{1+n/s}-1}{s-1} < 2s^{n/s} \qquad (\text{for } s \ge 2) \\
&\in \mathcal{O}(s^{n/s}) \subseteq \mathcal{O}(3^{n/3}) \qquad \square
\end{aligned}$$



MIS via Branching

Let $T(n)$ be the maximum number of subproblems considered by the branching algorithm on a graph with n vertices, then (very loosely):

$$T(0) = 1$$

$$T(n) \leq 1 + \sum_{w \in N[v]} T\Big(n - (d(w)+1)\Big) \qquad \text{for some } v \in V \text{ of minimum degree}$$

$$\leq 1 + (d(v)+1) \cdot T\Big(n - (d(v)+1)\Big) \qquad \text{since } T(\cdot) \text{ is nondecreasing}$$

$$= 1 + s \cdot T(n-s) \qquad \text{where } s = d(v)+1 \text{ and thus } s \in \{1, \ldots, n\}$$

It's not the same vertex and it's not the same degree each time

Lemma

$T(n) \in \mathcal{O}(3^{n/3}) \subset \mathcal{O}(1.44225^n)$

"Proof" (Error: s depends on v).

s是不一样的

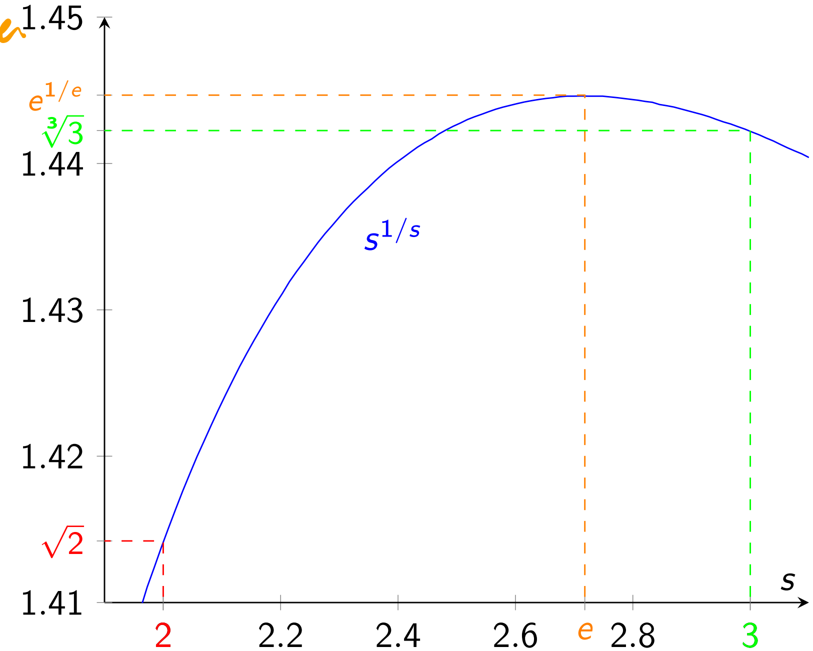
$$T(n) \leq 1 + s \cdot T(n-s)$$

$$\leq 1 + s + s^2 + \cdots + s^{n/s}$$

$$= \frac{s^{1+n/s}-1}{s-1} < 2s^{n/s} \qquad (\text{for } s \geq 2)$$

$$\in \mathcal{O}^*(s^{n/s}) \subseteq \mathcal{O}(3^{n/3}) \qquad \square$$

错误的证法 正确的在作业中



AADS Lecture 9, Part 3

Parameterized problems

"Bar fight prevention" aka k-Vertex Cover

Problem: Bouncer in a small city wants to block people at the door to prevent fights. Assume he knows everyone and knows which pairs of people would fight if they were both let in. Management only allows him to block $\leq k$ of the n people who wants in. Is that enough to prevent fights, and if so, who should be blocked?

Equivalent Problem: Given a graph (V, E) with $n = |V|$ vertices, is there a subset $C \subseteq V$ of size $|C| \leq k$ such that every edge has at least one endpoint in C? Such a set C is called a *k-Vertex Cover* in the graph, and its complement $V \setminus C$ is an *Independent Set* of size $n - k$.

For concreteness in the following, suppose $n = 1000$ and $k = 10$.
Naive 1: Try all 2^n subsets of people. ($2^{1000} \approx 1.07 \cdot 10^{301}$ cases).
Naive 2: Use MIS algorithm. ($2 \cdot 3^{1000/3} - 1 \approx 2.195 \cdot 10^{159}$ cases).
Better 1: Try all $\binom{n}{k}$ subsets of k people. ($\binom{1000}{10} \approx 2.63 \cdot 10^{23}$ cases).

"Bar fight prevention" aka k-Vertex Cover

Problem: Bouncer in a small city wants to block people at the door to prevent fights. Assume he knows everyone and knows which pairs of people would fight if they were both let in. Management only allows him to block $\leq k$ of the n people who wants in. Is that enough to prevent fights, and if so, who should be blocked?

Equivalent Problem: Given a graph (V, E) with $n = |V|$ vertices, is there a subset $C \subseteq V$ of size $|C| \leq k$ such that every edge has at least one endpoint in C? Such a set C is called a *k-Vertex Cover* in the graph, and its complement $V \setminus C$ is an *Independent Set* of size $n - k$.

For concreteness in the following, suppose $n = 1000$ and $k = 10$.
Naive 1: Try all 2^n subsets of people. ($2^{1000} \approx 1.07 \cdot 10^{301}$ cases).
Naive 2: Use MIS algorithm. ($2 \cdot 3^{1000/3} - 1 \approx 2.195 \cdot 10^{159}$ cases).
Better 1: Try all $\binom{n}{k}$ subsets of k people. ($\binom{1000}{10} \approx 2.63 \cdot 10^{23}$ cases).

“Bar fight prevention” aka k-Vertex Cover

Problem: Bouncer in a small city wants to block people at the door to prevent fights. Assume he knows everyone and knows which pairs of people would fight if they were both let in. Management only allows him to block $\leq k$ of the n people who wants in. Is that enough to prevent fights, and if so, who should be blocked?

Equivalent Problem: Given a graph (V, E) with $n = |V|$ vertices, is there a subset $C \subseteq V$ of size $|C| \leq k$ such that every edge has at least one endpoint in C? Such a set C is called a *k-Vertex Cover* in the graph, and its complement $V \setminus C$ is an *Independent Set* of size $n - k$.

For concreteness in the following, suppose $n = 1000$ and $k = 10$.
Naive 1: Try all 2^n subsets of people. ($2^{1000} \approx 1.07 \cdot 10^{301}$ cases).
Naive 2: Use MIS algorithm. ($2 \cdot 3^{1000/3} - 1 \approx 2.195 \cdot 10^{159}$ cases).
Better 1: Try all $\binom{n}{k}$ subsets of k people. ($\binom{1000}{10} \approx 2.63 \cdot 10^{23}$ cases).

"Bar fight prevention" aka k-Vertex Cover

Problem: Bouncer in a small city wants to block people at the door to prevent fights. Assume he knows everyone and knows which pairs of people would fight if they were both let in. Management only allows him to block $\leq k$ of the n people who wants in. Is that enough to prevent fights, and if so, who should be blocked?

Equivalent Problem: Given a graph (V, E) with $n = |V|$ vertices, is there a subset $C \subseteq V$ of size $|C| \leq k$ such that every edge has at least one endpoint in C? Such a set C is called a *k-Vertex Cover* in the graph, and its complement $V \setminus C$ is an *Independent Set* of size $n - k$.

补集

For concreteness in the following, suppose $n = 1000$ and $k = 10$.

Naive 1: Try all 2^n subsets of people. ($2^{1000} \approx 1.07 \cdot 10^{301}$ cases).

Naive 2: Use MIS algorithm. ($2 \cdot 3^{1000/3} - 1 \approx 2.195 \cdot 10^{159}$ cases).

Better 1: Try all $\binom{n}{k}$ subsets of k people. ($\binom{1000}{10} \approx 2.63 \cdot 10^{23}$ cases).

"Bar fight prevention" aka k-Vertex Cover

Problem: Bouncer in a small city wants to block people at the door to prevent fights. Assume he knows everyone and knows which pairs of people would fight if they were both let in. Management only allows him to block $\leq k$ of the n people who wants in. Is that enough to prevent fights, and if so, who should be blocked?

Equivalent Problem: Given a graph (V, E) with $n = |V|$ vertices, is there a subset $C \subseteq V$ of size $|C| \leq k$ such that every edge has at least one endpoint in C? Such a set C is called a *k-Vertex Cover* in the graph, and its complement $V \setminus C$ is an *Independent Set* of size $n - k$.

The minimum vertex cover is a complement of the MIS.

For concreteness in the following, suppose $n = 1000$ and $k = 10$.

Naive 1: Try all 2^n subsets of people. ($2^{1000} \approx 1.07 \cdot 10^{301}$ cases).

Naive 2: Use MIS algorithm. ($2 \cdot 3^{1000/3} - 1 \approx 2.195 \cdot 10^{159}$ cases).

Better 1: Try all $\binom{n}{k}$ subsets of k people. ($\binom{1000}{10} \approx 2.63 \cdot 10^{23}$ cases).

“Bar fight prevention” aka k-Vertex Cover

Problem: Bouncer in a small city wants to block people at the door to prevent fights. Assume he knows everyone and knows which pairs of people would fight if they were both let in. Management only allows him to block $\leq k$ of the n people who wants in. Is that enough to prevent fights, and if so, who should be blocked?

Equivalent Problem: Given a graph (V, E) with $n = |V|$ vertices, is there a subset $C \subseteq V$ of size $|C| \leq k$ such that every edge has at least one endpoint in C? Such a set C is called a *k-Vertex Cover* in the graph, and its complement $V \setminus C$ is an *Independent Set* of size $n - k$.

精确求MVC，看size与k的大小

For concreteness in the following, suppose $n = 1000$ and $k = 10$.
Naive 1: Try all 2^n subsets of people. $(2^{1000} \approx 1.07 \cdot 10^{301}$ cases).
Naive 2: Use MIS algorithm. $(2 \cdot 3^{1000/3} - 1 \approx 2.195 \cdot 10^{159}$ cases).
Better 1: Try all $\binom{n}{k}$ subsets of k people. $(\binom{1000}{10} \approx 2.63 \cdot 10^{23}$ cases).

"Bar fight prevention" aka k-Vertex Cover

Problem: Bouncer in a small city wants to block people at the door to prevent fights. Assume he knows everyone and knows which pairs of people would fight if they were both let in. Management only allows him to block $\leq k$ of the n people who wants in. Is that enough to prevent fights, and if so, who should be blocked?

Equivalent Problem: Given a graph (V, E) with $n = |V|$ vertices, is there a subset $C \subseteq V$ of size $|C| \leq k$ such that every edge has at least one endpoint in C? Such a set C is called a *k-Vertex Cover* in the graph, and its complement $V \setminus C$ is an *Independent Set* of size $n - k$.

For concreteness in the following, suppose $n = 1000$ and $k = 10$.
Naive 1: Try all 2^n subsets of people. ($2^{1000} \approx 1.07 \cdot 10^{301}$ cases).
Naive 2: Use MIS algorithm. ($2 \cdot 3^{1000/3} - 1 \approx 2.195 \cdot 10^{159}$ cases).
Better 1: Try all $\underbrace{\binom{n}{k}}_{\text{possible}}$ subsets of k people. ($\binom{1000}{10} \approx 2.63 \cdot 10^{23}$ cases).

"Bar fight prevention" via Kernelization

Consider the conflict graph $G = (V, E)$.

Idea: If $d(v) = 0$: let v in and drop v from G.
Why?

Idea: If $d(v) \geq k + 1$: reject v, drop v from G, and decrease k.
Why?

Note: If $d(v) \leq k$ for all v and $|E| > k^2$, there is no solution.
Why?

Better 2: The above ideas reduce to a graph H with $|V| \leq 2k^2$ vertices.
Why?

Now try all $\binom{2k^2}{k}$ subsets of k people. $\left(\binom{2 \cdot 10^2}{10} \approx 2.24 \cdot 10^{16}\right)$.

Idea: If $N[v] = \{v, w\}$: let v in, reject w, drop $N[v]$ from G, and decrease k.
Why?

Better 3: The above ideas reduce to a graph H with $|V| \leq k^2$ vertices.
Why?

Now try all $\binom{k^2}{k}$ subsets of k people. $\left(\binom{10^2}{10} \approx 1.73 \cdot 10^{13}\right)$.

"Bar fight prevention" via Kernelization

Consider the conflict graph $G = (V, E)$.

Idea: If $d(v) = 0$: let v in and drop v from G. 让不会与任何人打架的人进来
Why? Safe because no conflicts.

Idea: If $d(v) \geq k + 1$: reject v, drop v from G, and decrease k.
Why?

Note: If $d(v) \leq k$ for all v and $|E| > k^2$, there is no solution.
Why?

Better 2: The above ideas reduce to a graph H with $|V| \leq 2k^2$ vertices.
Why?

Now try all $\binom{2k^2}{k}$ subsets of k people. $\left(\binom{2 \cdot 10^2}{10} \approx 2.24 \cdot 10^{16}\right)$.

Idea: If $N[v] = \{v, w\}$: let v in, reject w, drop $N[v]$ from G, and decrease k.
Why?

Better 3: The above ideas reduce to a graph H with $|V| \leq k^2$ vertices.
Why?

Now try all $\binom{k^2}{k}$ subsets of k people. $\left(\binom{10^2}{10} \approx 1.73 \cdot 10^{13}\right)$.

“Bar fight prevention” via Kernelization

Consider the conflict graph $G = (V, E)$.

Idea: If $d(v) = 0$: let v in and drop v from G.
Why? Safe because no conflicts.

Idea: If $d(v) \geq k + 1$: reject v, drop v from G, and decrease k.
Why? 至少会和k+1打架的人，把他reject掉

Note: If $d(v) \leq k$ for all v and $|E| > k^2$, there is no solution.
Why?

Better 2: The above ideas reduce to a graph H with $|V| \leq 2k^2$ vertices.
Why?

Now try all $\binom{2k^2}{k}$ subsets of k people. $\left(\binom{2 \cdot 10^2}{10} \approx 2.24 \cdot 10^{16}\right)$.

Idea: If $N[v] = \{v, w\}$: let v in, reject w, drop $N[v]$ from G, and decrease k.
Why?

Better 3: The above ideas reduce to a graph H with $|V| \leq k^2$ vertices.
Why?

Now try all $\binom{k^2}{k}$ subsets of k people. $\left(\binom{10^2}{10} \approx 1.73 \cdot 10^{13}\right)$.

"Bar fight prevention" via Kernelization

Consider the conflict graph $G = (V, E)$.

Idea: If $d(v) = 0$: let v in and drop v from G.
Why? Safe because no conflicts.

Idea: If $d(v) \geq k + 1$: reject v, drop v from G, and decrease k.
Why? Not rejecting v means rejecting $d(v) > k$ people. 如果不把他拒绝掉，就得拒绝所有他的邻居 more than k.

Note: If $d(v) \leq k$ for all v and $|E| > k^2$, there is no solution.
Why?

Better 2: The above ideas reduce to a graph H with $|V| \leq 2k^2$ vertices.
Why?

Now try all $\binom{2k^2}{k}$ subsets of k people. $\left(\binom{2 \cdot 10^2}{10} \approx 2.24 \cdot 10^{16}\right)$.

Idea: If $N[v] = \{v, w\}$: let v in, reject w, drop $N[v]$ from G, and decrease k.
Why?

Better 3: The above ideas reduce to a graph H with $|V| \leq k^2$ vertices.
Why?

Now try all $\binom{k^2}{k}$ subsets of k people. $\left(\binom{10^2}{10} \approx 1.73 \cdot 10^{13}\right)$.

“Bar fight prevention” via Kernelization

Consider the conflict graph $G = (V, E)$.

Idea: If $d(v) = 0$: let v in and drop v from G.
Why? Safe because no conflicts.

Idea: If $d(v) \geq k + 1$: reject v, drop v from G, and decrease k.
Why? Not rejecting v means rejecting $d(v) > k$ people.

Note: If $d(v) \leq k$ for all v and $|E| > k^2$, there is no solution.
Why?

Better 2: The above ideas reduce to a graph H with $|V| \leq 2k^2$ vertices.
Why?

Now try all $\binom{2k^2}{k}$ subsets of k people. $\left(\binom{2 \cdot 10^2}{10} \approx 2.24 \cdot 10^{16}\right)$.

Idea: If $N[v] = \{v, w\}$: let v in, reject w, drop $N[v]$ from G, and decrease k.
Why?

Better 3: The above ideas reduce to a graph H with $|V| \leq k^2$ vertices.
Why?

Now try all $\binom{k^2}{k}$ subsets of k people. $\left(\binom{10^2}{10} \approx 1.73 \cdot 10^{13}\right)$.

“Bar fight prevention” via Kernelization

Consider the conflict graph $G = (V, E)$.

Idea: If $d(v) = 0$: let v in and drop v from G.
Why? Safe because no conflicts.

Idea: If $d(v) \geq k + 1$: reject v, drop v from G, and decrease k.
Why? Not rejecting v means rejecting $d(v) > k$ people.

Note: If $d(v) \leq k$ for all v and $|E| > k^2$, there is no solution.
Why? Each rejection resolves at most k conflicts. 因为每个人只参与k个冲突，拒绝这个人只能解决k个冲突，所以拒绝k个人无法解决超过k^2的冲突

Better 2: The above ideas reduce to a graph H with $|V| \leq 2k^2$ vertices.
Why?

Now try all $\binom{2k^2}{k}$ subsets of k people. $\left(\binom{2\cdot 10^2}{10} \approx 2.24 \cdot 10^{16}\right)$.

Idea: If $N[v] = \{v, w\}$: let v in, reject w, drop $N[v]$ from G, and decrease k.
Why?

Better 3: The above ideas reduce to a graph H with $|V| \leq k^2$ vertices.
Why?

Now try all $\binom{k^2}{k}$ subsets of k people. $\left(\binom{10^2}{10} \approx 1.73 \cdot 10^{13}\right)$.

"Bar fight prevention" via Kernelization

Consider the conflict graph $G = (V, E)$.

Idea: If $d(v) = 0$: let v in and drop v from G.
Why? Safe because no conflicts.

Idea: If $d(v) \geq k + 1$: reject v, drop v from G, and decrease k.
Why? Not rejecting v means rejecting $d(v) > k$ people.

Note: If $d(v) \leq k$ for all v and $|E| > k^2$, there is no solution.
Why? Each rejection resolves at most k conflicts.

Better 2: The above ideas reduce to a graph H with $|V| \leq 2k^2$ vertices.
Why?

Now try all $\binom{2k^2}{k}$ subsets of k people. $\left(\binom{2\cdot 10^2}{10} \approx 2.24 \cdot 10^{16}\right)$.

Idea: If $N[v] = \{v, w\}$: let v in, reject w, drop $N[v]$ from G, and decrease k.
Why?

Better 3: The above ideas reduce to a graph H with $|V| \leq k^2$ vertices.
Why?

Now try all $\binom{k^2}{k}$ subsets of k people. $\left(\binom{10^2}{10} \approx 1.73 \cdot 10^{13}\right)$.

"Bar fight prevention" via Kernelization

Consider the conflict graph $G = (V, E)$.

Idea: If $d(v) = 0$: let v in and drop v from G.
Why? Safe because no conflicts.

Idea: If $d(v) \geq k + 1$: reject v, drop v from G, and decrease k.
Why? Not rejecting v means rejecting $d(v) > k$ people.

Note: If $d(v) \leq k$ for all v and $|E| > k^2$, there is no solution.
Why? Each rejection resolves at most k conflicts.

Better 2: The above ideas reduce to a graph H with $|V| \leq 2k^2$ vertices.
Why? $|V| = \sum_{v \in V} 1 \leq \sum_{v \in V} d(v) = 2|E| \leq 2k^2$

Now try all $\binom{2k^2}{k}$ subsets of k people. $\left(\binom{2 \cdot 10^2}{10} \approx 2.24 \cdot 10^{16}\right)$.

Idea: If $N[v] = \{v, w\}$: let v in, reject w, drop $N[v]$ from G, and decrease k.
Why?

Better 3: The above ideas reduce to a graph H with $|V| \leq k^2$ vertices.
Why?

Now try all $\binom{k^2}{k}$ subsets of k people. $\left(\binom{10^2}{10} \approx 1.73 \cdot 10^{13}\right)$.

"Bar fight prevention" via Kernelization

Consider the conflict graph $G = (V, E)$.

Idea: If $d(v) = 0$: let v in and drop v from G.
Why? Safe because no conflicts.

Idea: If $d(v) \geq k + 1$: reject v, drop v from G, and decrease k.
Why? Not rejecting v means rejecting $d(v) > k$ people.

Note: If $d(v) \leq k$ for all v and $|E| > k^2$, there is no solution.
Why? Each rejection resolves at most k conflicts.

Better 2: The above ideas reduce to a graph H with $|V| \leq 2k^2$ vertices.
Why? $|V| = \sum_{v \in V} 1 \leq \sum_{v \in V} d(v) = 2|E| \leq 2k^2$
Now try all $\binom{2k^2}{k}$ subsets of k people. $\left(\binom{2 \cdot 10^2}{10} \approx 2.24 \cdot 10^{16}\right)$.

Idea: If $N[v] = \{v, w\}$: let v in, reject w, drop $N[v]$ from G, and decrease k.
Why?

Better 3: The above ideas reduce to a graph H with $|V| \leq k^2$ vertices.
Why?
Now try all $\binom{k^2}{k}$ subsets of k people. $\left(\binom{10^2}{10} \approx 1.73 \cdot 10^{13}\right)$.

"Bar fight prevention" via Kernelization

Consider the conflict graph $G = (V, E)$.

Idea: If $d(v) = 0$: let v in and drop v from G.
Why? Safe because no conflicts.

Idea: If $d(v) \geq k+1$: reject v, drop v from G, and decrease k.
Why? Not rejecting v means rejecting $d(v) > k$ people.

Note: If $d(v) \leq k$ for all v and $|E| > k^2$, there is no solution.
Why? Each rejection resolves at most k conflicts.

Better 2: The above ideas reduce to a graph H with $|V| \leq 2k^2$ vertices.
Why? $|V| = \sum_{v \in V} 1 \leq \sum_{v \in V} d(v) = 2|E| \leq 2k^2$ 一大内部解决
Now try all $\binom{2k^2}{k}$ subsets of k people. → 拒绝 $\left(\binom{2 \cdot 10^2}{10} \approx 2.24 \cdot 10^{16}\right)$.

Idea: If $N[v] = \{v, w\}$: let v in, reject w, drop $N[v]$ from G, and decrease k. 恰好只有一个冲突的人
Why?

Better 3: The above ideas reduce to a graph H with $|V| \leq k^2$ vertices.
Why?
Now try all $\binom{k^2}{k}$ subsets of k people. $\left(\binom{10^2}{10} \approx 1.73 \cdot 10^{13}\right)$.

"Bar fight prevention" via Kernelization

Consider the conflict graph $G = (V, E)$.

Idea: If $d(v) = 0$: let v in and drop v from G.
Why? Safe because no conflicts.

Idea: If $d(v) \geq k + 1$: reject v, drop v from G, and decrease k.
Why? Not rejecting v means rejecting $d(v) > k$ people.

Note: If $d(v) \leq k$ for all v and $|E| > k^2$, there is no solution.
Why? Each rejection resolves at most k conflicts.

Better 2: The above ideas reduce to a graph H with $|V| \leq 2k^2$ vertices.
Why? $|V| = \sum_{v \in V} 1 \leq \sum_{v \in V} d(v) = 2|E| \leq 2k^2$
Now try all $\binom{2k^2}{k}$ subsets of k people. $\left(\binom{2 \cdot 10^2}{10} \approx 2.24 \cdot 10^{16}\right)$.

Idea: If $N[v] = \{v, w\}$: let v in, reject w, drop $N[v]$ from G, and decrease k.
两者之一放进都一样，不会变得更糟，
Why? In any solution that lets w in, we can let v in instead. Never worse.

Better 3: The above ideas reduce to a graph H with $|V| \leq k^2$ vertices.
Why?
Now try all $\binom{k^2}{k}$ subsets of k people. $\left(\binom{10^2}{10} \approx 1.73 \cdot 10^{13}\right)$.

“Bar fight prevention” via Kernelization

Consider the conflict graph $G = (V, E)$.

Idea: If $d(v) = 0$: let v in and drop v from G.
Why? Safe because no conflicts.

Idea: If $d(v) \geq k + 1$: reject v, drop v from G, and decrease k.
Why? Not rejecting v means rejecting $d(v) > k$ people.

Note: If $d(v) \leq k$ for all v and $|E| > k^2$, there is no solution.
Why? Each rejection resolves at most k conflicts.

Better 2: The above ideas reduce to a graph H with $|V| \leq 2k^2$ vertices.
Why? $|V| = \sum_{v \in V} 1 \leq \sum_{v \in V} d(v) = 2|E| \leq 2k^2$
Now try all $\binom{2k^2}{k}$ subsets of k people. $\left(\binom{2 \cdot 10^2}{10} \approx 2.24 \cdot 10^{16}\right)$.

Idea: If $N[v] = \{v, w\}$: let v in, reject w, drop $N[v]$ from G, and decrease k.
Why? In any solution that lets w in, we can let v in instead. Never worse.

Better 3: The above ideas reduce to a graph H with $|V| \leq k^2$ vertices.
Why?
Now try all $\binom{k^2}{k}$ subsets of k people. $\left(\binom{10^2}{10} \approx 1.73 \cdot 10^{13}\right)$.

“Bar fight prevention” via Kernelization

Consider the conflict graph $G = (V, E)$.

Idea: If $d(v) = 0$: let v in and drop v from G.
Why? Safe because no conflicts.

Idea: If $d(v) \geq k + 1$: reject v, drop v from G, and decrease k.
Why? Not rejecting v means rejecting $d(v) > k$ people.

Note: If $d(v) \leq k$ for all v and $|E| > k^2$, there is no solution.
Why? Each rejection resolves at most k conflicts.

Better 2: The above ideas reduce to a graph H with $|V| \leq 2k^2$ vertices.
Why? $|V| = \sum_{v \in V} 1 \leq \sum_{v \in V} d(v) = 2|E| \leq 2k^2$
Now try all $\binom{2k^2}{k}$ subsets of k people. $\left(\binom{2 \cdot 10^2}{10} \approx 2.24 \cdot 10^{16}\right)$.

Idea: If $N[v] = \{v, w\}$: let v in, reject w, drop $N[v]$ from G, and decrease k.
Why? In any solution that lets w in, we can let v in instead. Never worse.

Better 3: The above ideas reduce to a graph H with $|V| \leq k^2$ vertices.
Why? $|V| = \sum_{v \in V} 1 = \frac{1}{2} \sum_{v \in V} 2 \leq \frac{1}{2} \sum_{v \in V} d(v) = |E| \leq k^2$
Now try all $\binom{k^2}{k}$ subsets of k people. $\left(\binom{10^2}{10} \approx 1.73 \cdot 10^{13}\right)$.

"Bar fight prevention" via Kernelization

Consider the conflict graph $G = (V, E)$.

Idea: If $d(v) = 0$: let v in and drop v from G.
Why? Safe because no conflicts.

Idea: If $d(v) \geq k + 1$: reject v, drop v from G, and decrease k.
Why? Not rejecting v means rejecting $d(v) > k$ people.

Note: If $d(v) \leq k$ for all v and $|E| > k^2$, there is no solution.
Why? Each rejection resolves at most k conflicts.

Better 2: The above ideas reduce to a graph H with $|V| \leq 2k^2$ vertices.
Why? $|V| = \sum_{v \in V} 1 \leq \sum_{v \in V} d(v) = 2|E| \leq 2k^2$
Now try all $\binom{2k^2}{k}$ subsets of k people. $\left(\binom{2 \cdot 10^2}{10} \approx 2.24 \cdot 10^{16}\right)$.

Idea: If $N[v] = \{v, w\}$: let v in, reject w, drop $N[v]$ from G, and decrease k.
Why? In any solution that lets w in, we can let v in instead. Never worse.

Better 3: The above ideas reduce to a graph H with $|V| \leq k^2$ vertices.
Why? $|V| = \sum_{v \in V} 1 = \frac{1}{2} \sum_{v \in V} 2 \leq \frac{1}{2} \sum_{v \in V} d(v) = |E| \leq k^2$
Now try all $\binom{k^2}{k}$ subsets of k people. $\left(\binom{10^2}{10} \approx 1.73 \cdot 10^{13}\right)$.

“Bar fight prevention” via Kernelization

```
function BarFightPrevention($k$, $G$)
    $k', H, C \leftarrow$ BFP-Kernel($k$, $G$)
    if $H$ has $\leq (k')^2$ edges and BFP-Brute-Force($k'$, $H$) returns a solution $C'$ then
        return $C \cup C'$
    return "No solution"

function BFP-Kernel($k$, $G$)
    $k' \leftarrow k,\ H \leftarrow G,\ C \leftarrow \emptyset$
    loop
        if Some $v$ has $d(v) = 0$ then
            $H \leftarrow H \setminus \{v\}$
        elseif Some $v$ has $d(v) \geq k' + 1$ then
            $H \leftarrow H \setminus \{v\},\ C \leftarrow C \cup \{v\},\ k' \leftarrow k' - 1$
        elseif Some $v$ has $N[v] = \{v, w\}$ for some $w$ then
            $H \leftarrow H \setminus N[v],\ C \leftarrow C \cup \{w\},\ k' \leftarrow k' - 1$
        else
            return $k', H, C$

function BFP-Brute-Force($k$, $G = (V, E)$)
    for every subset $C \subseteq V$ of size $k$ do
        if $C$ is a vertex cover of $G$ then
            return $C$
    return "No solution"
```

Kernelization

The subgraph H we reduced to before brute-forcing is called a *Kernel* for the Bar Fight Prevention problem, and the process of finding such a kernel is called *Kernelization*.

remove all the easy parts

reduce the hard core part of the instance

The general idea is to use the parameter k to quickly reduce to a smaller subproblem of the same type, whose size ideally depends only on k and not on n. For the bar fight prevention problem we have just shown that:

- If there is a solution for a given k and a given graph G with n vertices and m edges, then we can find a kernel H with at most k^2 vertices.
- Furthermore, such a kernel can be found in $\mathcal{O}(m+n)$ time, and checking if a given subset of size at most k is a solution can be done in $\mathcal{O}(k^2)$ time.
- Thus, for any fixed k, the total running time of this algorithm is $\mathcal{O}(m+n+\binom{k^2}{k}k^2) \subseteq \mathcal{O}(m+n+(ke)^{2k+2}) = \mathcal{O}_k(m+n)$.

Kernelization

The subgraph H we reduced to before brute-forcing is called a *Kernel* for the Bar Fight Prevention problem, and the process of finding such a kernel is called *Kernelization*.

The general idea is to use the parameter k to quickly reduce to a smaller subproblem of the same type, whose size ideally depends only on k and not on n. For the bar fight prevention problem we have just shown that:

- If there is a solution for a given k and a given graph G with n vertices and m edges, then we can find a kernel H with at most k^2 vertices.
- Furthermore, such a kernel can be found in $\mathcal{O}(m+n)$ time, and checking if a given subset of size at most k is a solution can be done in $\mathcal{O}(k^2)$ time.
- Thus, for any fixed k, the total running time of this algorithm is $\mathcal{O}(m+n+\binom{k^2}{k}k^2) \subseteq \mathcal{O}(m+n+(ke)^{2k+2}) = \mathcal{O}_k(m+n)$.

A constant but the constantes within the big O notation depend on k.

"Bar fight prevention" via Bounded Search Tree

Note: For each edge $(u, v) \in E$, in conflict graph. at least one of u, v must be rejected.

Idea: Pick arbitrary edge (u, v), and recursively try with u rejected and with v rejected.

```
function BFP-Bounded-Search(k, G)
    if G has an no edges then
        return ∅
    if k > 0 then
        Let (u, v) be an arbitrary edge of G
        for w ∈ {u, v} do
            if BFP-Bounded-Search(k − 1, G \ {w}) returns a solution C then
                return C ∪ {w}
    return "No solution"
```

This recursive procedure has depth at most k.

Thus the total number of subproblems considered at most 2^k.

If we start by rejecting all vertices of degree $d(v) \geq k+1$ (like in the kernelization approach), the resulting graph has at most $|E| = \frac{1}{2}\sum_{v \in V} d(v) \leq \frac{1}{2}nk$ edges, so constructing each subproblem can be done in $\mathcal{O}(nk)$ time.

The total running time is then $\mathcal{O}(m + nk \cdot 2^k)$. $(1000 \cdot 10 \cdot 2^{10} \approx 10^7)$

Part of Assignment 5 asks you to improve this.

"Bar fight prevention" via Bounded Search Tree

Note: For each edge $(u, v) \in E$, at least one of u, v must be rejected.
Idea: Pick arbitrary edge (u, v), and recursively try with u rejected and with v rejected.

function BFP-Bounded-Search(k, G)
    if G has an no edges **then**
        return $\emptyset$
    if $k > 0$ **then**
        Let (u, v) be an arbitrary edge of G
        for $w \in \{u, v\}$ **do**
            if BFP-Bounded-Search($k-1$, $G \setminus \{w\}$) returns a solution C **then**
                return $C \cup \{w\}$
    return "No solution"

This recursive procedure has depth at most k.
Thus the total number of subproblems considered at most 2^k.
If we start by rejecting all vertices of degree $d(v) \geq k+1$ (like in the kernelization approach), the resulting graph has at most $|E| = \frac{1}{2}\sum_{v \in V} d(v) \leq \frac{1}{2}nk$ edges, so constructing each subproblem can be done in $\mathcal{O}(nk)$ time.
The total running time is then $\mathcal{O}(m + nk \cdot 2^k)$. $\quad (1000 \cdot 10 \cdot 2^{10} \approx 10^7)$
Part of Assignment 5 asks you to improve this.

"Bar fight prevention" via Bounded Search Tree

$k-1$

Note: For each edge $(u, v) \in E$, at least one of u, v must be rejected.
Idea: Pick arbitrary edge (u, v), and recursively try with u rejected and with v rejected.

```
function BFP-Bounded-Search(k, G)
    if G has an no edges then
        return ∅
    if k > 0 then
        Let (u, v) be an arbitrary edge of G
        for w ∈ {u, v} do
            if BFP-Bounded-Search(k − 1, G \ {w}) returns a solution C then
                return C ∪ {w}
    return "No solution"
```

This recursive procedure has depth at most k.
Thus the total number of subproblems considered at most 2^k.
If we start by rejecting all vertices of degree $d(v) \geq k+1$ (like in the kernelization approach), the resulting graph has at most $|E| = \frac{1}{2}\sum_{v \in V} d(v) \leq \frac{1}{2}nk$ edges, so constructing each subproblem can be done in $\mathcal{O}(nk)$ time.
The total running time is then $\mathcal{O}(m + nk \cdot 2^k)$. $(1000 \cdot 10 \cdot 2^{10} \approx 10^7)$
Part of Assignment 5 asks you to improve this.

“Bar fight prevention” via Bounded Search Tree

Note: For each edge $(u, v) \in E$, at least one of u, v must be rejected.
Idea: Pick arbitrary edge (u, v), and recursively try with u rejected and with v rejected.

```
function BFP-Bounded-Search(k, G)
    if G has an no edges then
        return ∅
    if k > 0 then
        Let (u, v) be an arbitrary edge of G
        for w ∈ {u, v} do
            if BFP-Bounded-Search(k − 1, G \ {w}) returns a solution C then
                return C ∪ {w}
    return “No solution”
```

This recursive procedure has depth at most k.
Thus the total number of subproblems considered at most 2^k.
If we start by rejecting all vertices of degree $d(v) \geq k + 1$ (like in the kernelization approach), the resulting graph has at most $|E| = \frac{1}{2}\sum_{v \in V} d(v) \leq \frac{1}{2}nk$ edges, so constructing each subproblem can be done in $\mathcal{O}(nk)$ time.
The total running time is then $\mathcal{O}(m + nk \cdot 2^k)$. $(1000 \cdot 10 \cdot 2^{10} \approx 10^7)$
Part of Assignment 5 asks you to improve this.

"Bar fight prevention" via Bounded Search Tree

Note: For each edge $(u, v) \in E$, at least one of u, v must be rejected.
Idea: Pick arbitrary edge (u, v), and recursively try with u rejected and with v rejected.

function BFP-Bounded-Search(k, G)
    if G has an no edges **then**
        return $\emptyset$
    if $k > 0$ **then**
        Let (u, v) be an arbitrary edge of G
        for $w \in \{u, v\}$ **do**
            if BFP-Bounded-Search($k - 1, G \setminus \{w\}$) returns a solution C **then**
                return $C \cup \{w\}$
    return "No solution"

往两个端点找
∴最多 次

This recursive procedure has depth at most k.
Thus the total number of subproblems considered at most 2^k.
If we start by rejecting all vertices of degree $d(v) \geq k + 1$ (like in the kernelization approach), the resulting graph has at most $|E| = \frac{1}{2} \sum_{v \in V} d(v) \leq \frac{1}{2} nk$ edges, so constructing each subproblem can be done in $\mathcal{O}(nk)$ time.
The total running time is then $\mathcal{O}(m + nk \cdot 2^k)$. $(1000 \cdot 10 \cdot 2^{10} \approx 10^7)$
Part of Assignment 5 asks you to improve this.

"Bar fight prevention" via Bounded Search Tree

Note: For each edge $(u, v) \in E$, at least one of u, v must be rejected.
Idea: Pick arbitrary edge (u, v), and recursively try with u rejected and with v rejected.

```
function BFP-Bounded-Search(k, G)
    if G has an no edges then
        return ∅
    if k > 0 then
        Let (u, v) be an arbitrary edge of G
        for w ∈ {u, v} do
            if BFP-Bounded-Search(k − 1, G \ {w}) returns a solution C then
                return C ∪ {w}
    return "No solution"
```

This recursive procedure has depth at most k.
Thus the total number of subproblems considered at most 2^k.
If we start by rejecting all vertices of degree $d(v) \geq k + 1$ (like in the kernelization approach), the resulting graph has at most $|E| = \frac{1}{2} \sum_{v \in V} d(v) \leq \frac{1}{2} nk$ edges, so constructing each subproblem can be done in $\mathcal{O}(nk)$ time.
The total running time is then $\mathcal{O}(m + nk \cdot 2^k)$. $(1000 \cdot 10 \cdot 2^{10} \approx 10^7)$
Part of Assignment 5 asks you to improve this.

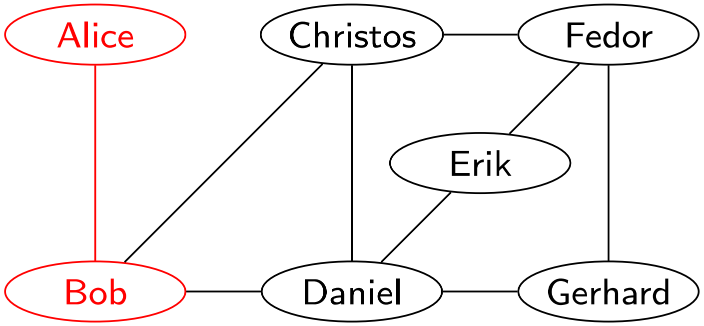
"Bar fight prevention" via Bounded Search Tree

Note: For each edge $(u, v) \in E$, at least one of u, v must be rejected.
Idea: Pick arbitrary edge (u, v), and recursively try with u rejected and with v rejected.

function BFP-Bounded-Search(k, G)
    if G has an no edges **then**
        return $\emptyset$
    if $k > 0$ **then**
        Let (u, v) be an arbitrary edge of G
        for $w \in \{u, v\}$ **do**
            if BFP-Bounded-Search$(k - 1, G \setminus \{w\})$ returns a solution C **then**
                return $C \cup \{w\}$
    return "No solution"

This recursive procedure has depth at most k.
Thus the total number of subproblems considered at most 2^k.
If we start by rejecting all vertices of degree $d(v) \geq k + 1$ (like in the kernelization approach), the resulting graph has at most $|E| = \frac{1}{2} \sum_{v \in V} d(v) \leq \frac{1}{2} nk$ edges, so constructing each subproblem can be done in $\mathcal{O}(nk)$ time.
The total running time is then $\mathcal{O}(m + nk \cdot 2^k)$. $\quad (1000 \cdot 10 \cdot 2^{10} \approx 10^7)$
Part of Assignment 5 asks you to improve this.

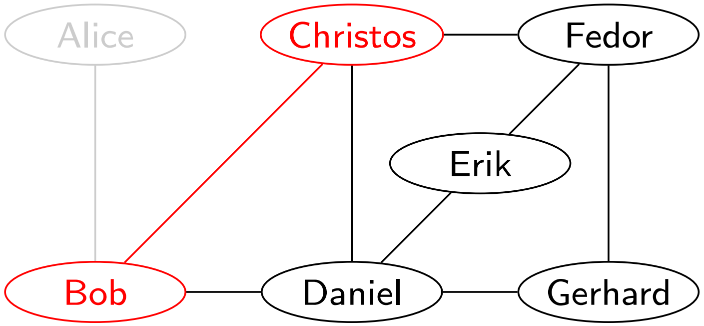
"Bar fight prevention" via Bounded Search Tree



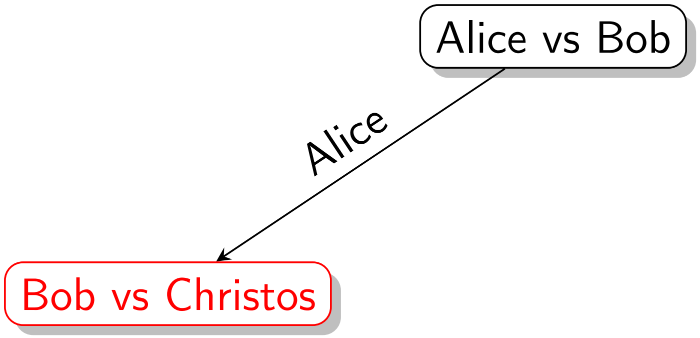
$k = 3$

Alice vs Bob

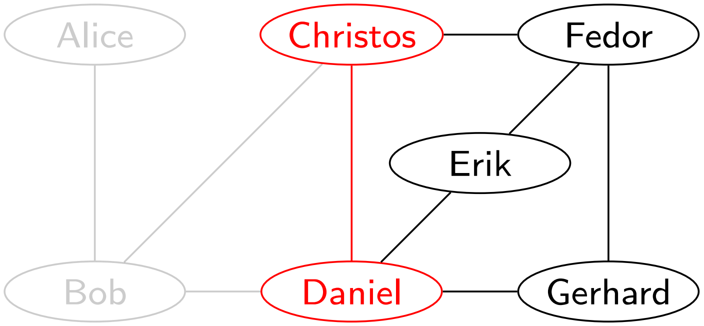
“Bar fight prevention” via Bounded Search Tree



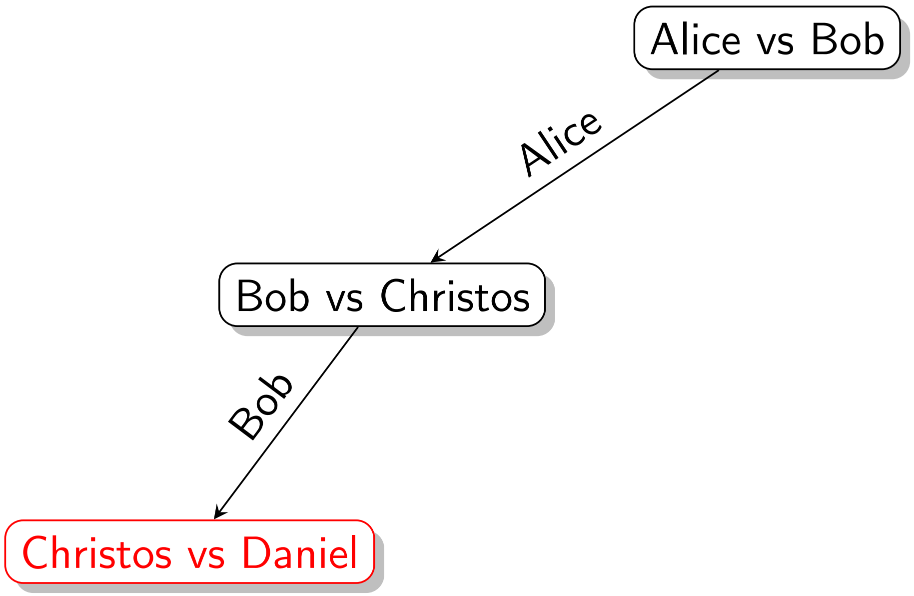
$k = 2$



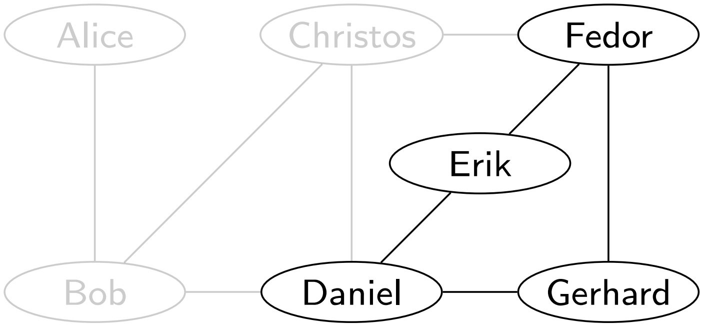
"Bar fight prevention" via Bounded Search Tree



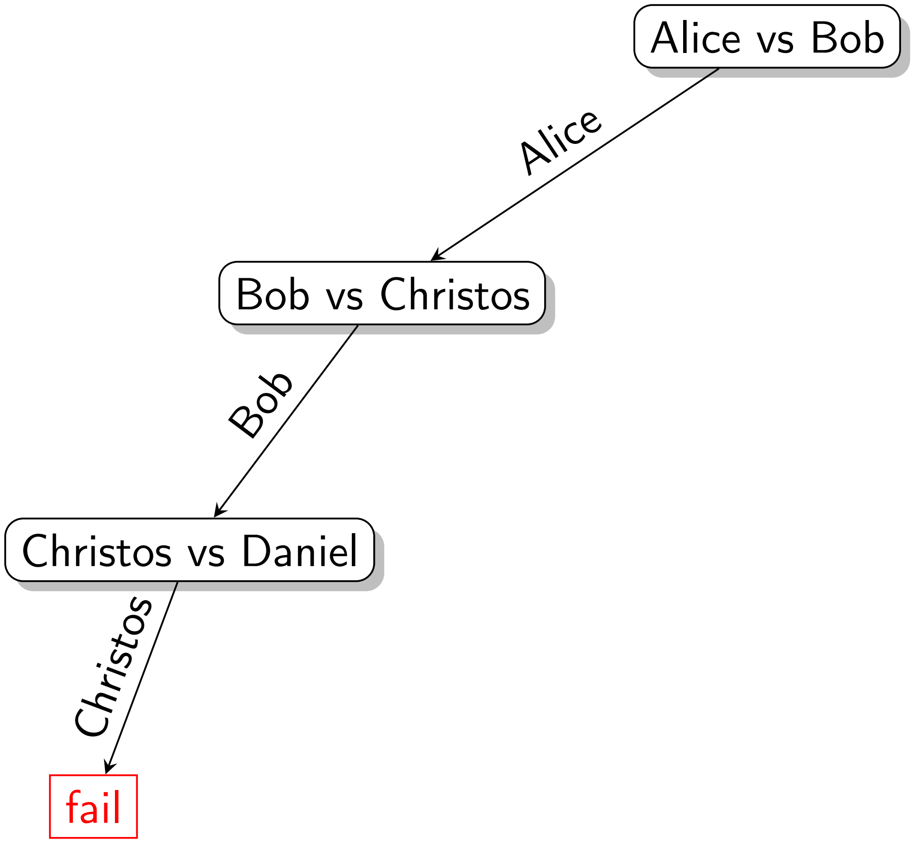
$k = 1$



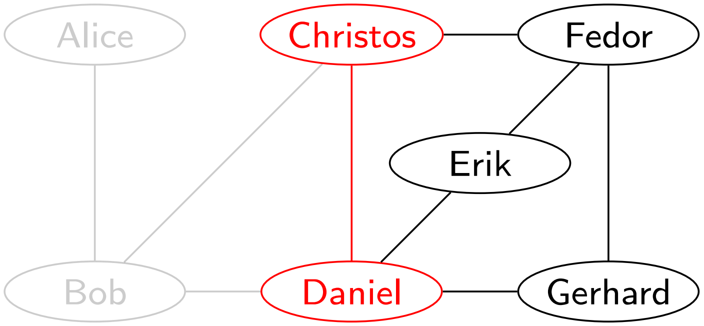
“Bar fight prevention” via Bounded Search Tree



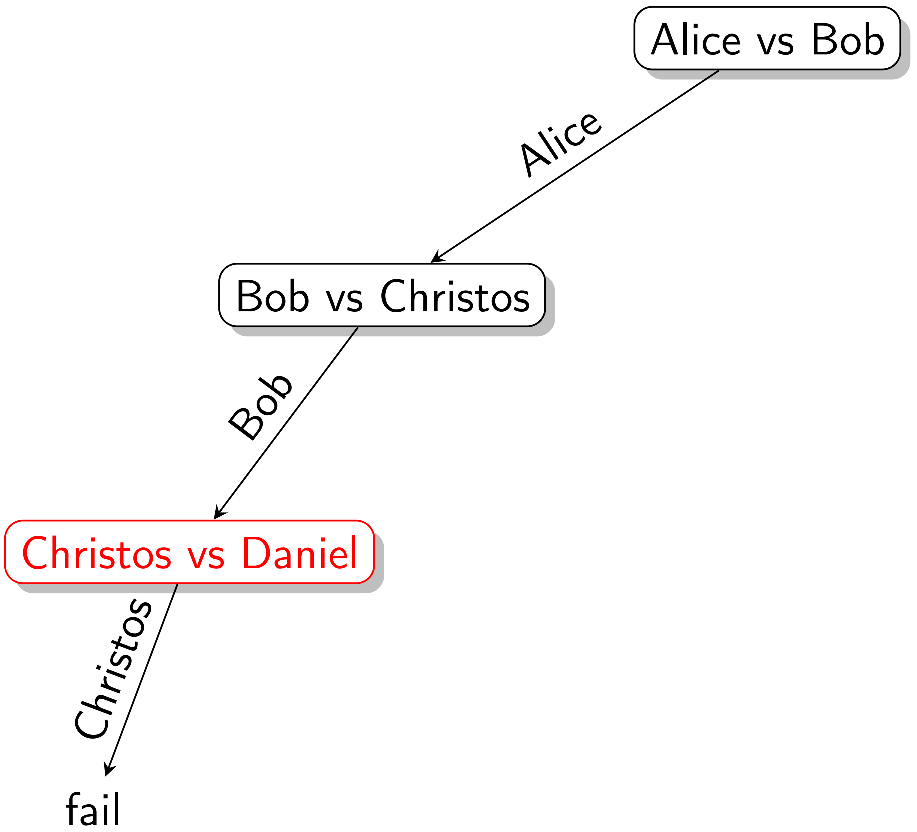
$k = 0$



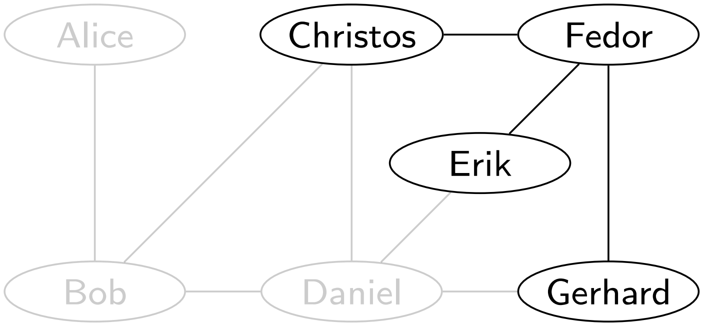
"Bar fight prevention" via Bounded Search Tree



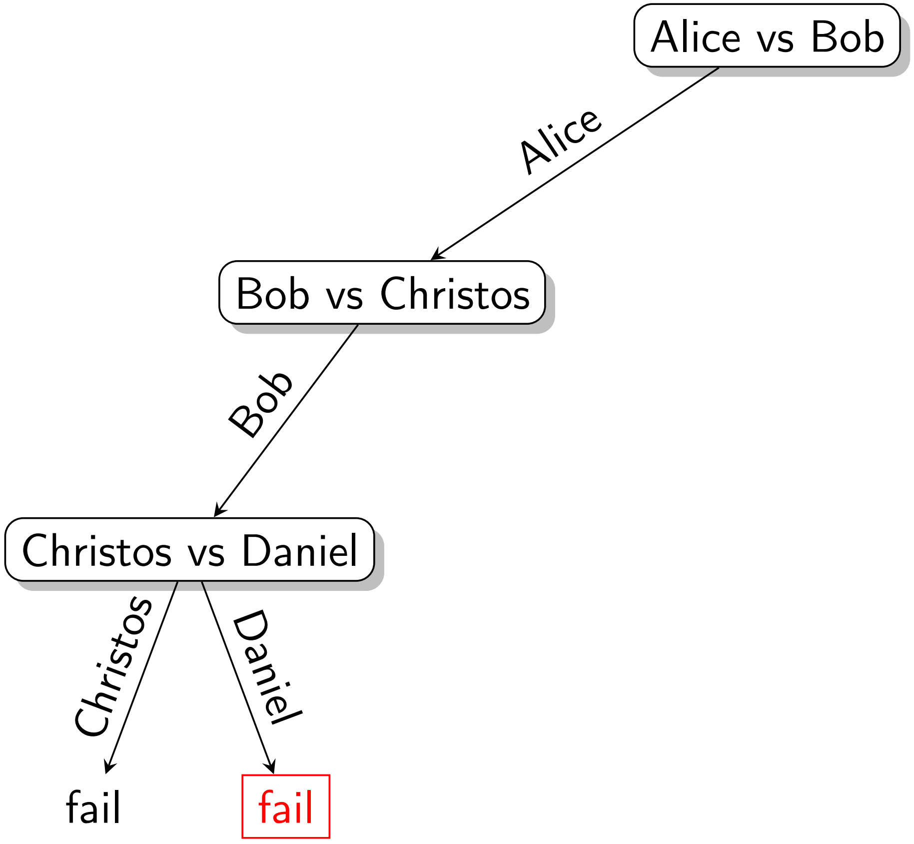
$k = 1$



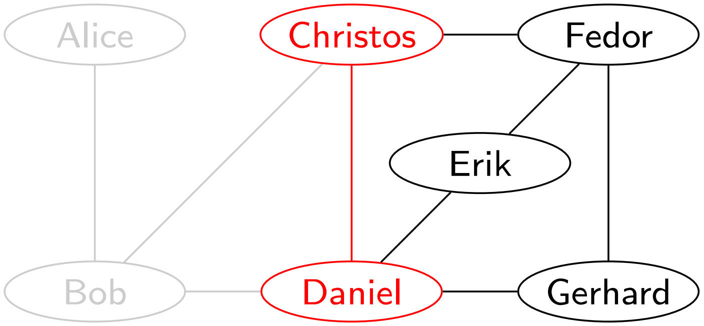
“Bar fight prevention” via Bounded Search Tree



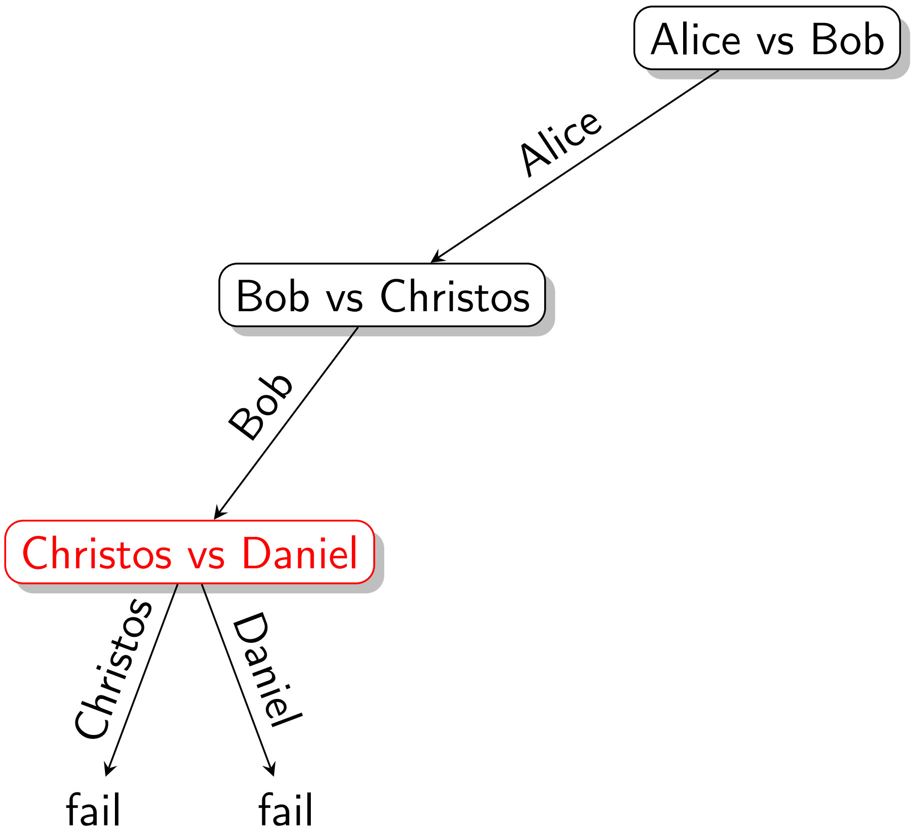
$k = 0$



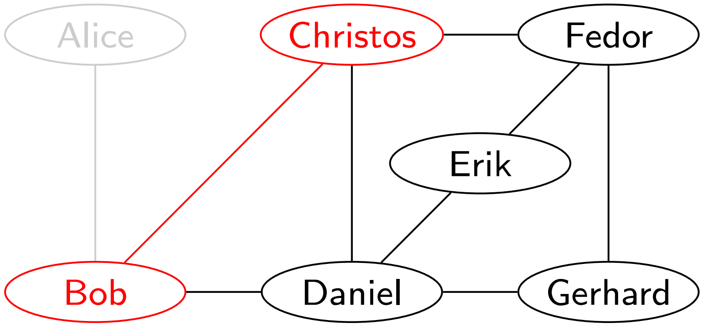
“Bar fight prevention” via Bounded Search Tree



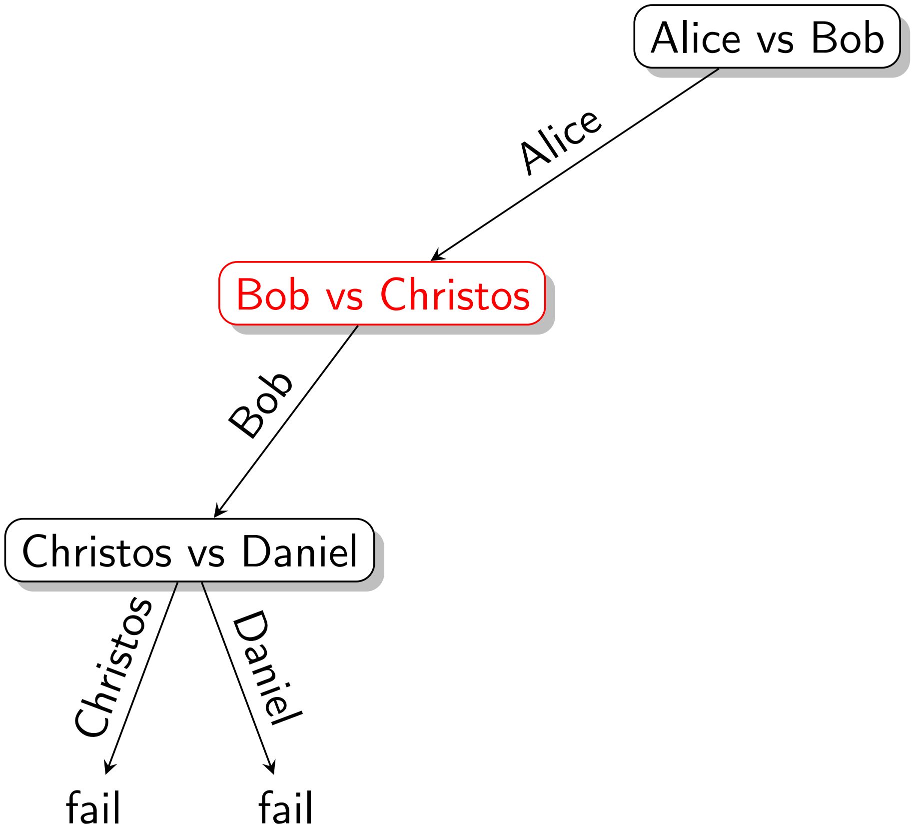
$k = 1$



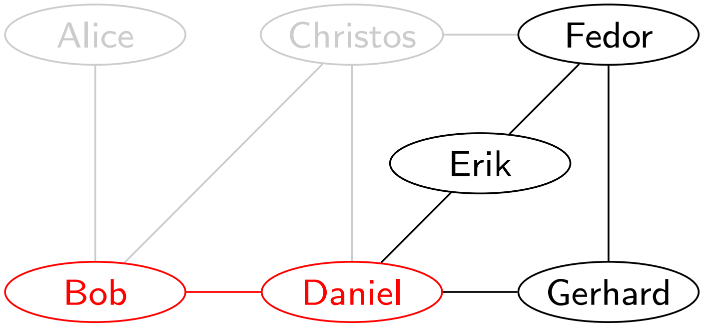
"Bar fight prevention" via Bounded Search Tree



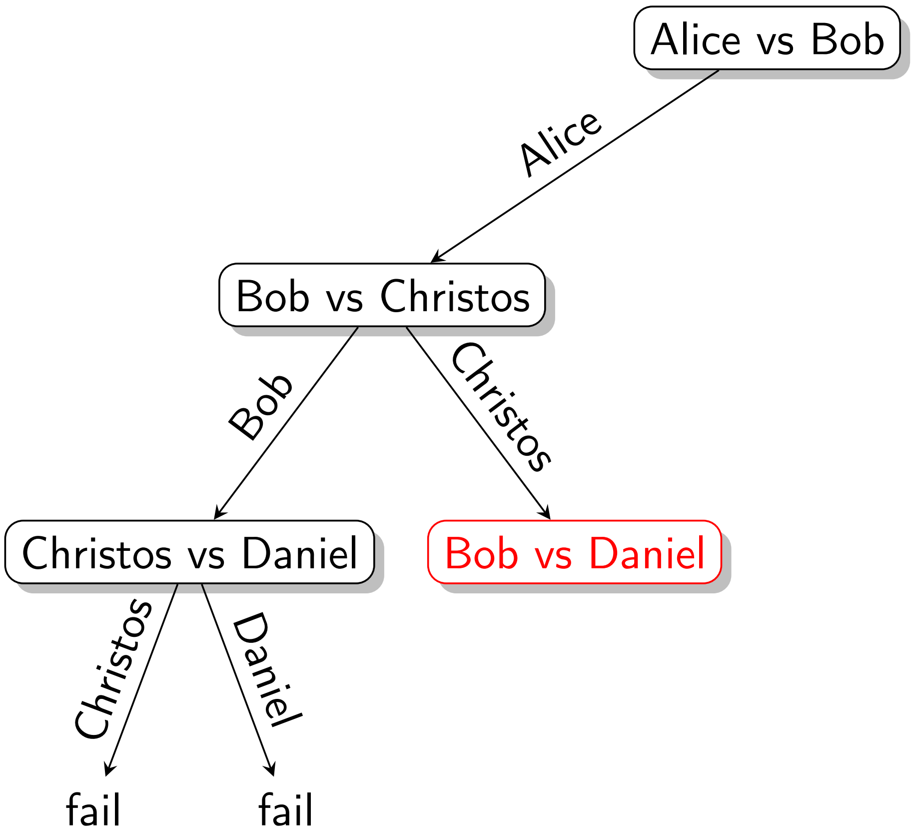
$k = 2$



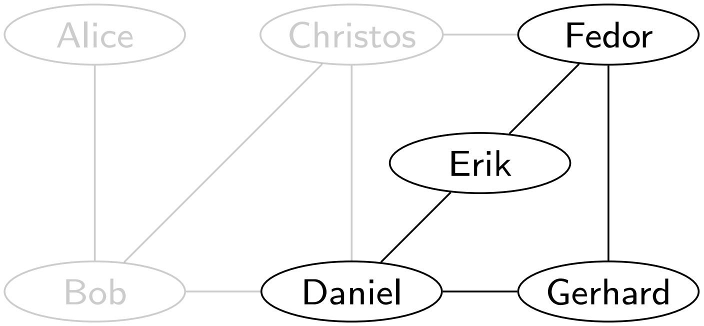
"Bar fight prevention" via Bounded Search Tree



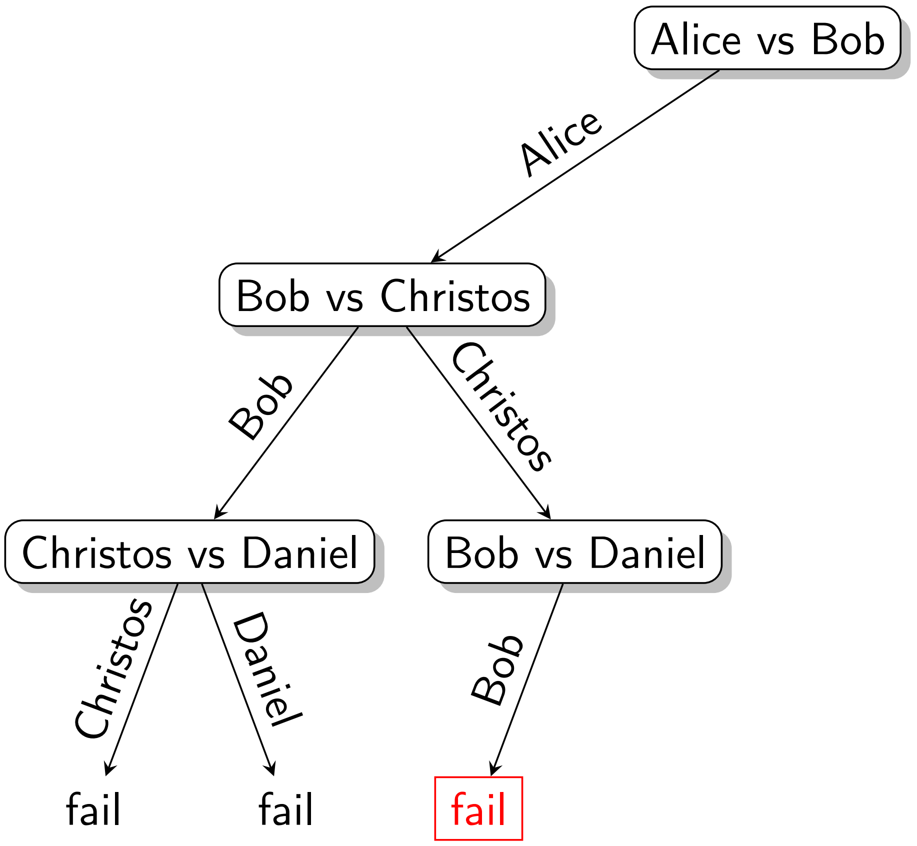
$k = 1$



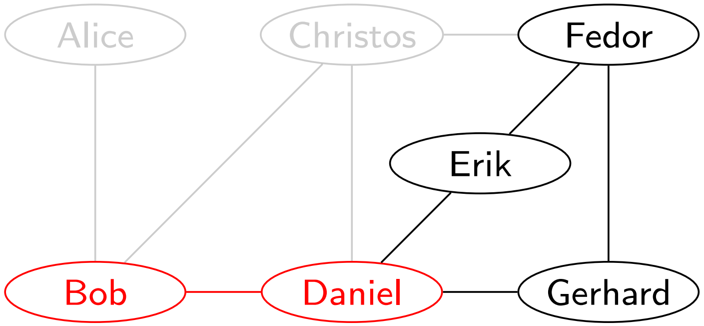
“Bar fight prevention” via Bounded Search Tree



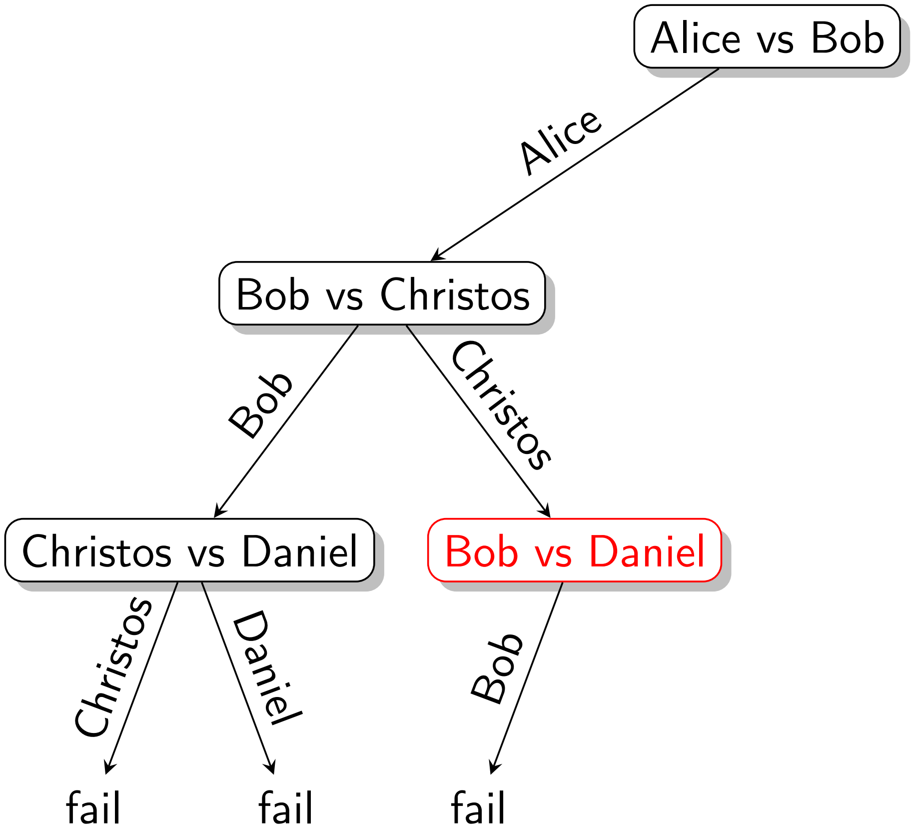
$k = 0$



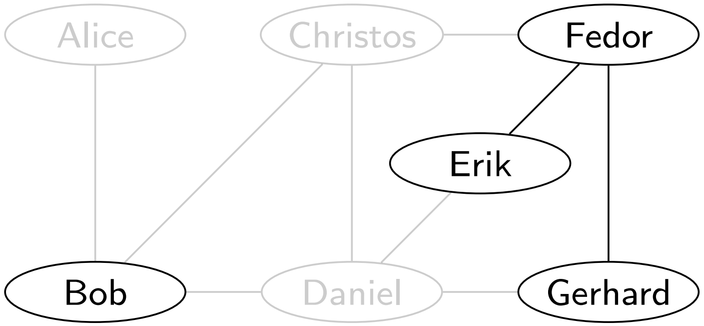
“Bar fight prevention” via Bounded Search Tree



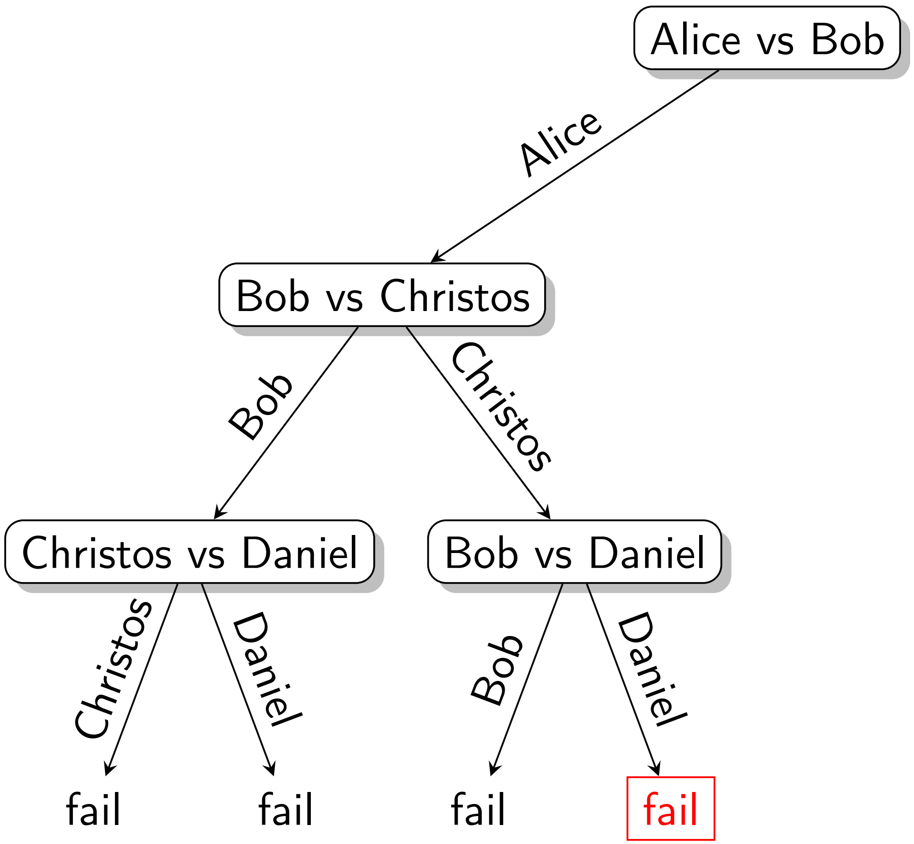
$k = 1$



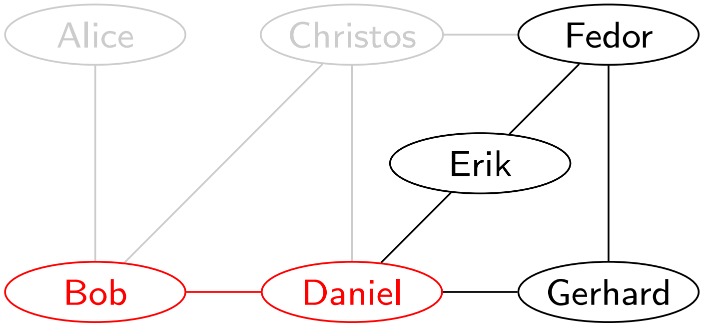
"Bar fight prevention" via Bounded Search Tree



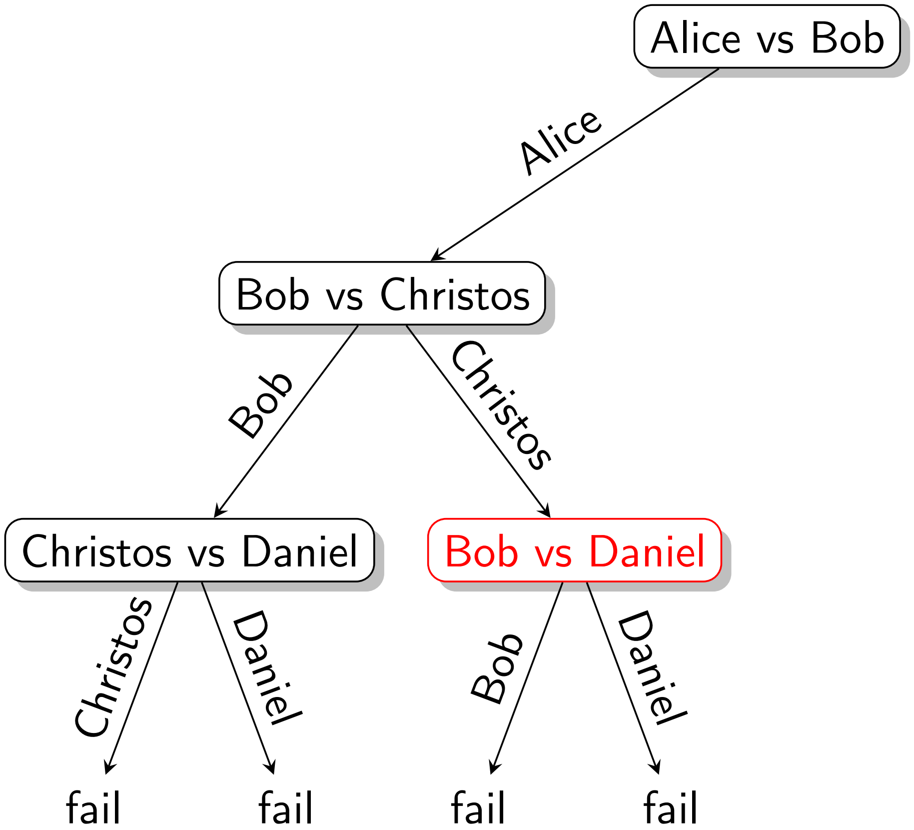
$k = 0$



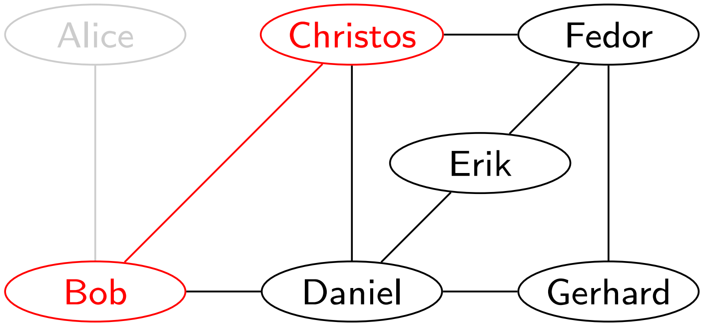
“Bar fight prevention” via Bounded Search Tree



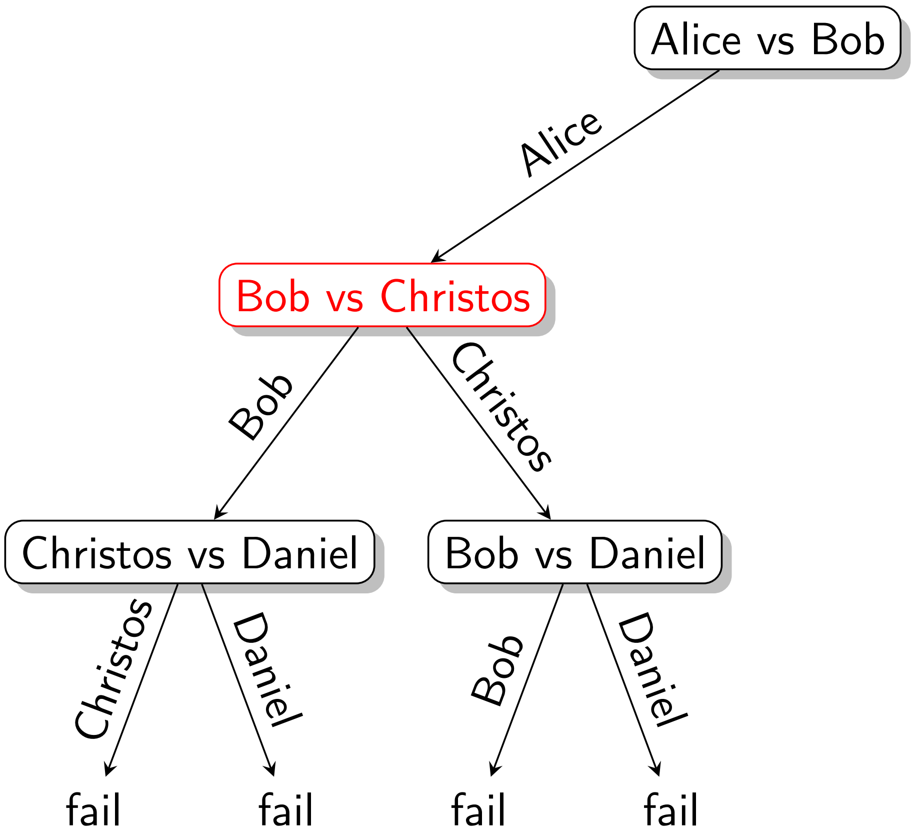
$k = 1$



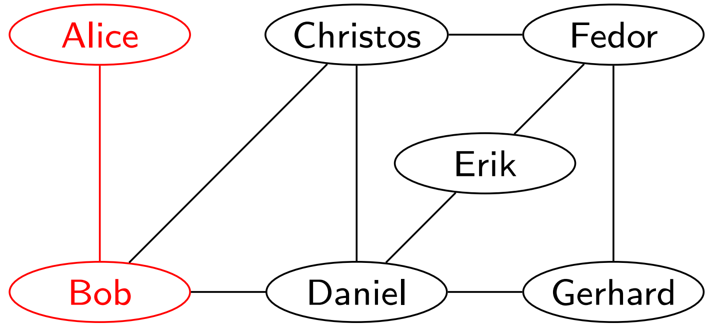
“Bar fight prevention” via Bounded Search Tree



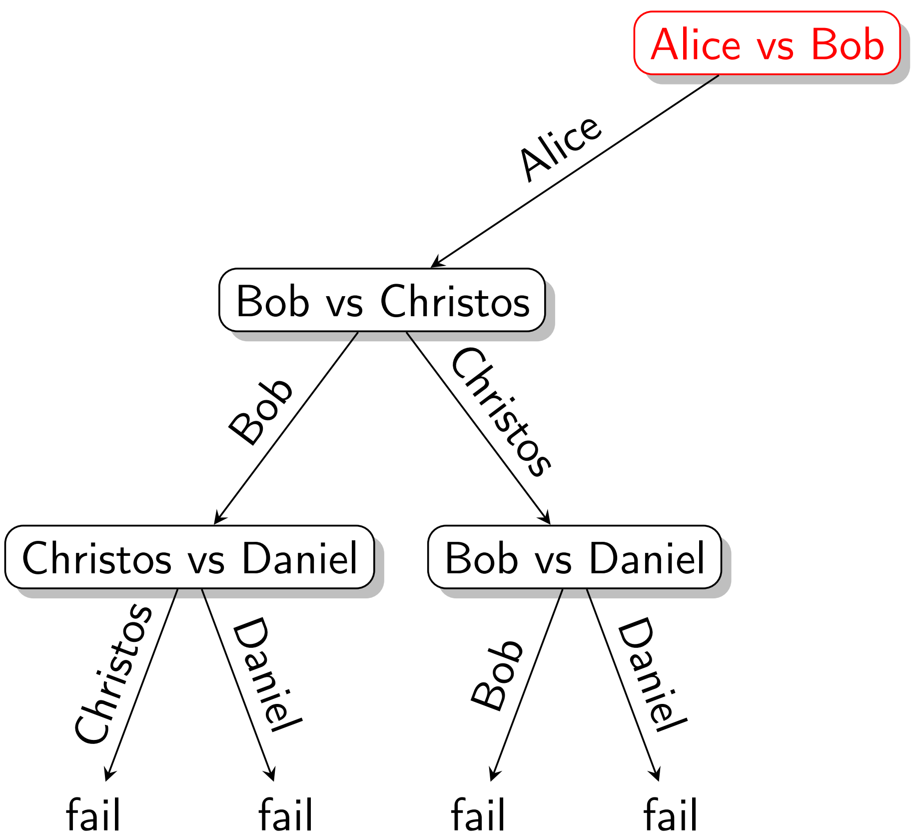
$k = 2$



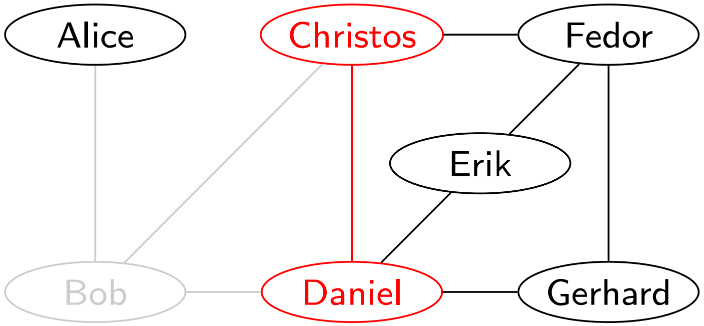
“Bar fight prevention” via Bounded Search Tree



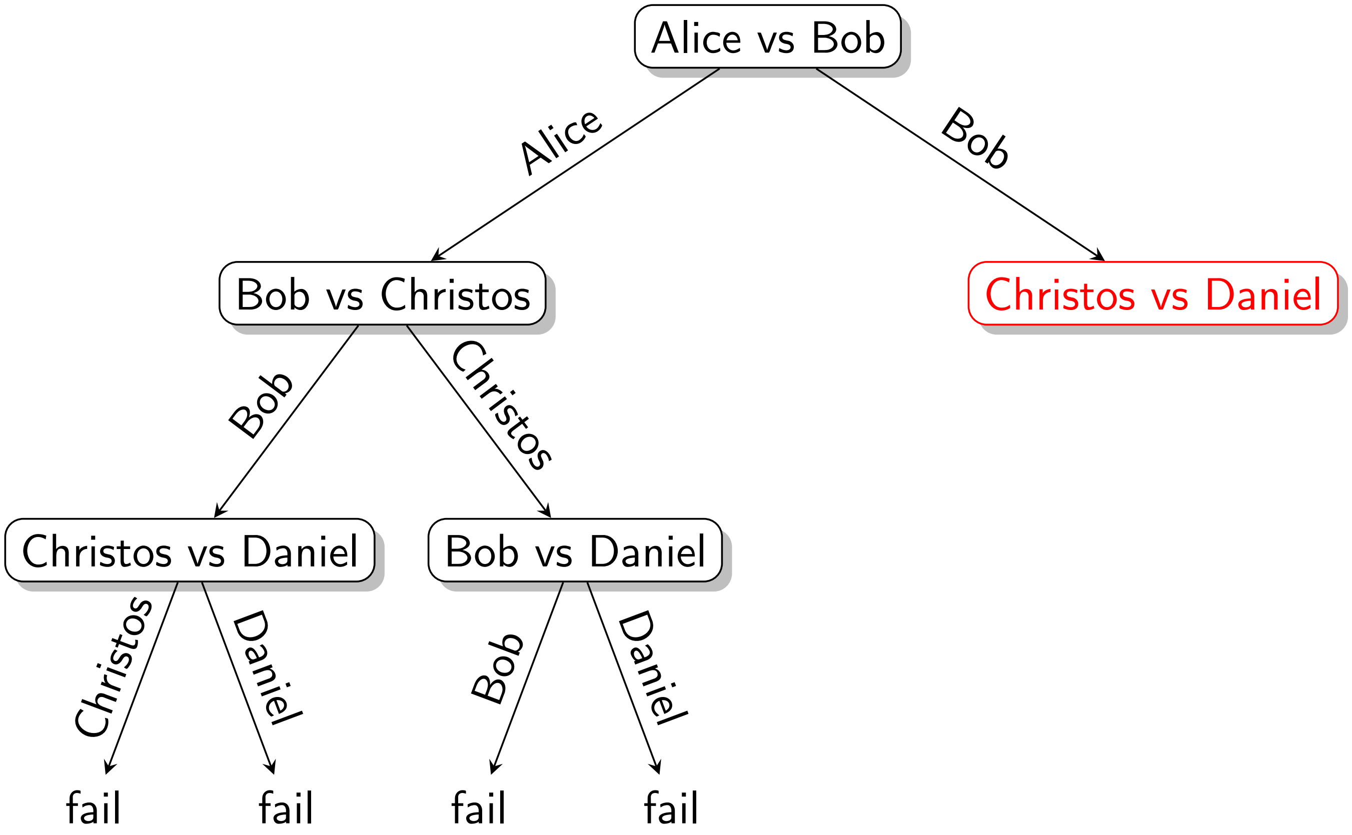
$k = 3$



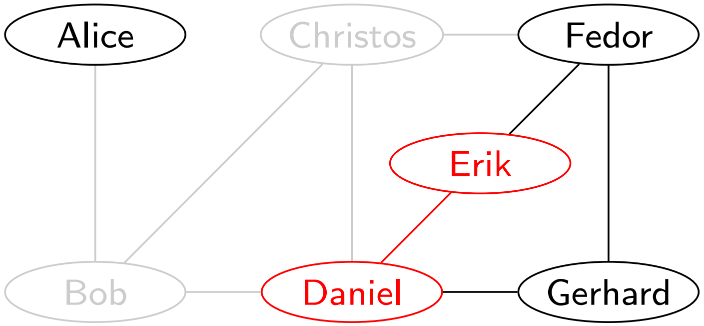
“Bar fight prevention” via Bounded Search Tree



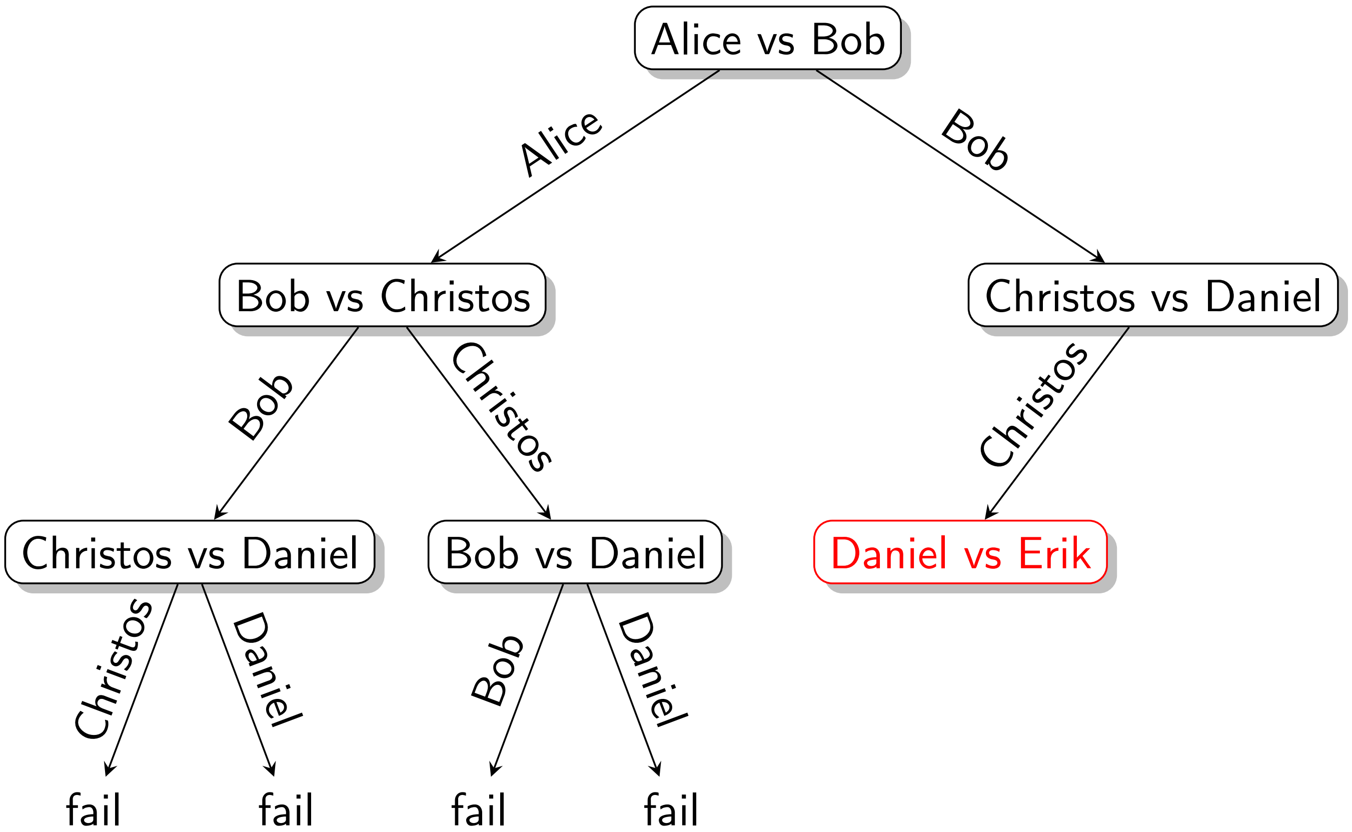
$k = 2$



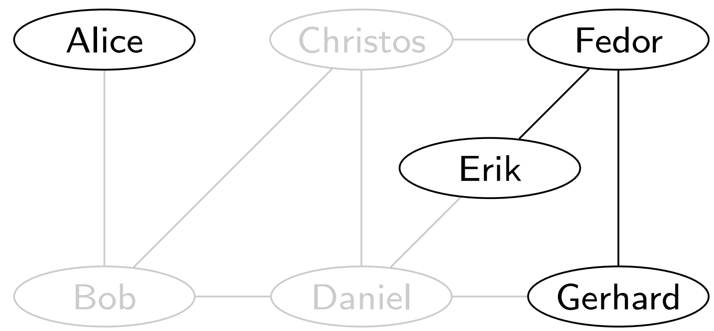
"Bar fight prevention" via Bounded Search Tree



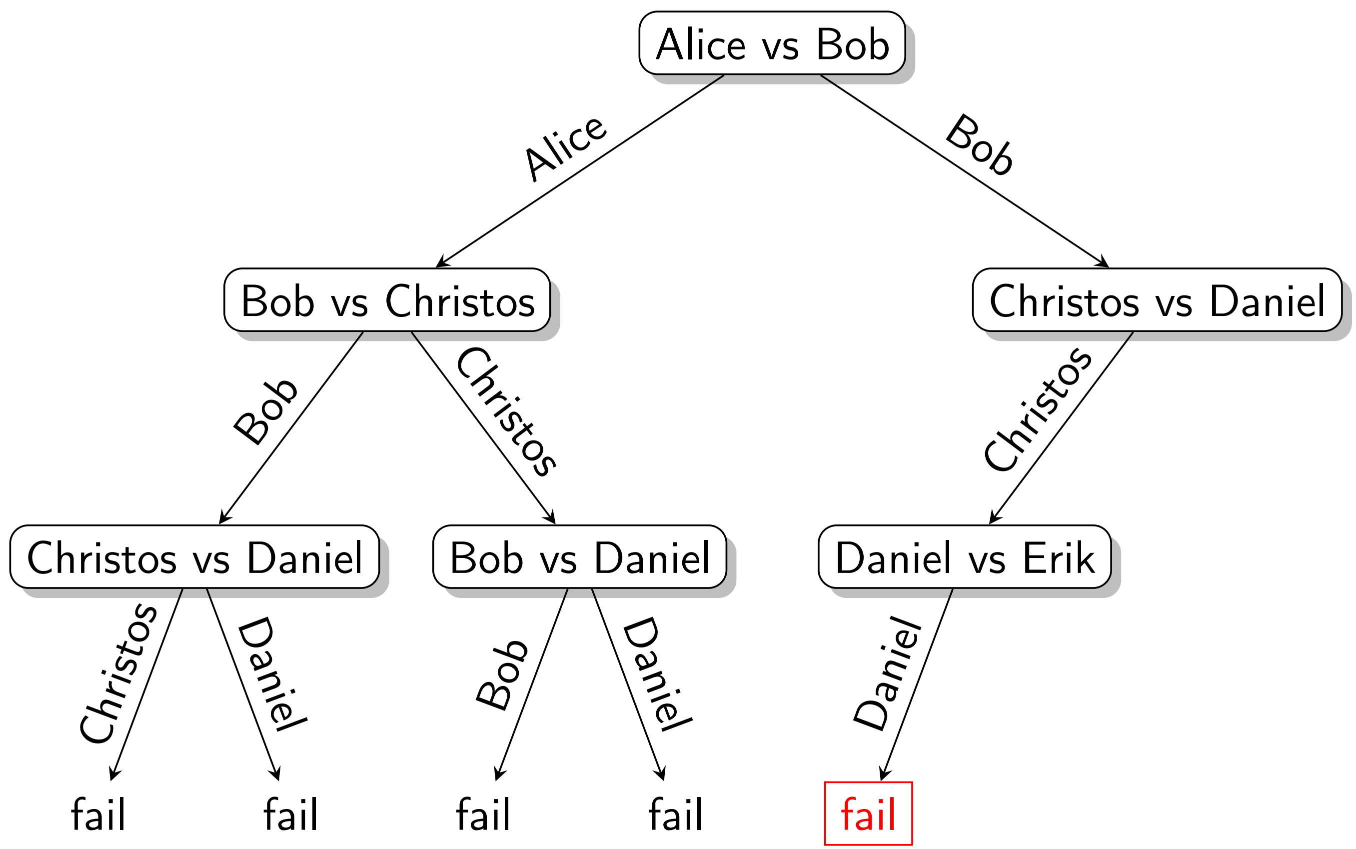
$k = 1$



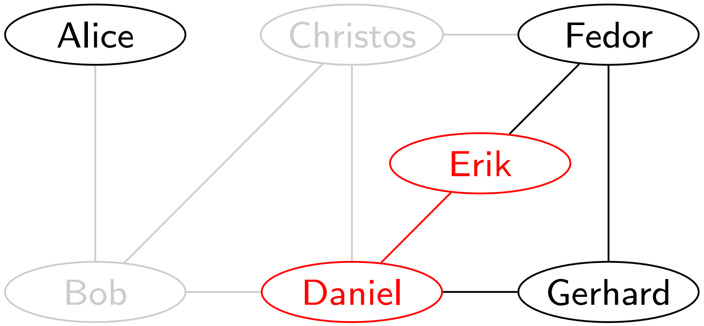
"Bar fight prevention" via Bounded Search Tree



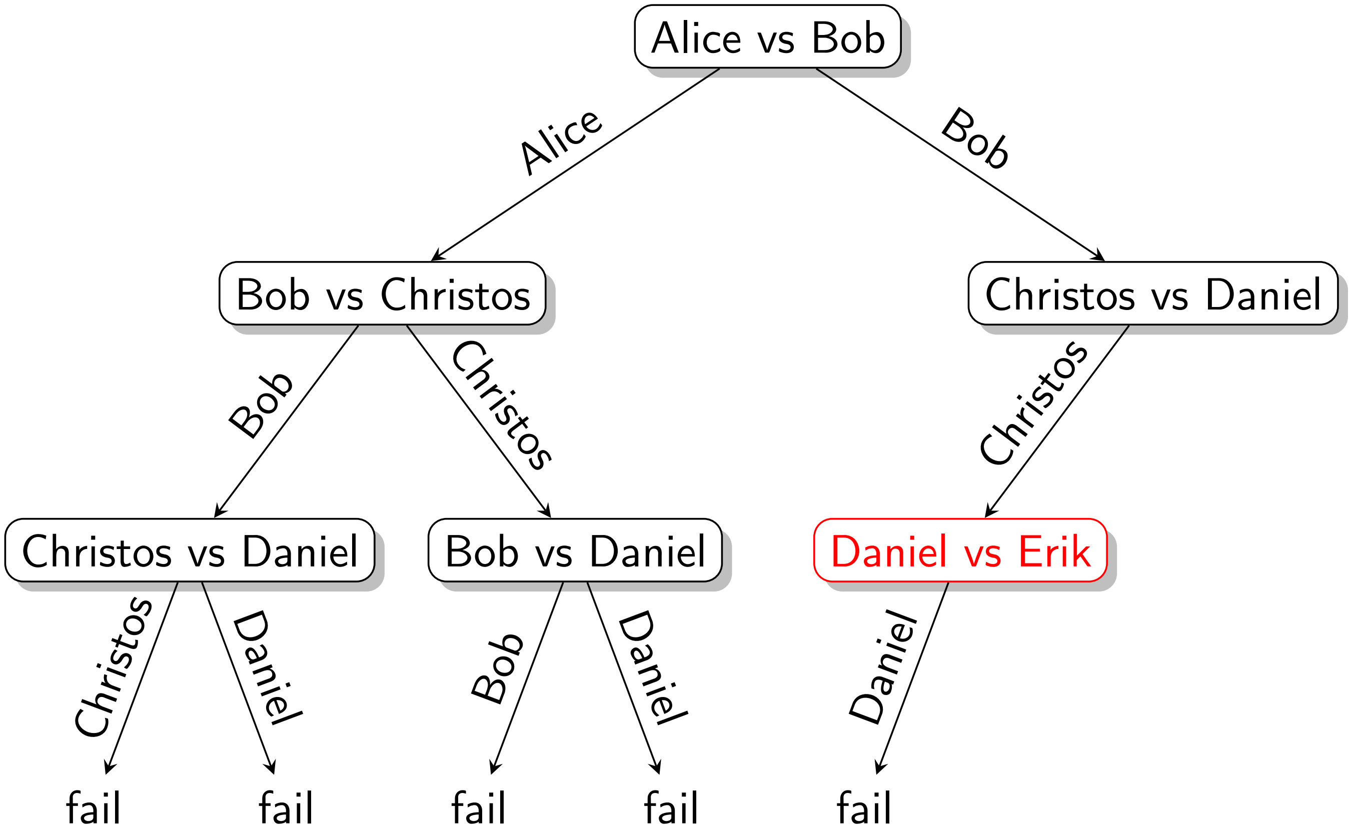
$k = 0$



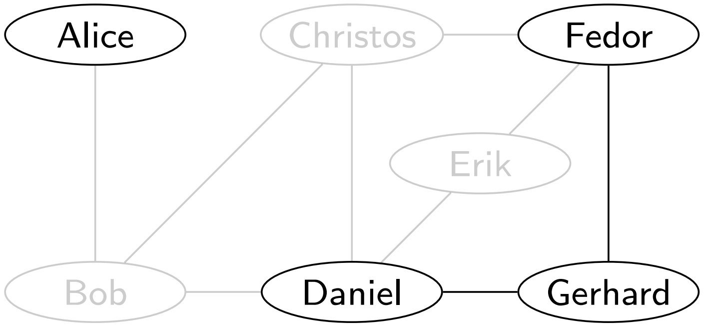
“Bar fight prevention” via Bounded Search Tree



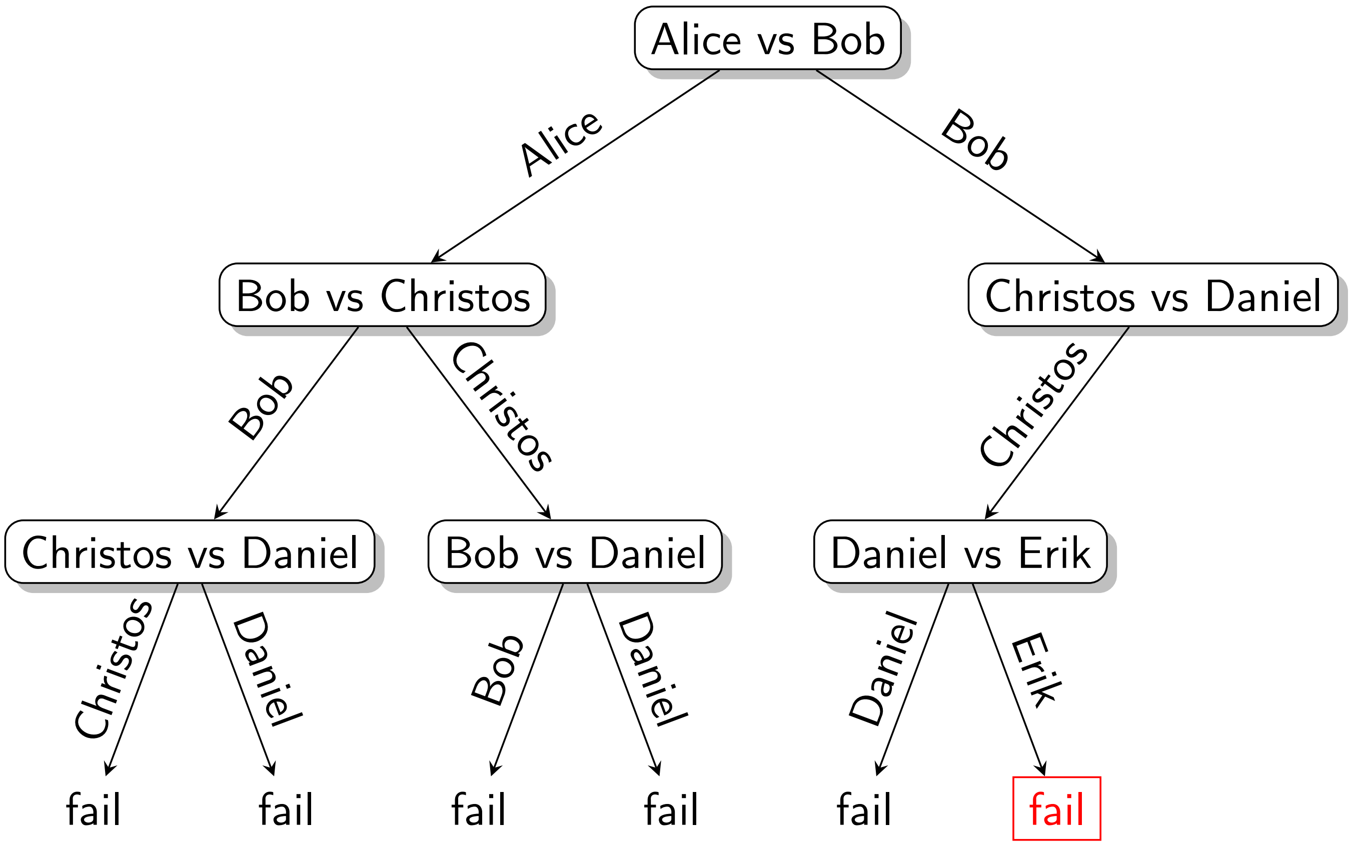
$k = 1$



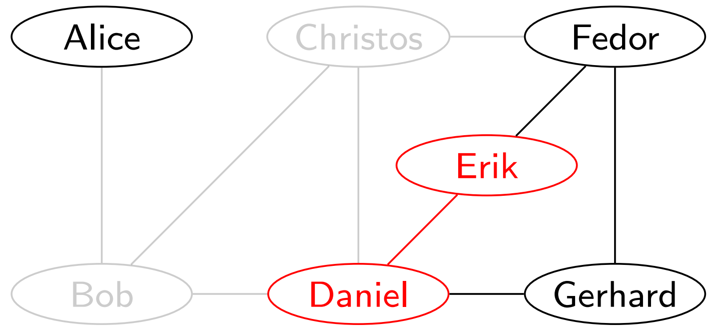
“Bar fight prevention” via Bounded Search Tree



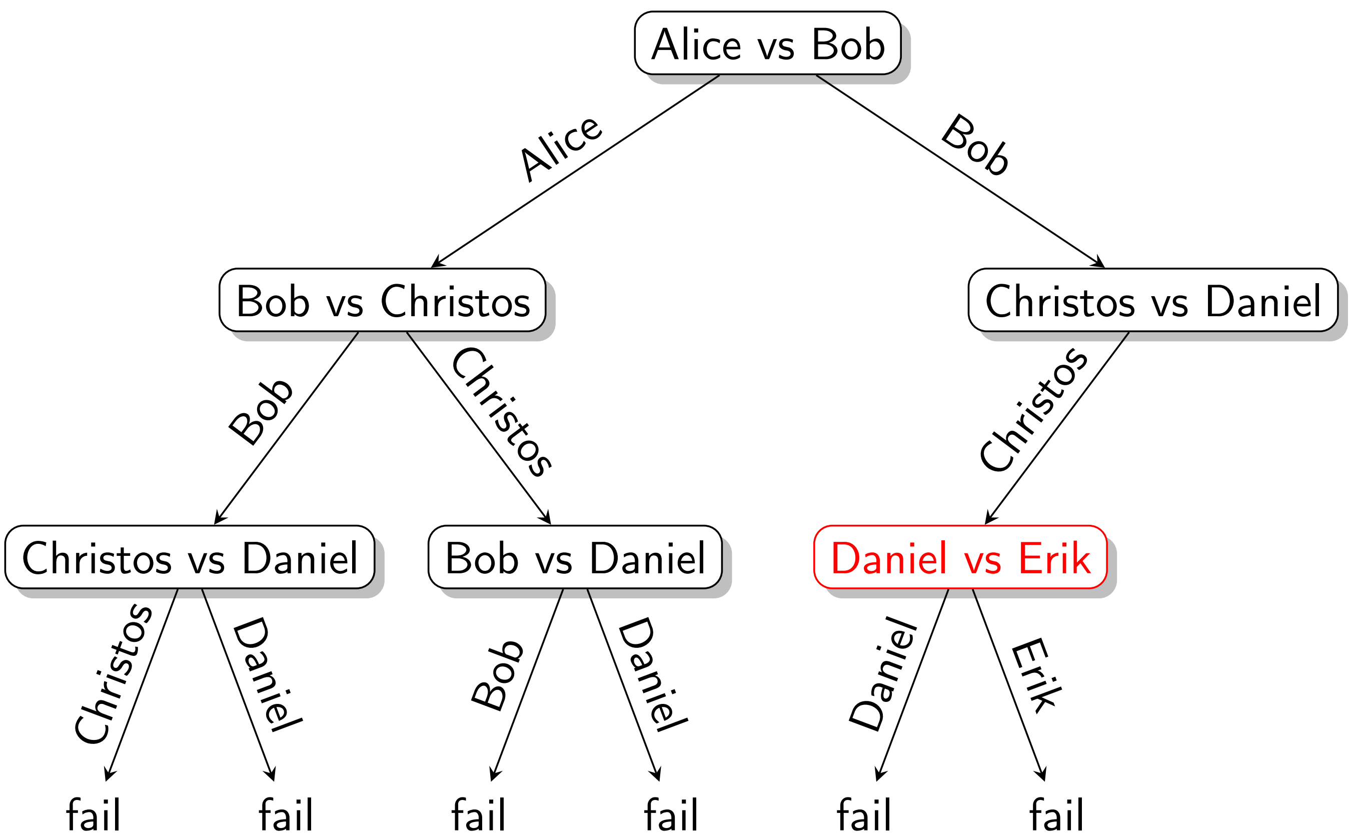
$k = 0$



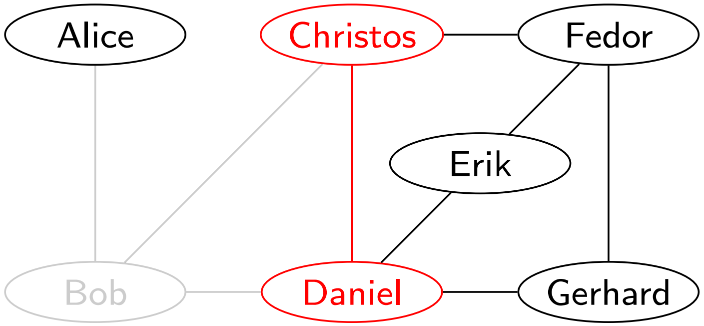
“Bar fight prevention” via Bounded Search Tree



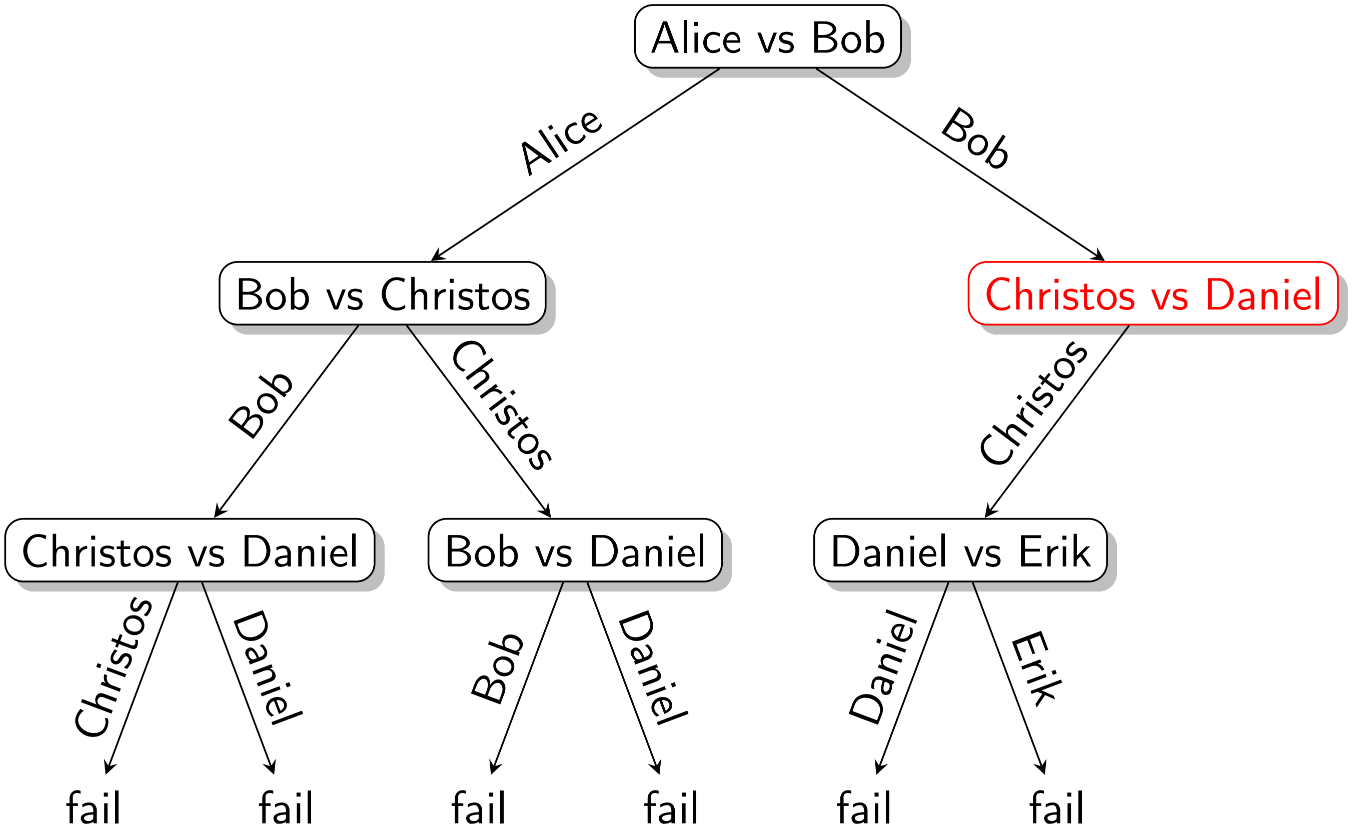
$k = 1$



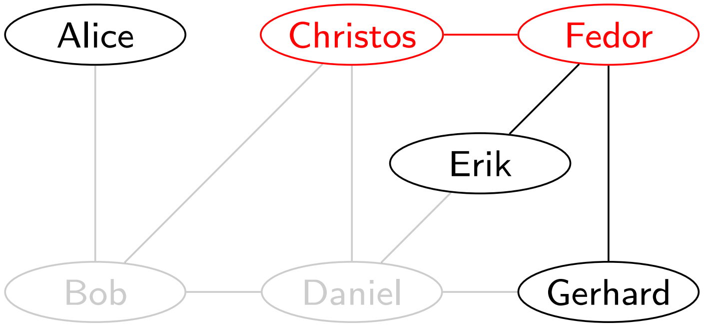
"Bar fight prevention" via Bounded Search Tree



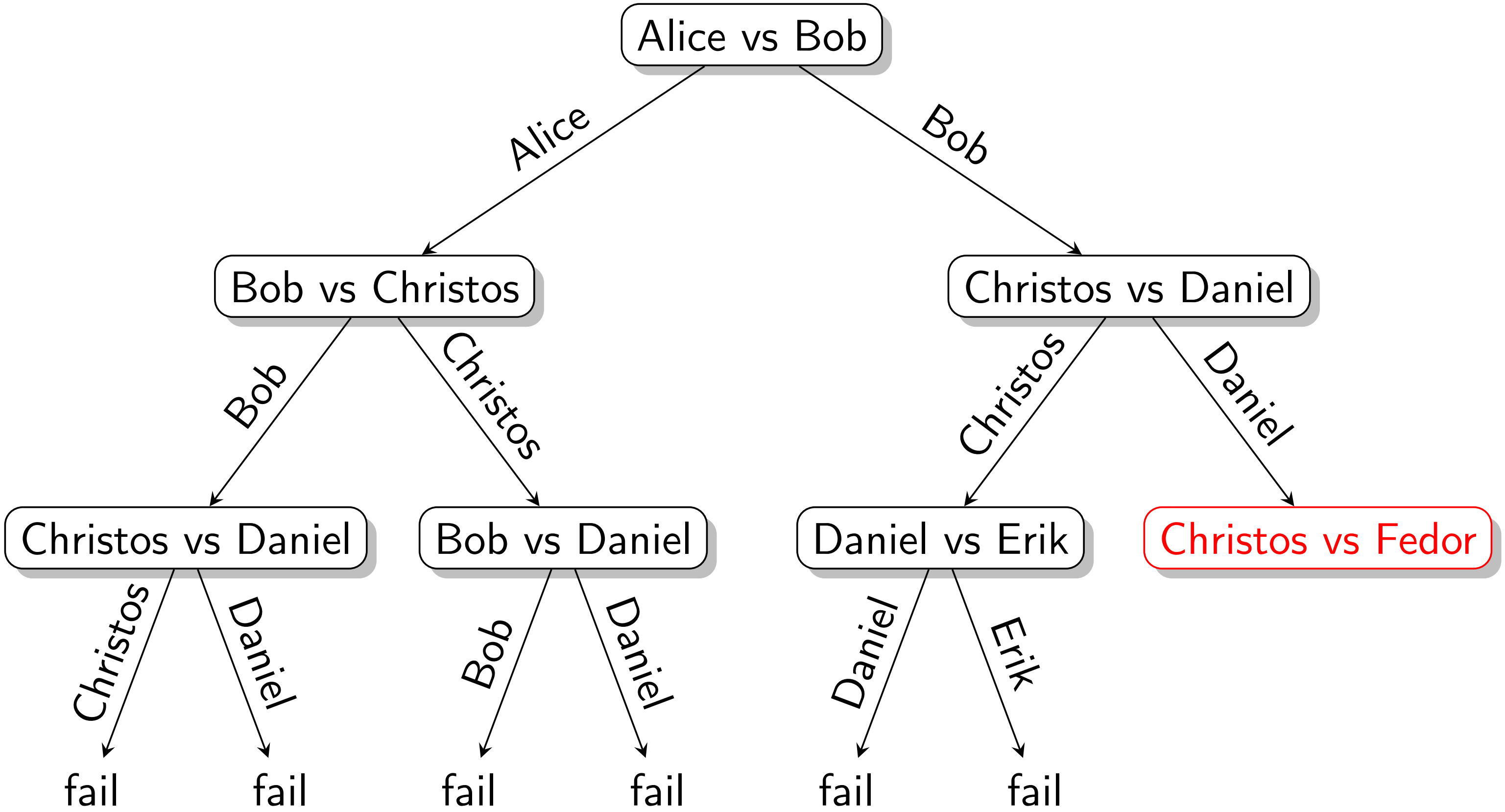
$k = 2$



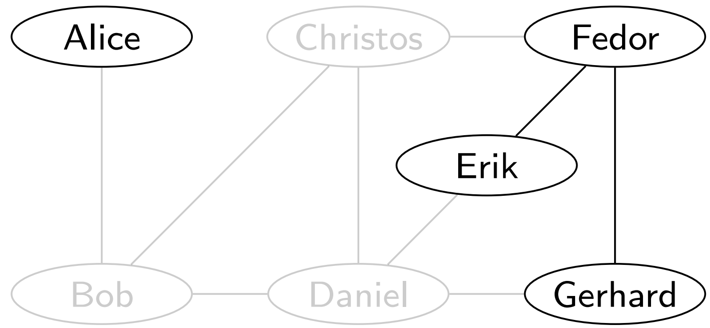
“Bar fight prevention” via Bounded Search Tree



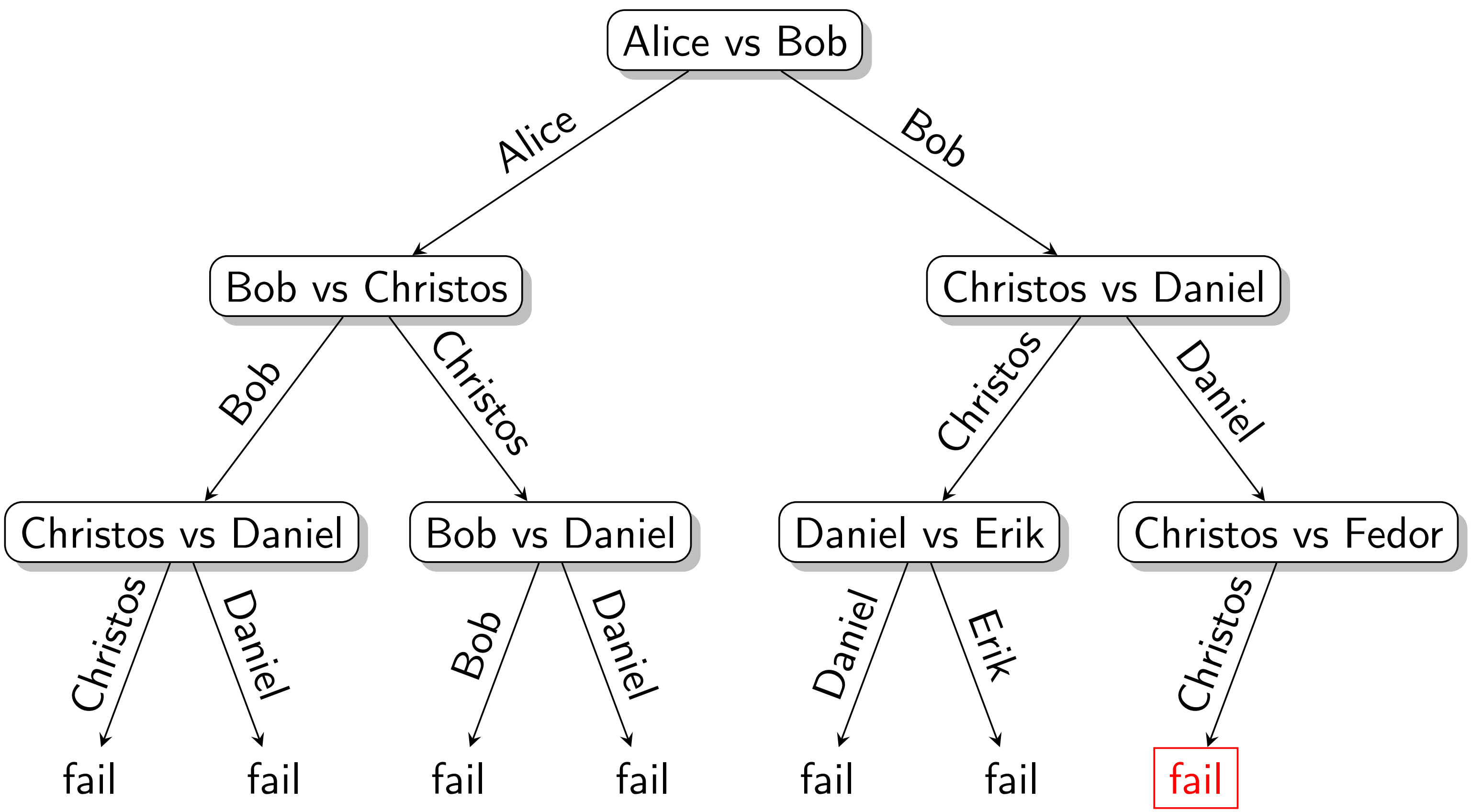
$k = 1$



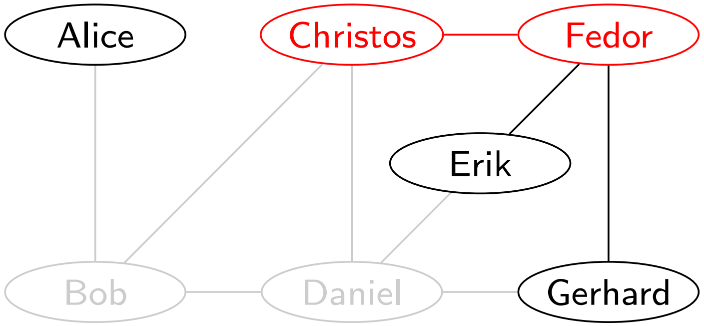
“Bar fight prevention” via Bounded Search Tree



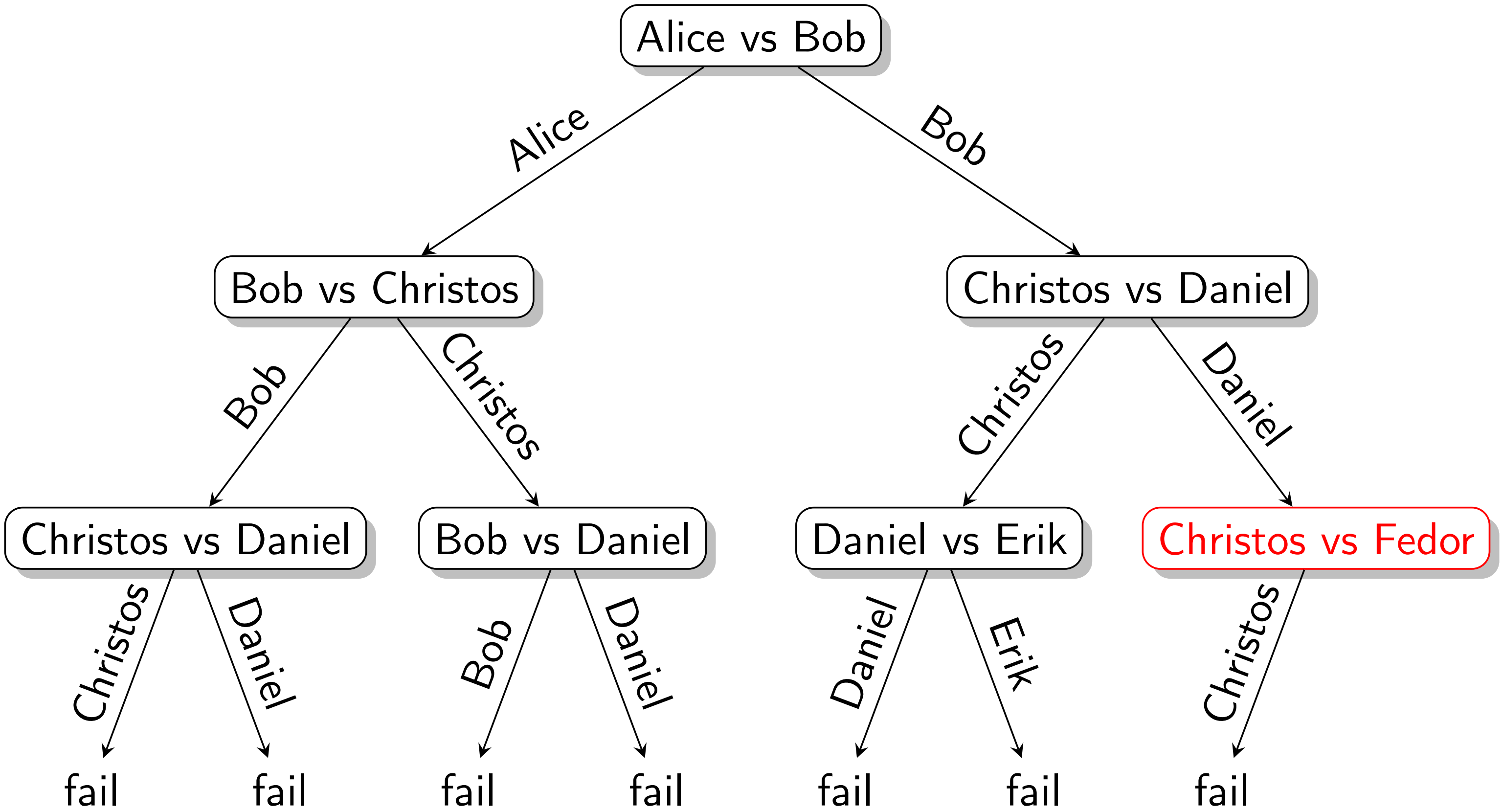
$k = 0$



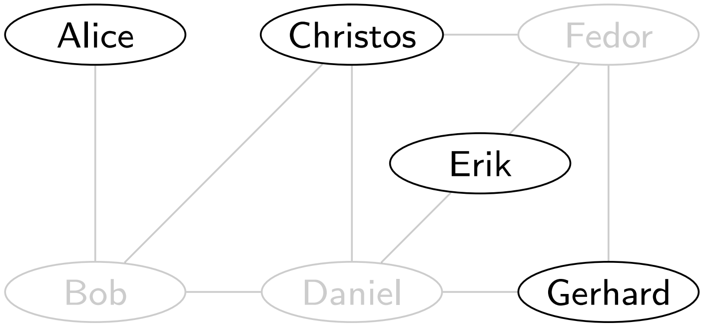
“Bar fight prevention” via Bounded Search Tree



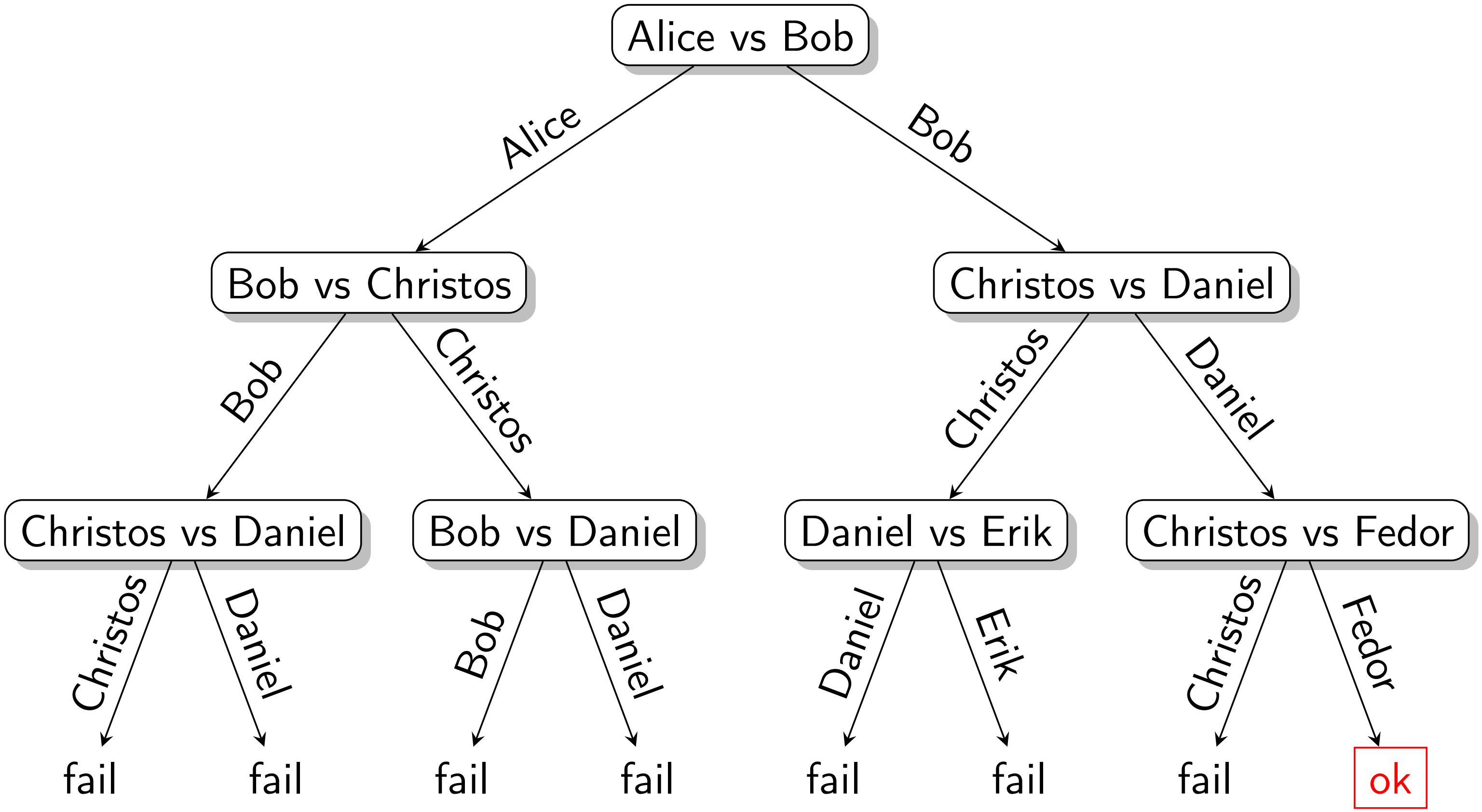
$k = 1$



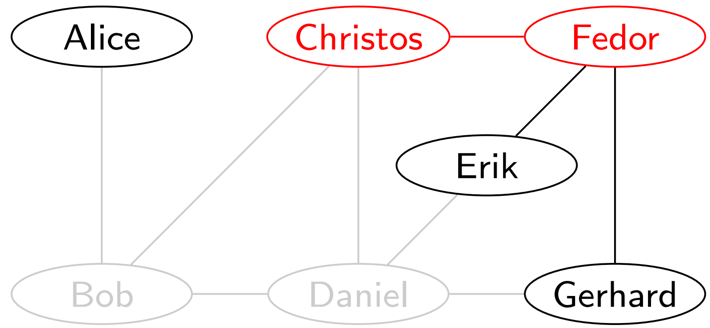
“Bar fight prevention” via Bounded Search Tree



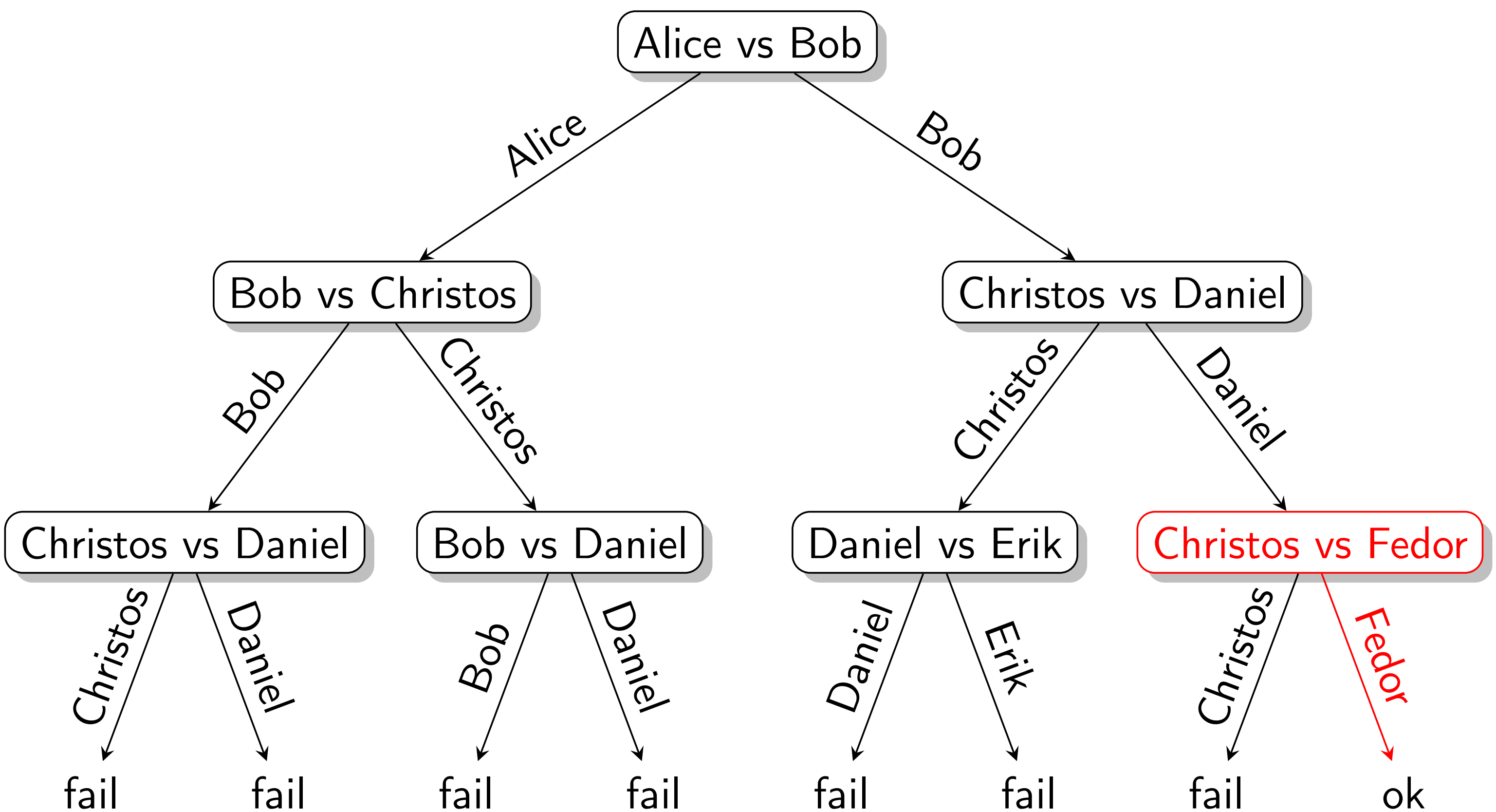
$k = 0 \qquad C = \emptyset$



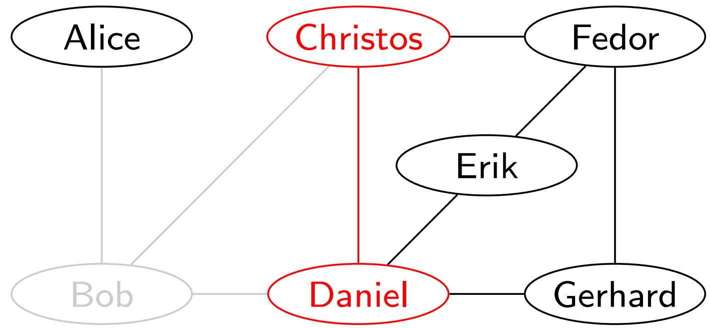
“Bar fight prevention” via Bounded Search Tree



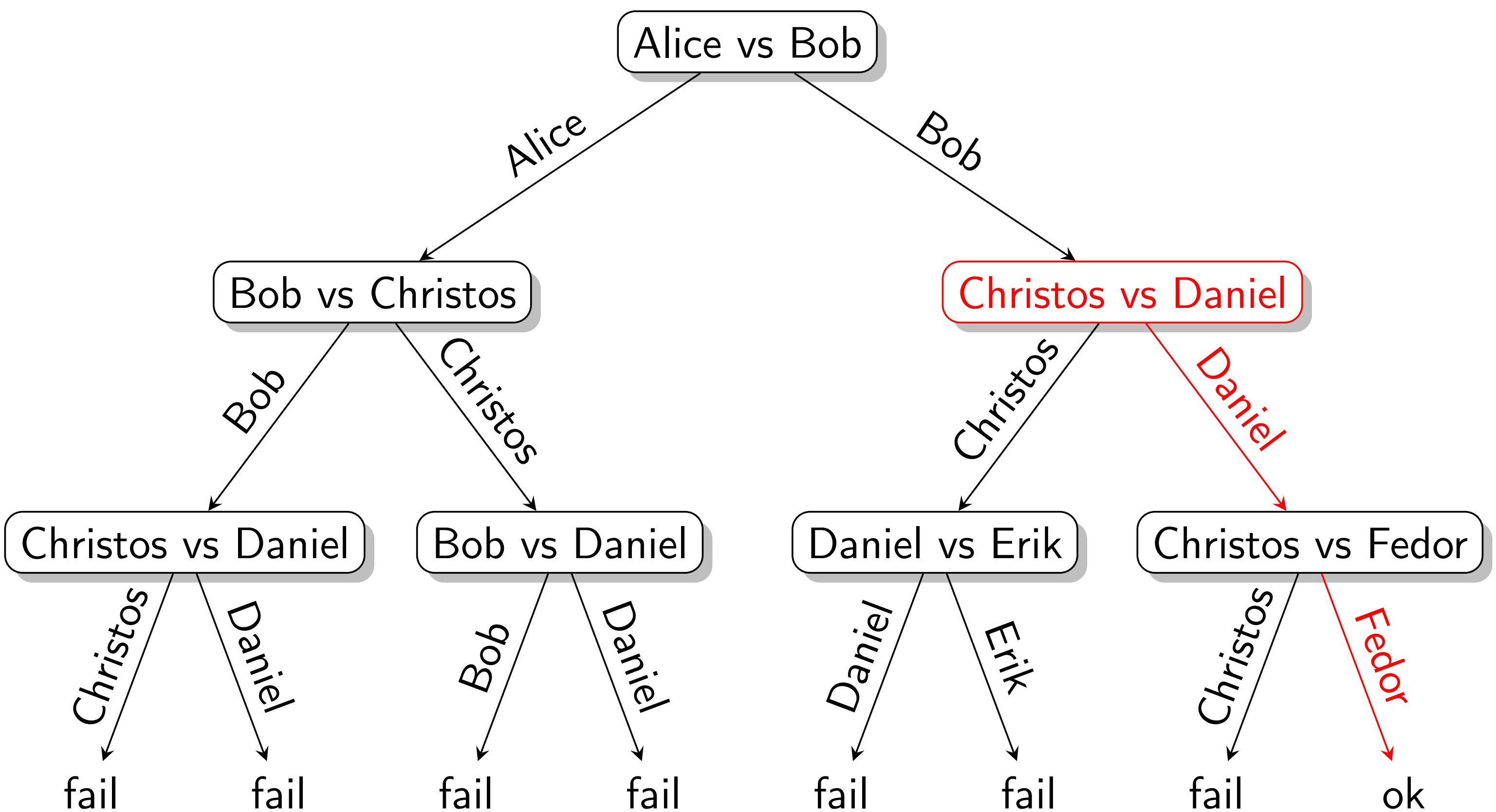
$k = 1$ $\qquad C = \{\text{Fedor}\}$



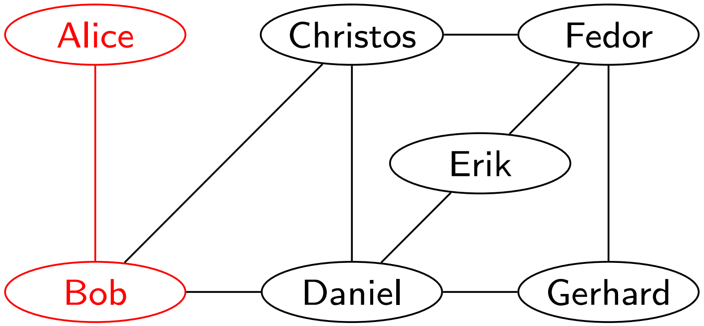
“Bar fight prevention” via Bounded Search Tree



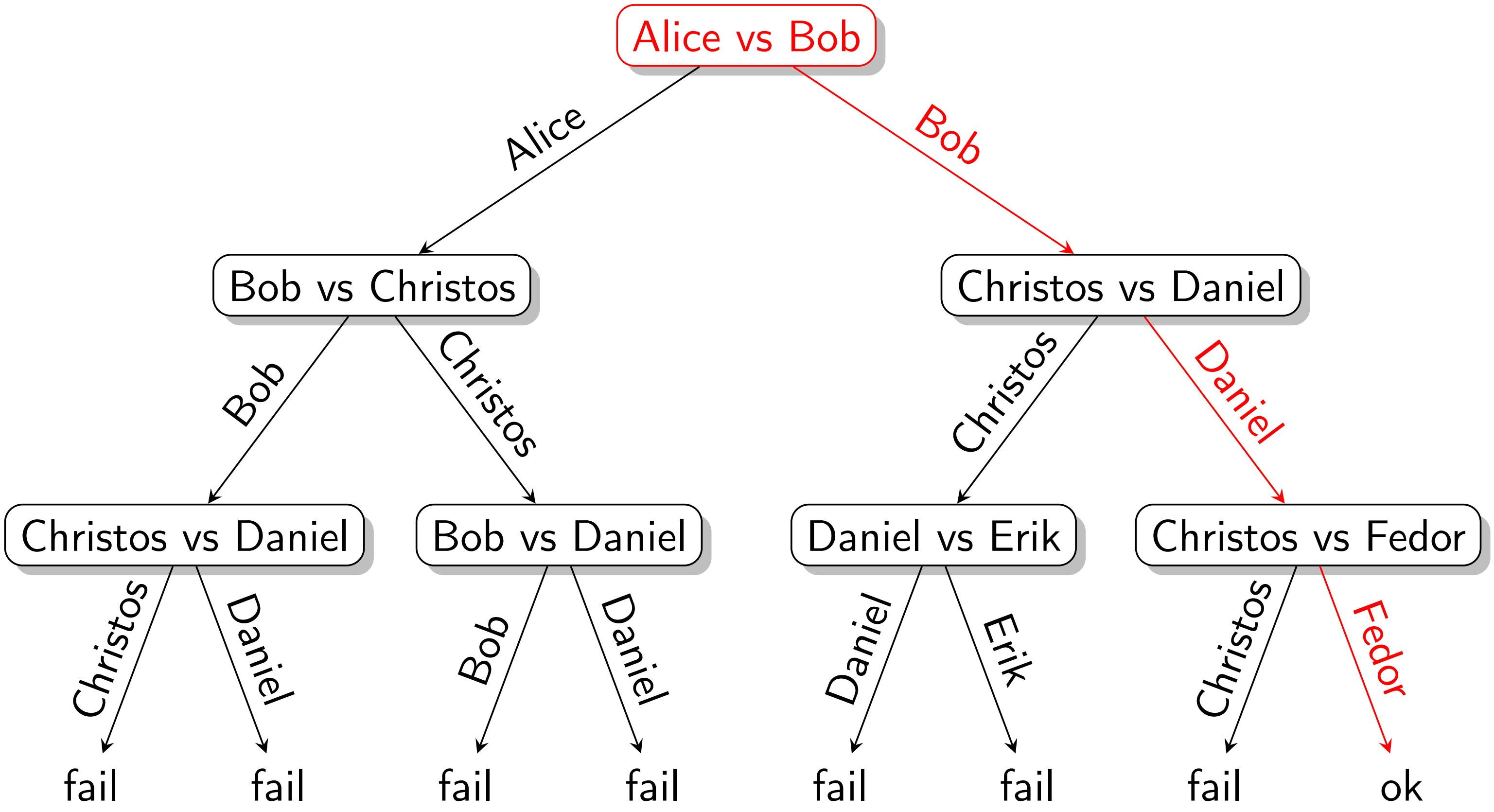
$k = 2$ $\qquad C = \{\text{Fedor}, \text{Daniel}\}$



“Bar fight prevention” via Bounded Search Tree



$k = 3$ $\qquad$ $C = \{\text{Fedor}, \text{Daniel}, \text{Bob}\}$



AADS Lecture 9, Part 4

FPT vs XP

FPT vs XP

An important feature of the Bar Fight Prevention problem is the existence of the *parameter* k. The problem of finding the minimum k that works is NP-complete, but for any fixed constant k we have just seen two linear-time algorithms!

We say the problem is *parameterized* by the *parameter* k. In this case k is the maximum solution size, but other problems may have different parameters (and may have more than one).

Definition: A parameterized problem is *Fixed Parameter Tractable (FPT)* if it is has an algorithm with running time $f(k) \cdot n^c$ for some function f and some constant $c \in \mathbb{R}$.

Definition: A parameterized problem is *Slice-wise Polynomial (XP)* if it is has an algorithm with running time $f(k) \cdot n^{g(k)}$ for some functions f, g.

Note: FPT $\subset$ XP, why?

FPT vs XP

An important feature of the Bar Fight Prevention problem is the existence of the *parameter* k. The problem of finding the minimum k that works is NP-complete, but for any fixed constant k we have just seen two linear-time algorithms!

We say the problem is *parameterized* by the *parameter* k. In this case k is the maximum solution size, but other problems may have different parameters (and may have more than one).

Definition: A parameterized problem is *Fixed Parameter Tractable (FPT)* if it is has an algorithm with running time $f(k) \cdot n^c$ for some function f and some constant $c \in \mathbb{R}$.

Definition: A parameterized problem is *Slice-wise Polynomial (XP)* if it is has an algorithm with running time $f(k) \cdot n^{g(k)}$ for some functions f, g.

Note: FPT $\subset$ XP, why?

FPT vs XP

An important feature of the Bar Fight Prevention problem is the existence of the *parameter* k. The problem of finding the minimum k that works is NP-complete, but for any fixed constant k we have just seen two linear-time algorithms!

We say the problem is *parameterized* by the *parameter* k. In this case k is the maximum solution size, but other problems may have different parameters (and may have more than one).

Definition: A parameterized problem is *Fixed Parameter Tractable (FPT)* if it is has an algorithm with running time $f(k) \cdot n^c$ for some function f and some constant $c \in \mathbb{R}$.

Definition: A parameterized problem is *Slice-wise Polynomial (XP)* if it is has an algorithm with running time $f(k) \cdot n^{g(k)}$ for some functions f, g.

Note: FPT $\subset$ XP, why?

FPT vs XP

An important feature of the Bar Fight Prevention problem is the existence of the *parameter* k. The problem of finding the minimum k that works is NP-complete, but for any fixed constant k we have just seen two linear-time algorithms!

We say the problem is *parameterized* by the *parameter* k. In this case k is the maximum solution size, but other problems may have different parameters (and may have more than one).

Definition: A parameterized problem is *Fixed Parameter Tractable (FPT)* if it is has an algorithm with running time $f(k) \cdot n^c$ for some function f and some constant $c \in \mathbb{R}$. → Bar Fight ...

Definition: A parameterized problem is *Slice-wise Polynomial (XP)* if it is has an algorithm with running time $f(k) \cdot n^{g(k)}$ for some functions f, g.

Note: FPT $\subset$ XP, why?

FPT vs XP

An important feature of the Bar Fight Prevention problem is the existence of the *parameter* k. The problem of finding the minimum k that works is NP-complete, but for any fixed constant k we have just seen two linear-time algorithms!

We say the problem is *parameterized* by the *parameter* k. In this case k is the maximum solution size, but other problems may have different parameters (and may have more than one).

Definition: A parameterized problem is *Fixed Parameter Tractable (FPT)* if it is has an algorithm with running time $f(k) \cdot n^c$ for some function f and some constant $c \in \mathbb{R}$.

Definition: A parameterized problem is *Slice-wise Polynomial (XP)* if it is has an algorithm with running time $f(k) \cdot n^{g(k)}$ for some functions f, g.

Note: FPT $\subset$ XP, why? 如果一个是FPT，则它一定是XP.
因为它的 $g(k)=c$

FPT vs XP

An important feature of the Bar Fight Prevention problem is the existence of the *parameter* k. The problem of finding the minimum k that works is NP-complete, but for any fixed constant k we have just seen two linear-time algorithms!

We say the problem is *parameterized* by the *parameter* k. In this case k is the maximum solution size, but other problems may have different parameters (and may have more than one).

Definition: A parameterized problem is *Fixed Parameter Tractable (FPT)* if it is has an algorithm with running time $f(k) \cdot n^c$ for some function f and some constant $c \in \mathbb{R}$.

Definition: A parameterized problem is *Slice-wise Polynomial (XP)* if it is has an algorithm with running time $f(k) \cdot n^{g(k)}$ for some functions f, g.

Note: FPT $\subset$ XP, why? Simply set $g(k) = c$.

Example: Vertex k-Coloring

Problem: Given graph G and an integer k, does G have a proper vertex coloring with k colors?

the end points
of every edge
have distinct colors

Lemma

Unless $P = NP$, this problem is not XP and therefore not FPT.

Proof.

The problem is NP-hard even for $k = 5$, so unless $P = NP$ there can be no algorithm for general k with running time $f(k) \cdot n^{g(k)}$. $\square$

如果是则

反证 would give you a polynomial time

for solving it for k=5

但除非 P=NP, 不然不可能

Example: k-Clique

Problem: Given graph G and an integer k, does G have a clique of size k?

全连接图

Lemma

k-clique is XP.

Proof.

A simple brute-force algorithm is to check every k-subset of the vertices. There are $\binom{n}{k} \leq n^k$ such subsets, and we can check in $\mathcal{O}(k^2)$ time whether a given subset forms a clique. Thus the running time of this algorithm is $\mathcal{O}(k^2 \cdot n^k)$ which proves the problem is in $\boxed{\text{XP}}$. □

It is unknown whether k-clique is FPT, but it is widely believed that $\mathcal{O}(n^k)$ is optimal which would prove it is not.

Example: k-Clique parameterized by Δ

Problem: Given graph G with maximum degree Δ, does G have a clique of size k?

Lemma

k-clique is FPT when parameterized by the maximum degree Δ.

Proof.

A naive algorithm is for each vertex to try all subsets of its neighbors. There are at most $n \cdot 2^{\Delta}$ such subsets and each can be checked in $\mathcal{O}(\Delta^2)$ time. The total time is thus $\mathcal{O}((2^{\Delta} \cdot \Delta^2) \cdot n)$, which proves the problem is FPT. □

In fact, we can easily improve this algorithm to run in $\mathcal{O}(\binom{\Delta}{k-1} \cdot k^2 \cdot n) \subseteq \mathcal{O}(\Delta^{k-1} \cdot k^2 \cdot n)$ time.

There are often many possible choices of parameter. Choosing the right one for a specific problem is an art.

Summary

Todays topics were "Exact exponential algorithms" and "Parameterized Complexity". We have covered

- The natural brute force algorithm for problems in NP.
- An exact $\mathcal{O}^\star(2^n)$-time dynamic programming algorithm for TSP.
- An exact $\mathcal{O}^\star(3^{n/3})$-time branching algorithm for MIS.
- A kernelization for the "Bar Fight Prevention" problem, a.k.a. k-vertex cover.
- A bounded search tree algorithm for k-vertex cover.
- Definitions of parameterized complexity, FPT and XP.
- Examples of problems in FPT, XP but not FPT, and not XP.
- Next time: Approximation algorithms

Summary

Todays topics were "Exact exponential algorithms" and "Parameterized Complexity". We have covered

- The natural brute force algorithm for problems in NP. $\surd$
- An exact $\mathcal{O}^\star(2^n)$-time dynamic programming algorithm for TSP.
- An exact $\mathcal{O}^\star(3^{n/3})$-time branching algorithm for MIS.
- A kernelization for the "Bar Fight Prevention" problem, a.k.a. k-vertex cover.
- A bounded search tree algorithm for k-vertex cover.
- Definitions of parameterized complexity, FPT and XP.
- Examples of problems in FPT, XP but not FPT, and not XP.
- Next time: Approximation algorithms

Summary

Todays topics were "Exact exponential algorithms" and "Parameterized Complexity". We have covered

- The natural brute force algorithm for problems in NP.
- An exact $\mathcal{O}^\star(2^n)$-time dynamic programming algorithm for TSP.$\surd$
- An exact $\mathcal{O}^\star(3^{n/3})$-time branching algorithm for MIS.
- A kernelization for the "Bar Fight Prevention" problem, a.k.a. k-vertex cover.
- A bounded search tree algorithm for k-vertex cover.
- Definitions of parameterized complexity, FPT and XP.
- Examples of problems in FPT, XP but not FPT, and not XP.
- Next time: Approximation algorithms

Summary

Todays topics were "Exact exponential algorithms" and "Parameterized Complexity". We have covered

- The natural brute force algorithm for problems in NP.
- An exact $\mathcal{O}^\star(2^n)$-time dynamic programming algorithm for TSP.
- An exact $\mathcal{O}^\star(3^{n/3})$-time branching algorithm for MIS. $\surd$
- A kernelization for the "Bar Fight Prevention" problem, a.k.a. k-vertex cover.
- A bounded search tree algorithm for k-vertex cover.
- Definitions of parameterized complexity, FPT and XP.
- Examples of problems in FPT, XP but not FPT, and not XP.
- Next time: Approximation algorithms

Summary

Todays topics were “Exact exponential algorithms” and “Parameterized Complexity”. We have covered

- The natural brute force algorithm for problems in NP.
- An exact $\mathcal{O}^\star(2^n)$-time dynamic programming algorithm for TSP.
- An exact $\mathcal{O}^\star(3^{n/3})$-time branching algorithm for MIS.
- A kernelization for the “Bar Fight Prevention” problem, a.k.a. k-vertex cover.
- A bounded search tree algorithm for k-vertex cover.
- Definitions of parameterized complexity, FPT and XP.
- Examples of problems in FPT, XP but not FPT, and not XP.
- Next time: Approximation algorithms

Summary

Todays topics were "Exact exponential algorithms" and "Parameterized Complexity". We have covered

- The natural brute force algorithm for problems in NP.
- An exact $\mathcal{O}^\star(2^n)$-time dynamic programming algorithm for TSP. DP
- An exact $\mathcal{O}^\star(3^{n/3})$-time branching algorithm for MIS. 剪枝
- A kernelization for the "Bar Fight Prevention" problem, a.k.a. k-vertex cover. 去简单
- A bounded search tree algorithm for k-vertex cover.
- Definitions of parameterized complexity, FPT and XP.
- Examples of problems in FPT, XP but not FPT, and not XP.
- Next time: Approximation algorithms

Summary

Todays topics were "Exact exponential algorithms" and "Parameterized Complexity". We have covered

- The natural brute force algorithm for problems in NP.
- An exact $\mathcal{O}^\star(2^n)$-time dynamic programming algorithm for TSP.
- An exact $\mathcal{O}^\star(3^{n/3})$-time branching algorithm for MIS.
- A kernelization for the "Bar Fight Prevention" problem, a.k.a. k-vertex cover.
- A bounded search tree algorithm for k-vertex cover.
- Definitions of parameterized complexity, FPT and XP.
- Examples of problems in FPT, XP but not FPT, and not XP.
- Next time: Approximation algorithms

Summary

Todays topics were "Exact exponential algorithms" and "Parameterized Complexity". We have covered

- The natural brute force algorithm for problems in NP.
- An exact $\mathcal{O}^\star(2^n)$-time dynamic programming algorithm for TSP.
- An exact $\mathcal{O}^\star(3^{n/3})$-time branching algorithm for MIS.
- A kernelization for the "Bar Fight Prevention" problem, a.k.a. k-vertex cover.
- A bounded search tree algorithm for k-vertex cover.
- Definitions of parameterized complexity, FPT and XP.
- Examples of problems in FPT, XP but not FPT, and not XP.
- Next time: Approximation algorithms

Summary

Todays topics were "Exact exponential algorithms" and "Parameterized Complexity". We have covered

- The natural brute force algorithm for problems in NP.
- An exact $\mathcal{O}^\star(2^n)$-time dynamic programming algorithm for TSP.
- An exact $\mathcal{O}^\star(3^{n/3})$-time branching algorithm for MIS.
- A kernelization for the "Bar Fight Prevention" problem, a.k.a. k-vertex cover.
- A bounded search tree algorithm for k-vertex cover.
- Definitions of parameterized complexity, FPT and XP.
- Examples of problems in FPT, XP but not FPT, and not XP.
- Next time: Approximation algorithms