Exact Algorithms

Lecture

Monday 15:45 – 17:15 5056
Wednesday 11:45 – 13:15 Uhr AH VI

Excercise

Tuesday 15:45 – 17:15 Uhr

Joachim Kneis, Somnath Sikdar
E-Mail: {kneis,sikdar}@informatik.rwth-aachen.de
{6114, 6115}, E2, 1st floor, Ahornstrasse 55
http://tcs.rwth-aachen.de/teaching/Exakte_Algorithmen/WS2010

Exam

M.Sc. Computer Science

- Written exam at the end of the semester (2 hours)
- 1st exam: 14.2.2011
- 2nd exam: 23.03.2011
- Requirements: 50% points from weekly homework assignments

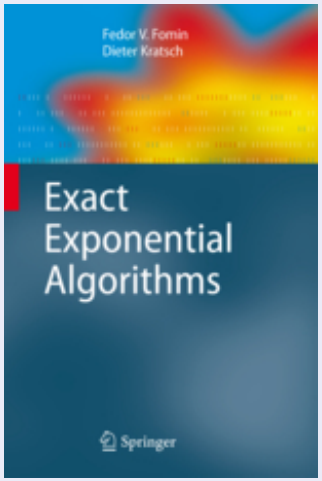
Diplom Computer Science

- Theoretische Informatik or
- Vertiefungsgebiet Effiziente Algorithmen

Everything else

?????

Literature



Exact Exponential Algorithms
Fedor V. Fomin, Dieter Kratsch
EATCS Series, Springer
ISBN: 978-3-642-16532-0
To appear in November

Introduction

Exact Algorithms

Given a hard problem, find an exact (optimal) solution in reasonable time.

Introduction

Exact Algorithms

Given a hard problem, find an exact (optimal) solution in reasonable time.

Hard problem: An NP-hard problem

Exact solution: Guaranteed to find the correct solution for decision problems or the optimal solution for optimization problems.

Reasonable time: Not polynomial, but as fast as possible. Typical exponential, sometimes subexponential

Moderately Exponential Time Algorithms

Introduction

Exact Algorithms

Given a hard problem, find an exact (optimal) solution in reasonable time.

Hard problem: An NP-hard problem

Exact solution: Guaranteed to find the correct solution for decision problems or the optimal solution for optimization problems.

Reasonable time: Not polynomial, but as fast as possible. Typical exponential, sometimes subexponential

Moderately Exponential Time Algorithms

Example (SAT)

Decide whether a formula in CNF with n variables has a satisfying assignment in time $O(2^n p(n))$

NP-complete Problems

Many important problems are NP-complete (or even harder).

Definition

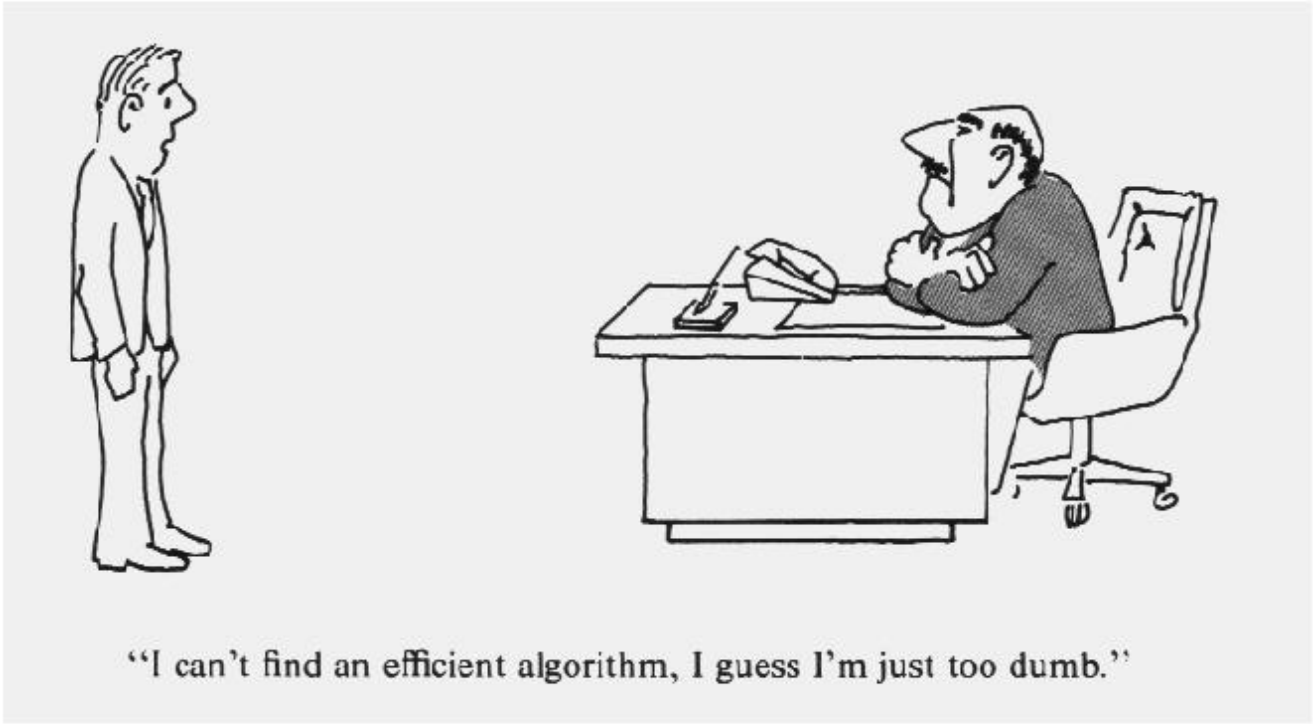
A *language* L is NP-complete, if

- $L \in NP$, i.e., can be decided by a nondeterminstic Turing machine in polynomial time
- Every language in NP can be reduced to L in polynomial time.

Theorem

If there exists a deterministic algorithm that decides a NP-complete problem in polynomial time, then $P = NP$.

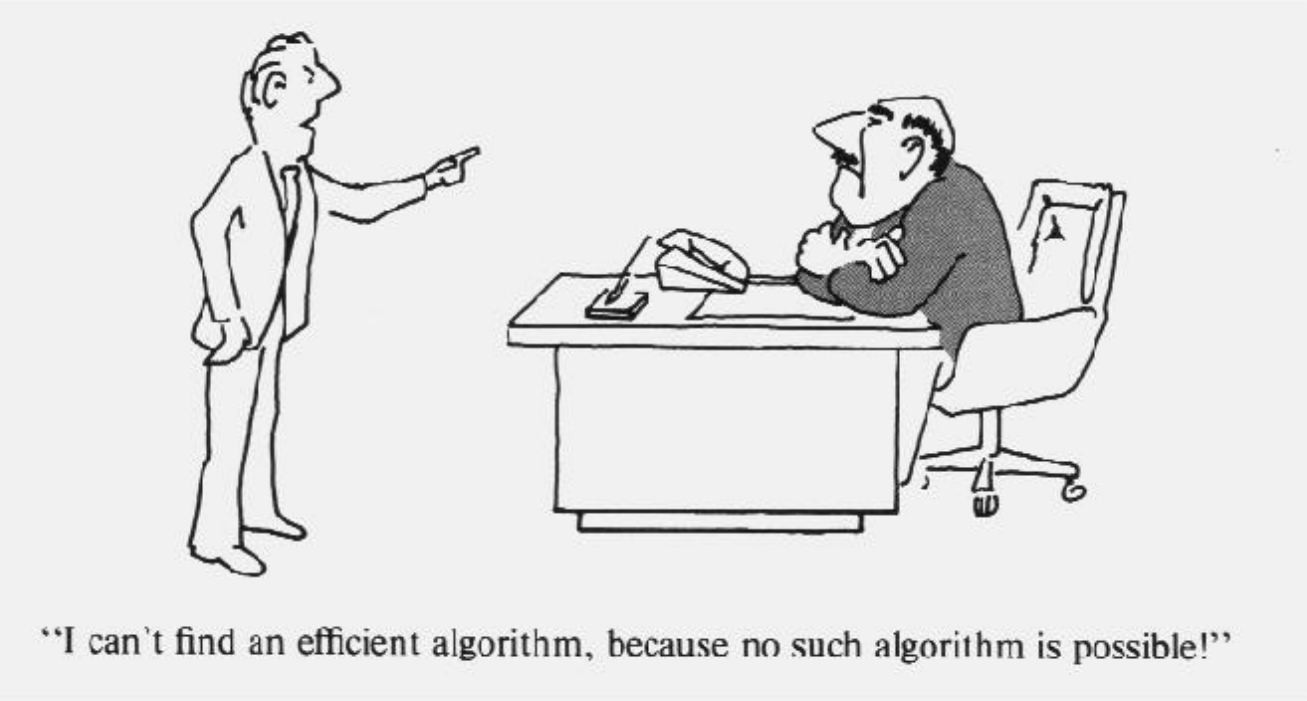
Luckily, it is not like . . .



"I can't find an efficient algorithm, I guess I'm just too dumb."

Garey and Johnson. Computers and Intractability.

But unfortunately it is neither . . .



"I can't find an efficient algorithm, because no such algorithm is possible!"

Garey and Johnson. Computers and Intractability.

Current state of the art

"I can't find an efficient algorithm, but neither can all these famous people."

Garey and Johnson. Computers and Intractability.

NP-complete Problems

Are all NP-complete problems hard to solve in practice?

NP-complete Problems

Are all NP-complete problems hard to solve in practice?

Why is SAT NP-hard?

Because every computation of a nondeterminstic Turing machine T can be simulated by a formula φ.

NP-complete Problems

Are all NP-complete problems hard to solve in practice?

Why is SAT NP-hard?

Because every computation of a nondeterminstic Turing machine T can be simulated by a formula φ.

φ has a satisfying assignment iff T accepts.

NP-complete Problems

Are all NP-complete problems hard to solve in practice?

Why is SAT NP-hard?

Because every computation of a nondeterminstic Turing machine T can be simulated by a formula φ.

φ has a satisfying assignment iff T accepts.

It is hard to decide whether a formula has a satisfying assignment for some formulas.

NP-complete Problems

Are all NP-complete problems hard to solve in practice?

Why is SAT NP-hard?

Because every computation of a nondeterminstic Turing machine T can be simulated by a formula φ.

φ has a satisfying assignment iff T accepts.

It is hard to decide whether a formula has a satisfying assignment for some formulas.

Most formulas are simpler!

NP-complete Problems

Are all NP-complete problems hard to solve in practice?

Why is SAT NP-hard?

Because every computation of a nondeterminstic Turing machine T can be simulated by a formula φ.

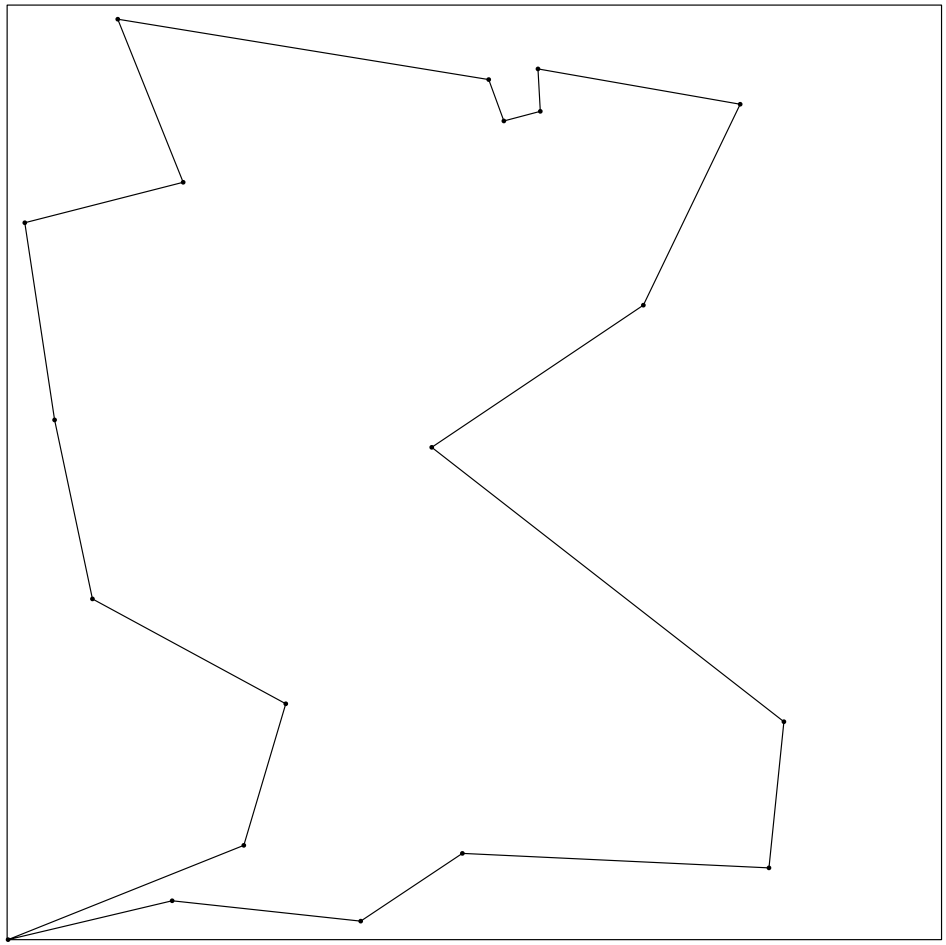
φ has a satisfying assignment iff T accepts.

It is hard to decide whether a formula has a satisfying assignment for some formulas.

Most formulas are simpler!

Alternative Methods? Do we need exponential time?

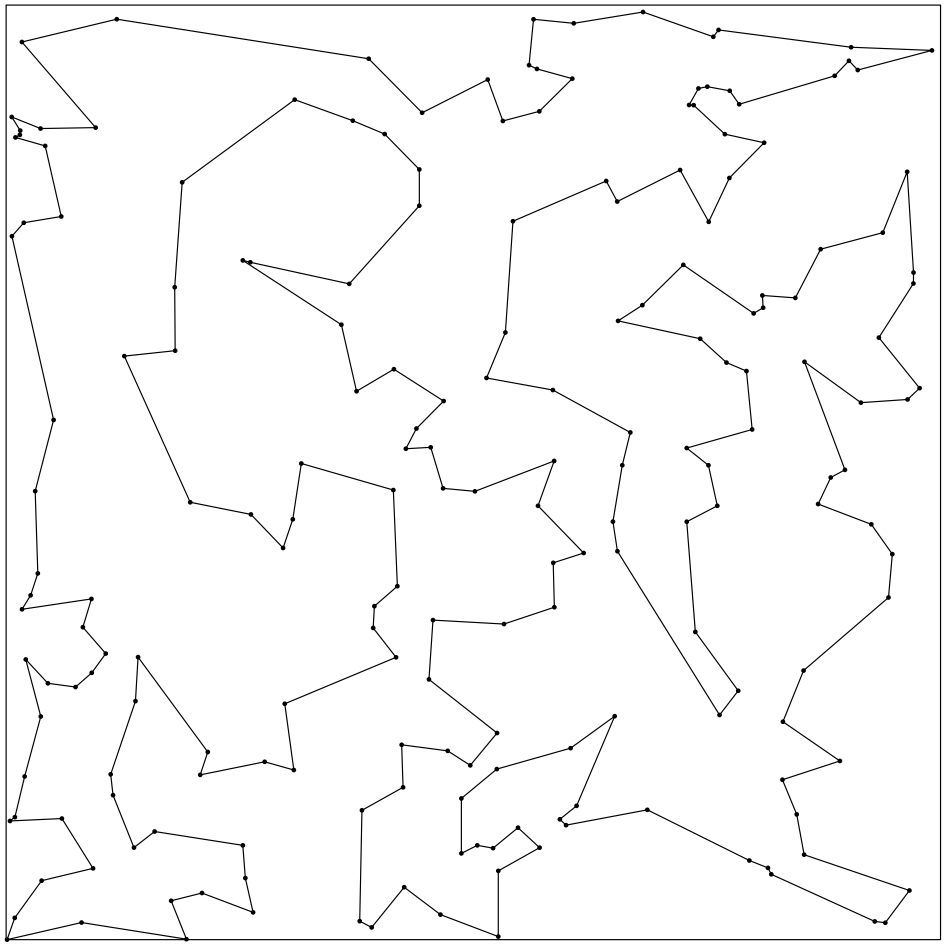
Heuristics: Traveling Salesman



Example

Simulated Annealing

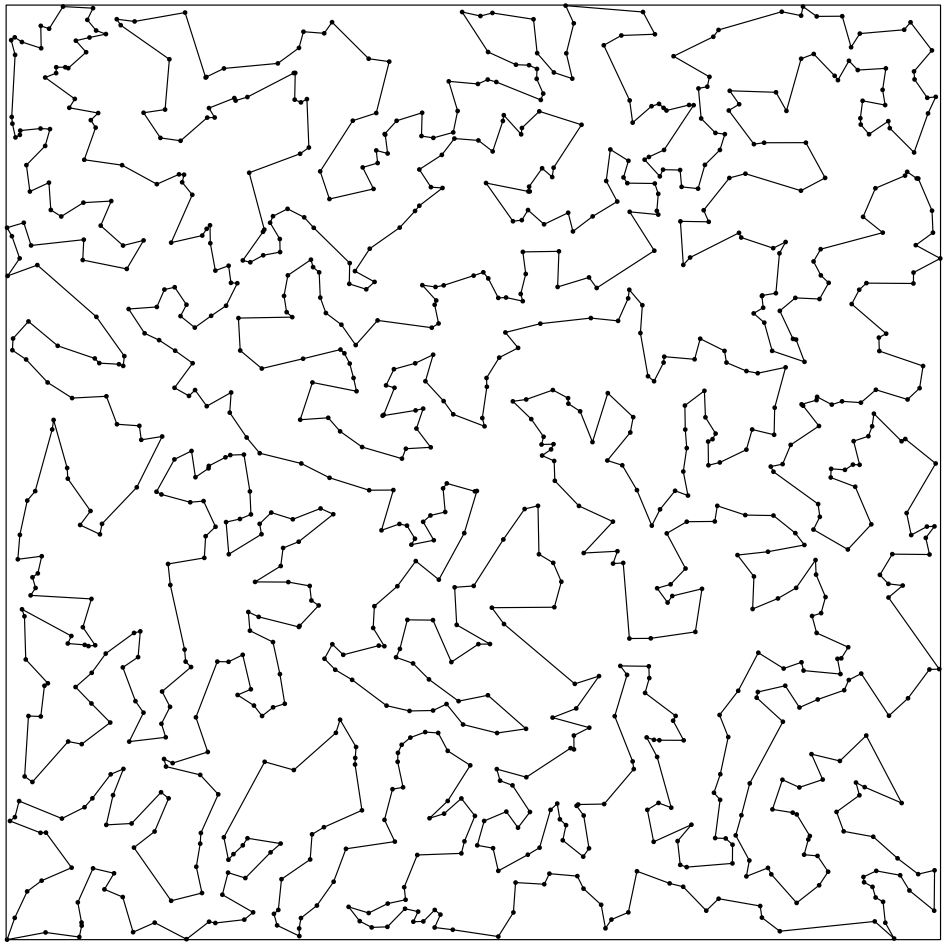
Heuristics: Traveling Salesman



Example

Simulated Annealing

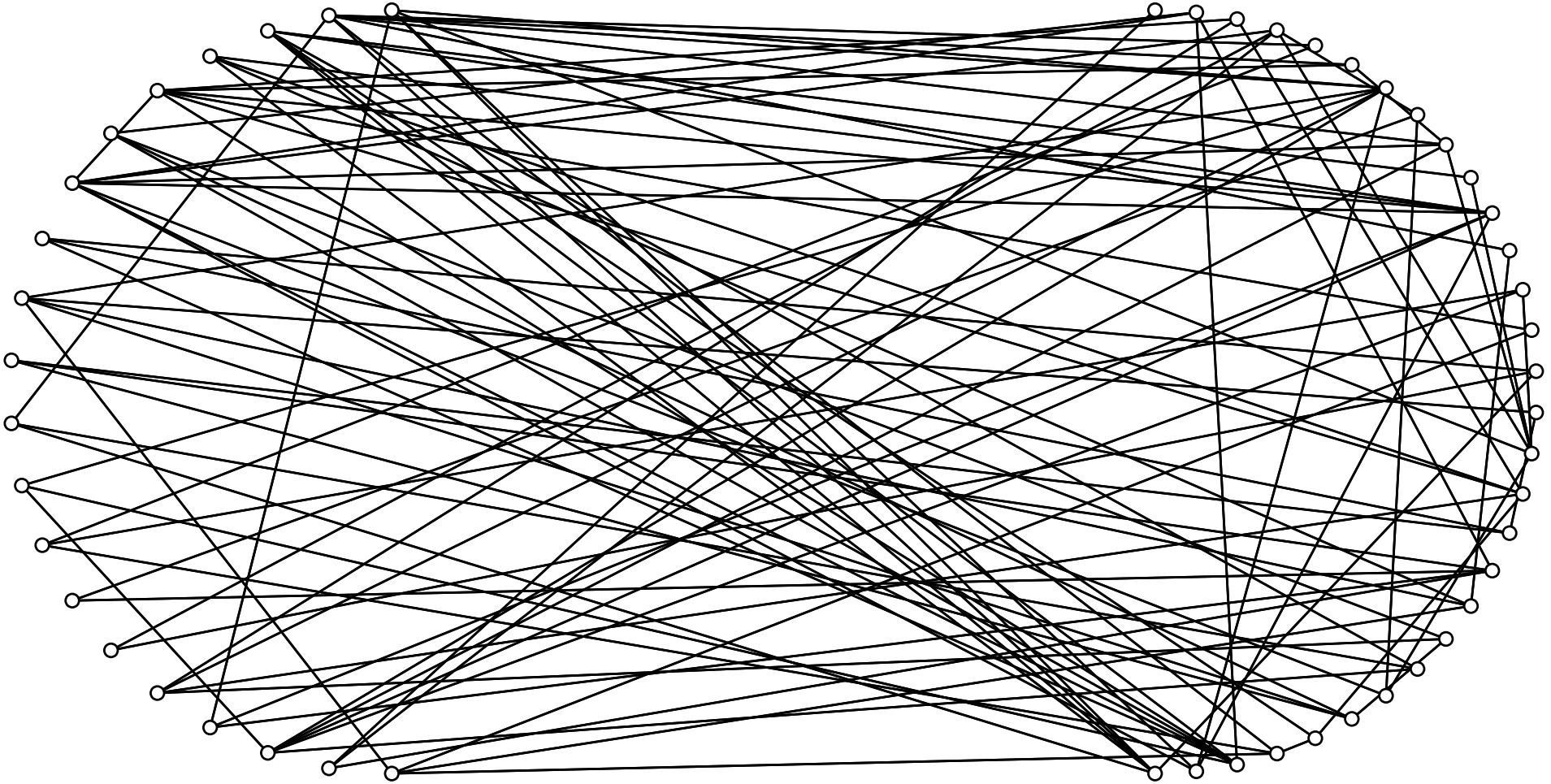
Heuristics: Traveling Salesman



Example

Simulated Annealing

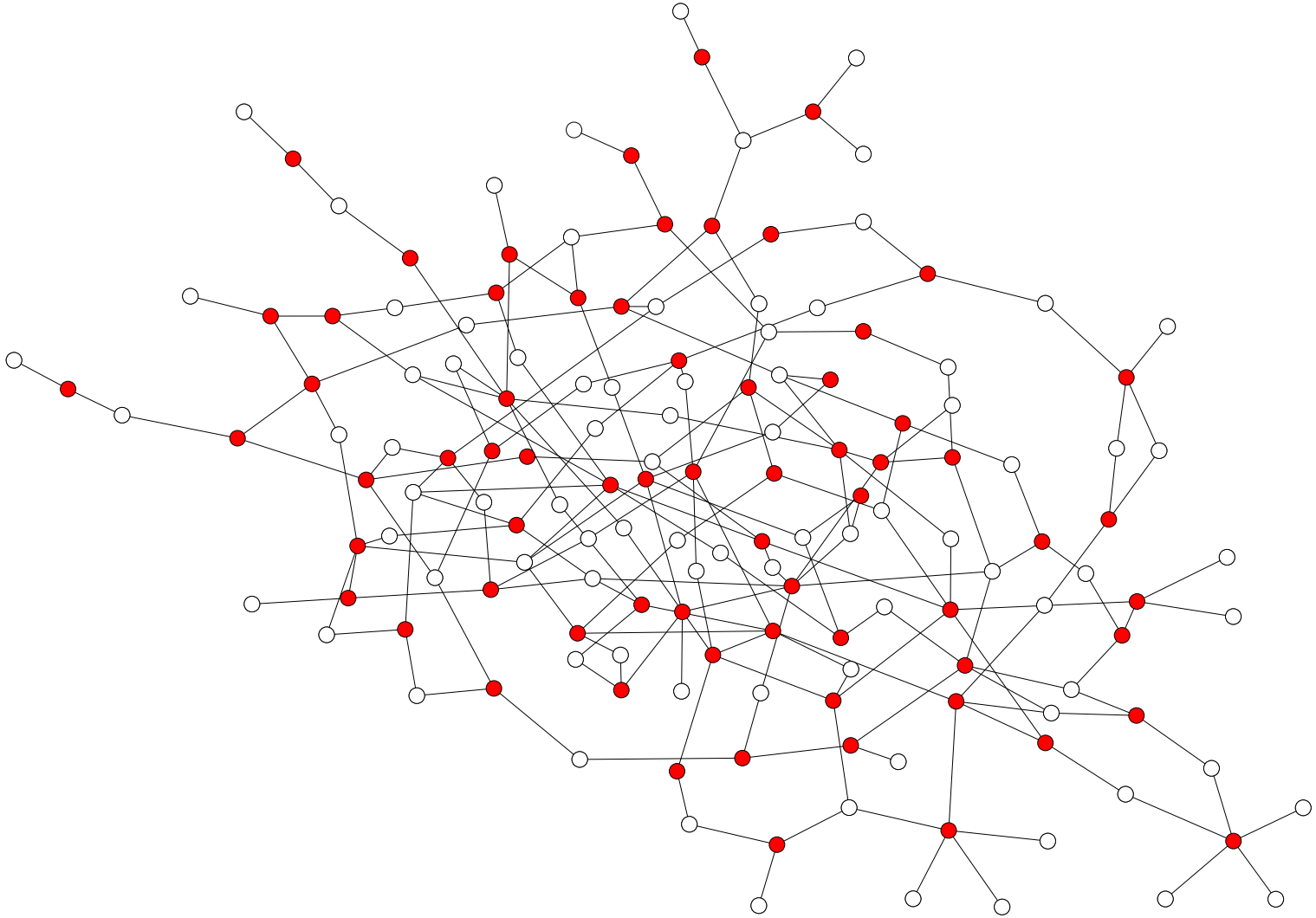
Approximation: MaxCut



Example

Greedy 2-Approximation (here: 76 out of 100 edges)

Parameterized Complexity: Vertex Cover



Example

$2^k \cdot poly(|V|)$ algorithm if parameter k is small

What else?

Something is missing?

What else?

Something is missing?

What about randomized algorithms?

What else?

Something is missing?

What about randomized algorithms?

Probably P=BPP (bounded-error probabilistic polynomial time)

- Speed-Up (e.g. Primality Testing)
- But no big Speed-Up

What else?

Something is missing?

What about randomized algorithms?

Probably P=BPP (bounded-error probabilistic polynomial time)

- Speed-Up (e.g. Primality Testing)
- But no big Speed-Up

For some problems, everything clever fails

What else?

Something is missing?

What about randomized algorithms?

Probably P=BPP (bounded-error probabilistic polynomial time)

- Speed-Up (e.g. Primality Testing)
- But no big Speed-Up

For some problems, everything clever fails

Example (Independent Set)

- No constant approximation
- No $f(k)poly(|V|)$-algorithm
- NP-hard on cubic graphs and on planar graphs

Are NP-hard problems really important?

Example

- Integer Multi-Commodity Flow in Railroad Networks
- Unsplittable Shortest Path Routing in Data Networks
- Shortest Superstring in DNA Sequences

Are NP-hard problems really important?

Example

- Integer Multi-Commodity Flow in Railroad Networks
- Unsplittable Shortest Path Routing in Data Networks
- Shortest Superstring in DNA Sequences

We need fast exact algorithms!

Are NP-hard problems really important?

Example

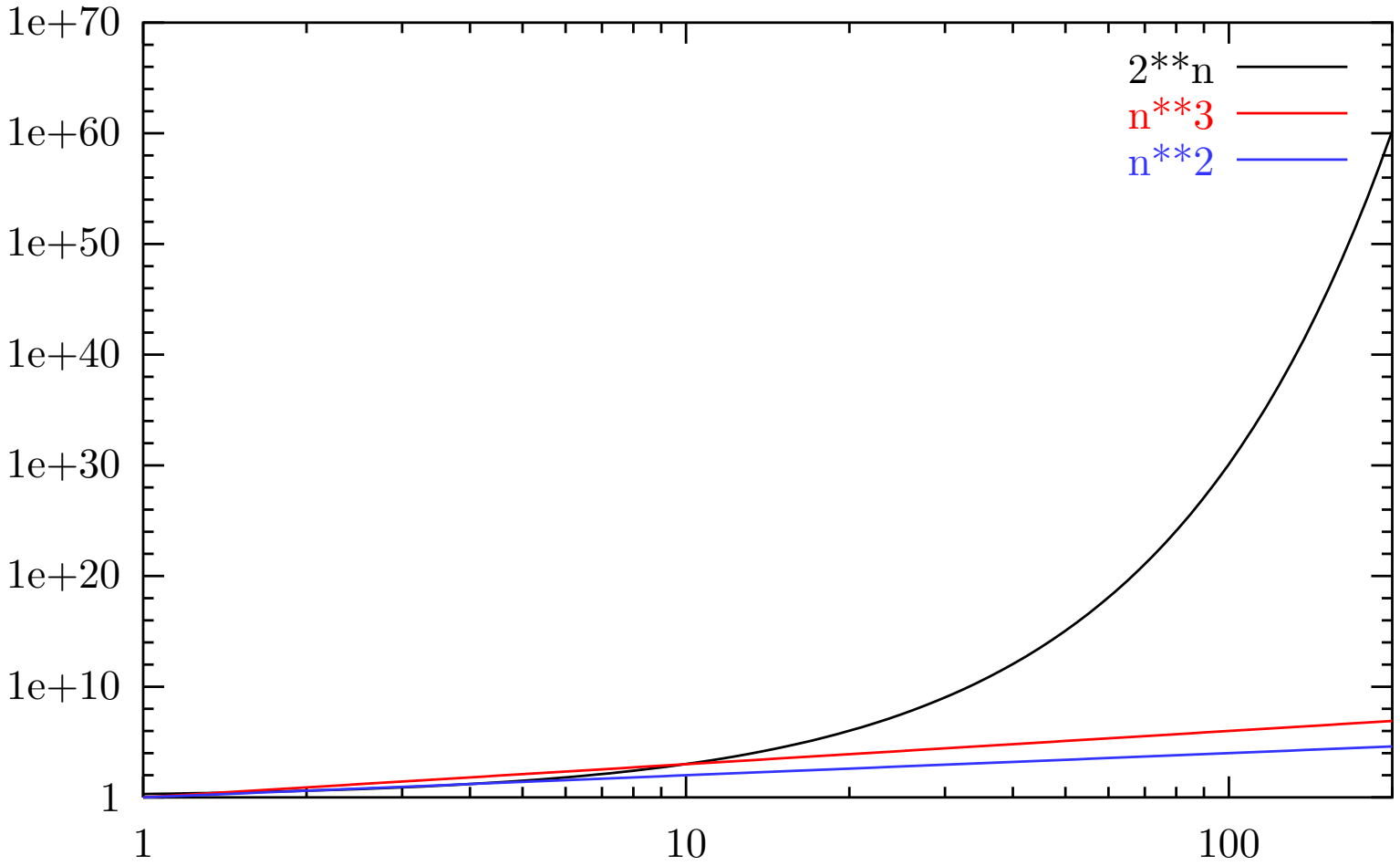
- Integer Multi-Commodity Flow in Railroad Networks
- Unsplittable Shortest Path Routing in Data Networks
- Shortest Superstring in DNA Sequences

We need fast exact algorithms!

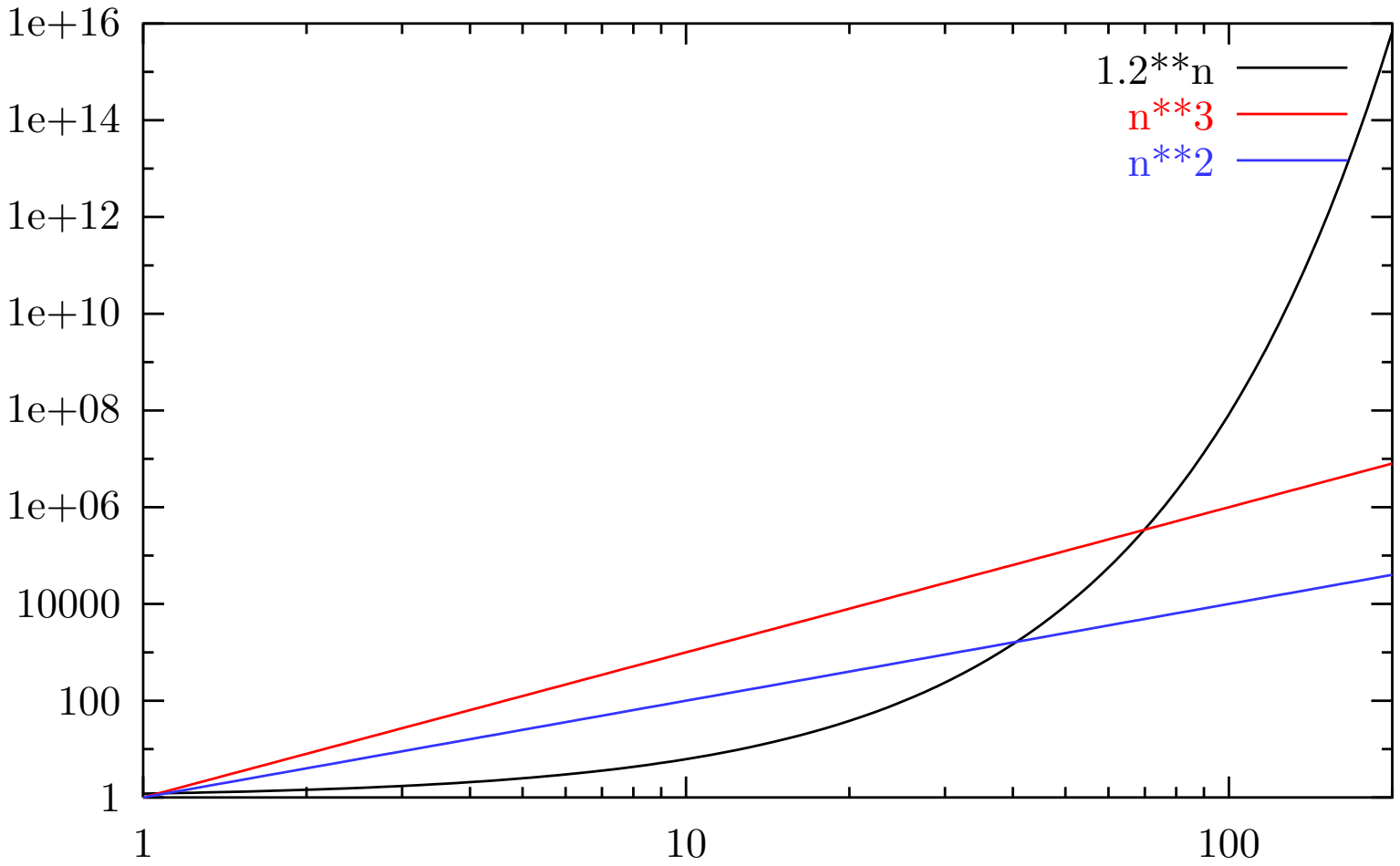
This lecture:

- Easier problems and simple algorithms
- General concepts and methods
- Theoretical background

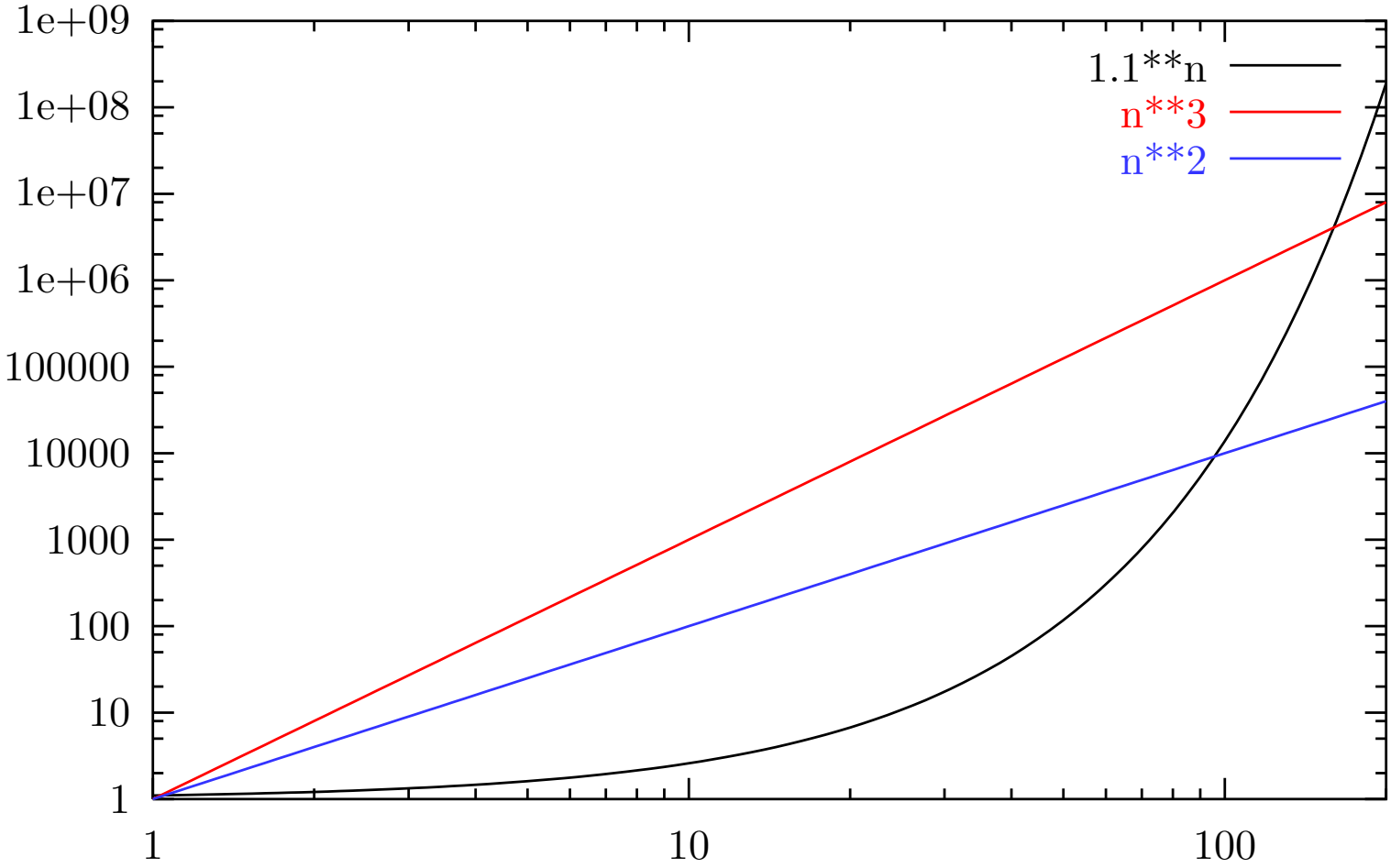
Runtime Comparison



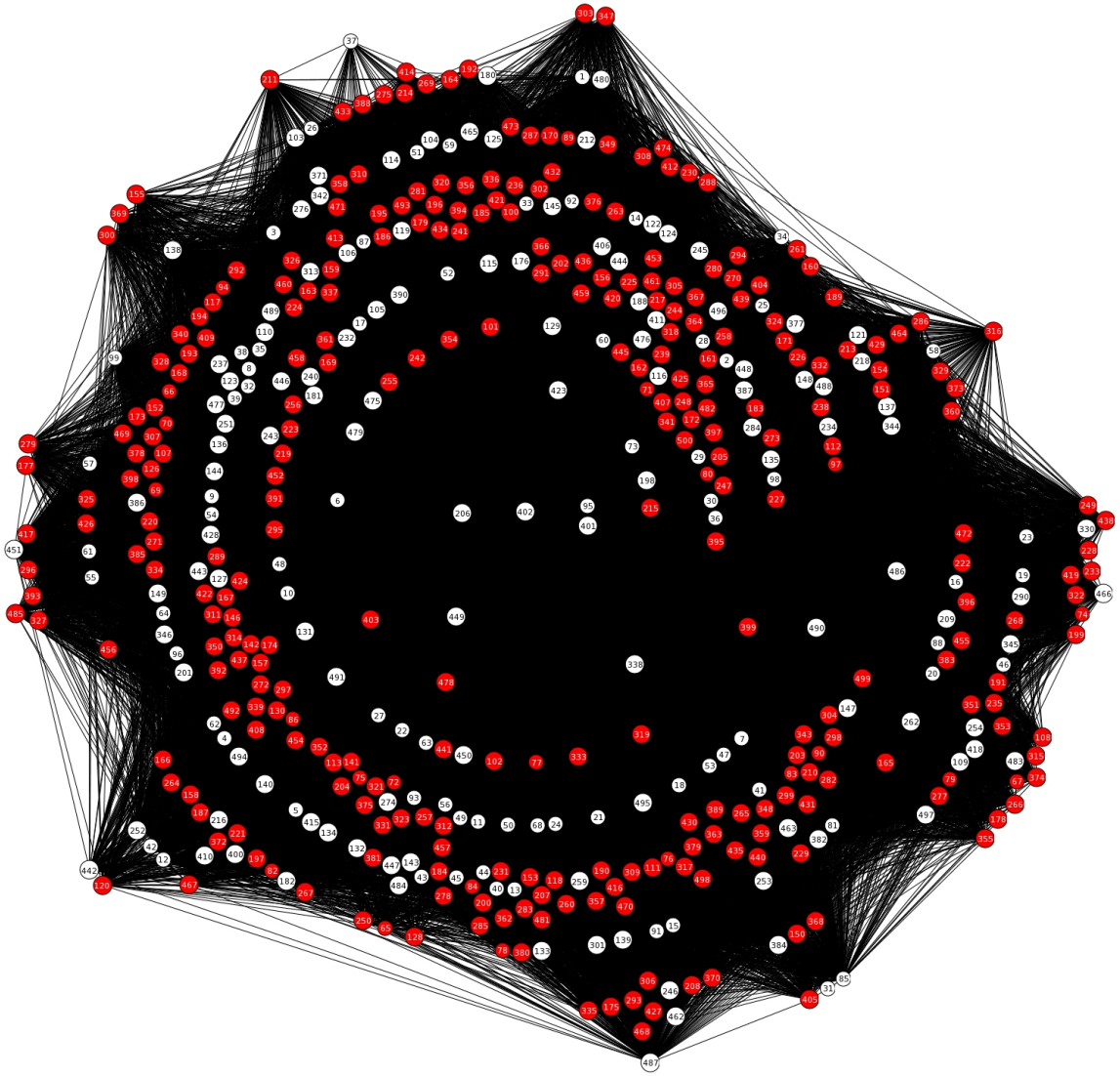
Runtime Comparison



Runtime Comparison



Independent Set in Chromosome Comparison



500	nodes
$\approx$ 50000	edges
306	Max IS
$\approx$ 5	minutes

1000	nodes
$\approx$ 200000	edges
426	Max IS
$\approx$ 40	minutes

Overview

1. Introduction
2. Branching
3. Measure&Conquer
4. Dynamic Programming
5. Local Search
6. Split & List
7. Lower Bounds
8. Inclusion-exclusion
9. Subset Convolution

Th O^*-Notation

Often: polynomial factors not that important

Definition

$f(n) \in O^*(g(n))$ if $f(n) \in O(g(n)poly(n))$

Example

- $n^2 2^n \in O^*(2^n)$
- $\sqrt{2}^n poly(n) \in O^*(1.42^n)$ because $(\sqrt{2} = 1.4142135623\ldots)$
- $\sqrt{2}^n poly(n) \in O(1.42^n)$

Th O^*-Notation

Often: polynomial factors not that important

Definition

$f(n) \in O^*(g(n))$ if $f(n) \in O(g(n)poly(n))$

Example

- $n^2 2^n \in O^*(2^n)$
- $\sqrt{2}^n poly(n) \in O^*(1.42^n)$ because $(\sqrt{2} = 1.4142135623\ldots)$

In practice: polynomial factors are important!

Example ($O(1.4^n n^5)$ versus $O(1.5^n n^2)$)

- $1.5^n n^2 \notin O^*(1.4^n)$
- $1.4^n n^5 \in O^*(1.5^n)$
- $1.4^n n^5 \geq 1.5^n n^2$ for $n \leq 230$
- $1.5^{230} 230^2 \geq 10^{45}$

Overview

1. Introduction
2. Branching
3. Measure&Conquer
4. Dynamic Programming
5. Local Search
6. Split & List
7. Lower Bounds
8. Inclusion-exclusion
9. Subset Convolution

What is Branching?

Idea

Given input instance I, compute other instances $I_1, \ldots I_k$ such that a solution to I can be computed from solutions for $I_1, \ldots I_k$.

- Construct $I_1, \ldots I_k$ efficiently
- $I_1, \ldots I_k$ are smaller than I
- Solve $I_1, \ldots I_k$ recursively
- Small instances can be solved efficiently

What is Branching?

Idea

Given input instance I, compute other instances $I_1, \ldots I_k$ such that a solution to I can be computed from solutions for $I_1, \ldots I_k$.

- Construct $I_1, \ldots I_k$ efficiently
- $I_1, \ldots I_k$ are smaller than I
- Solve $I_1, \ldots I_k$ recursively
- Small instances can be solved efficiently

Example (SAT)

Input: clauses $\{c_1, \ldots, c_m\}$ over variables $\{x_1, \ldots x_n\}$
Output: an assignment to the variables that satisfies each clause

What is Branching?

Idea

Given input instance I, compute other instances $I_1, \ldots I_k$ such that a solution to I can be computed from solutions for $I_1, \ldots I_k$.

- Construct $I_1, \ldots I_k$ efficiently
- $I_1, \ldots I_k$ are smaller than I
- Solve $I_1, \ldots I_k$ recursively
- Small instances can be solved efficiently

Simple branching algorithm

If $n = 0$, return whether an empty clause exists
I_1: remove all clauses that contain x_1 and remove $\overline{x_1}$ from all clauses
I_2: remove all clauses that contain $\overline{x_1}$ and remove x_1 from all clauses
If I_1 or I_2 has a solution, return "yes" otherwise "no"

Search Trees

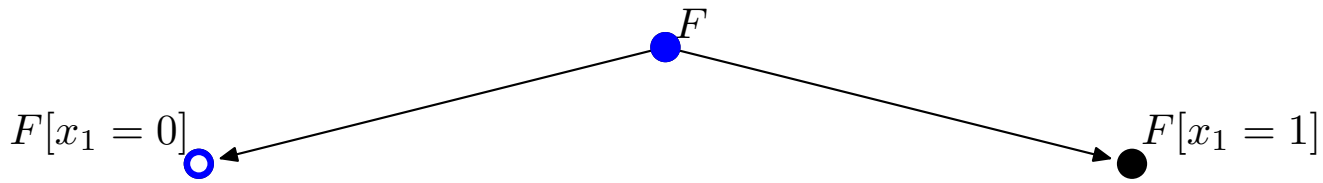
$$F = \{\overline{x_2}, x_3\}, \{x_1, x_2\}, \{\overline{x_1}, x_3\}, \{\overline{x_1}, x_2, x_3\}, \{x_1, \overline{x_3}\}$$

Run of a branching algorithm can be represented as a search tree:

Search Trees

$$F = \{\overline{x_2}, x_3\}, \{x_1, x_2\}, \{\overline{x_1}, x_3\}, \{\overline{x_1}, x_2, x_3\}, \{x_1, \overline{x_3}\}$$

Run of a branching algorithm can be represented as a search tree:

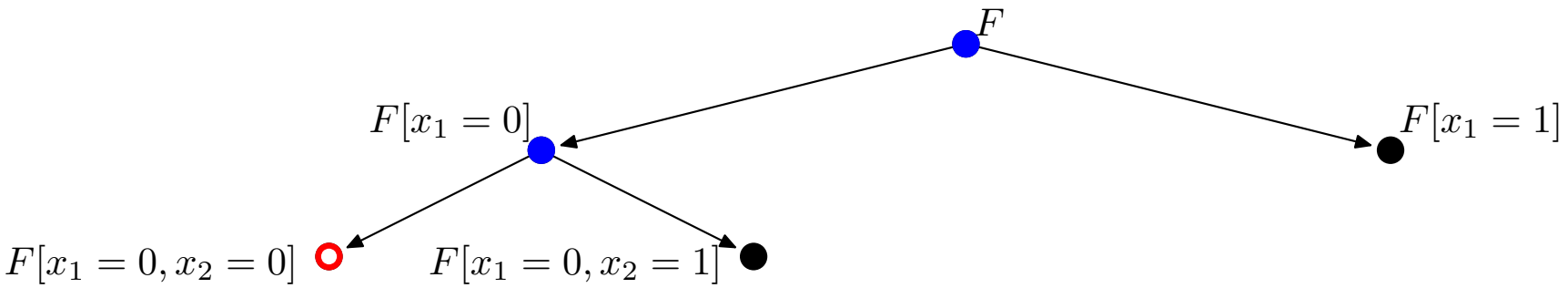


$$F[x_1 = 0] = \{\overline{x_2}, x_3\}, \{x_2\}, \{\overline{x_3}\}$$

Search Trees

$$F = \{\overline{x_2}, x_3\}, \{x_1, x_2\}, \{\overline{x_1}, x_3\}, \{\overline{x_1}, x_2, x_3\}, \{x_1, \overline{x_3}\}$$

Run of a branching algorithm can be represented as a search tree:

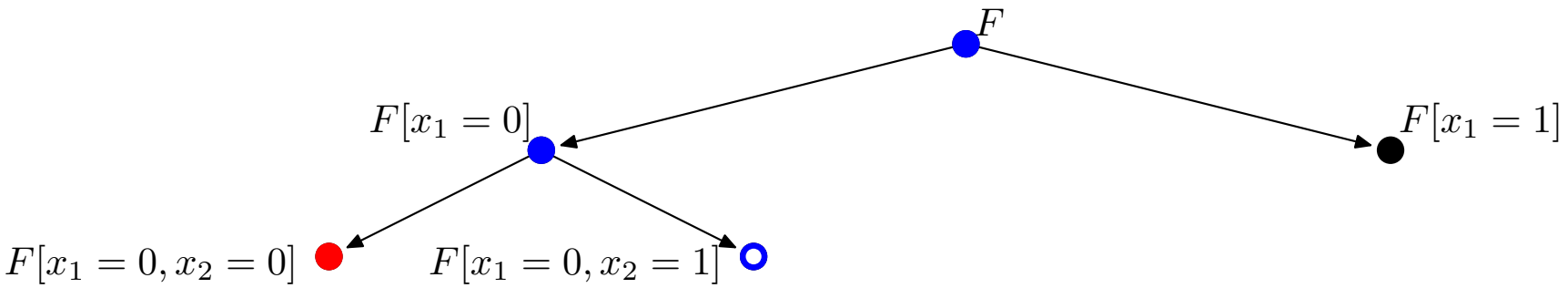


$$F[x_1 = 0, x_2 = 0] = \{\}, \{\overline{x_3}\}$$

Search Trees

$$F = \{\overline{x_2}, x_3\}, \{x_1, x_2\}, \{\overline{x_1}, x_3\}, \{\overline{x_1}, x_2, x_3\}, \{x_1, \overline{x_3}\}$$

Run of a branching algorithm can be represented as a search tree:

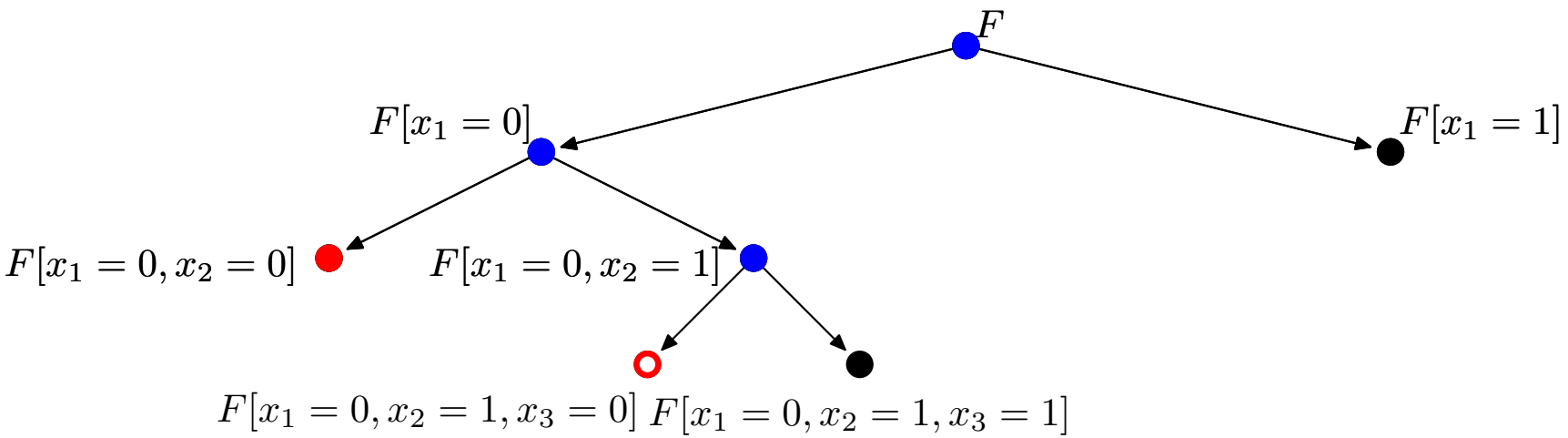


$$F[x_1 = 0, x_2 = 1] = \{x_3\}, \{\overline{x_3}\}$$

Search Trees

$$F = \{\overline{x_2}, x_3\}, \{x_1, x_2\}, \{\overline{x_1}, x_3\}, \{\overline{x_1}, x_2, x_3\}, \{x_1, \overline{x_3}\}$$

Run of a branching algorithm can be represented as a search tree:

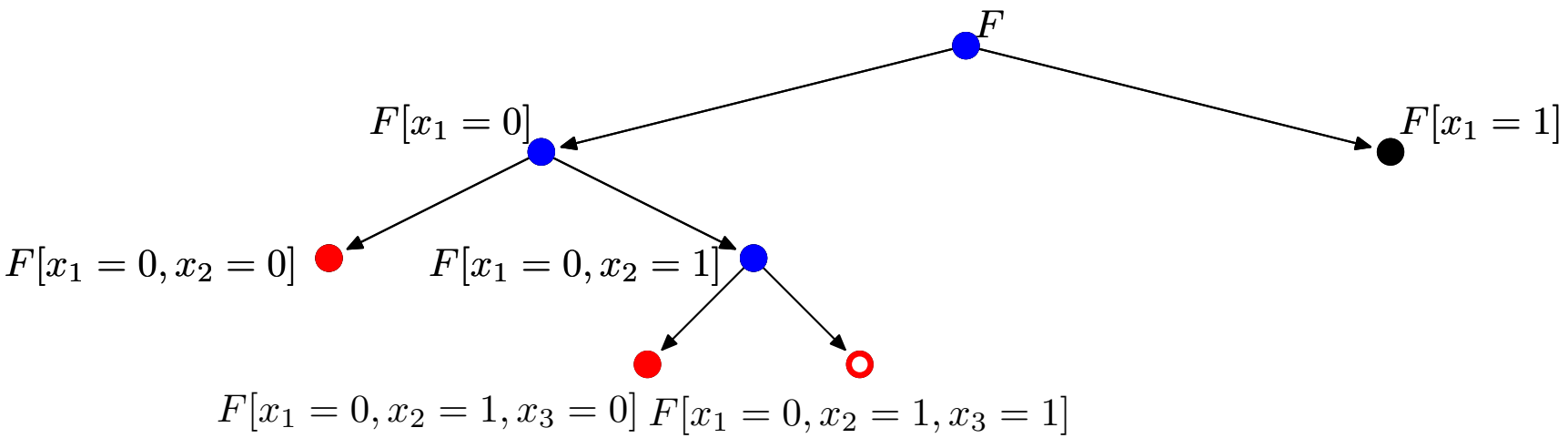


$$F[x_1 = 0, x_2 = 1, x_3 = 0] = \{\}$$

Search Trees

$$F = \{\overline{x_2}, x_3\}, \{x_1, x_2\}, \{\overline{x_1}, x_3\}, \{\overline{x_1}, x_2, x_3\}, \{x_1, \overline{x_3}\}$$

Run of a branching algorithm can be represented as a search tree:

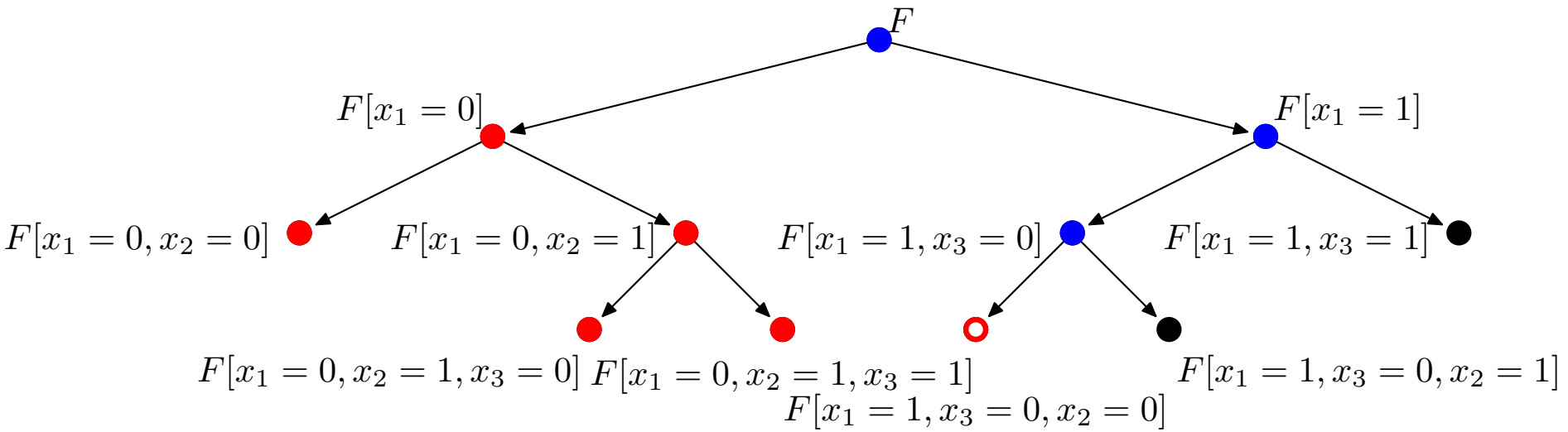


$$F[x_1 = 0, x_2 = 1, x_3 = 1] = \{\}$$

Search Trees

$$F = \{\overline{x_2}, x_3\}, \{x_1, x_2\}, \{\overline{x_1}, x_3\}, \{\overline{x_1}, x_2, x_3\}, \{x_1, \overline{x_3}\}$$

Run of a branching algorithm can be represented as a search tree:

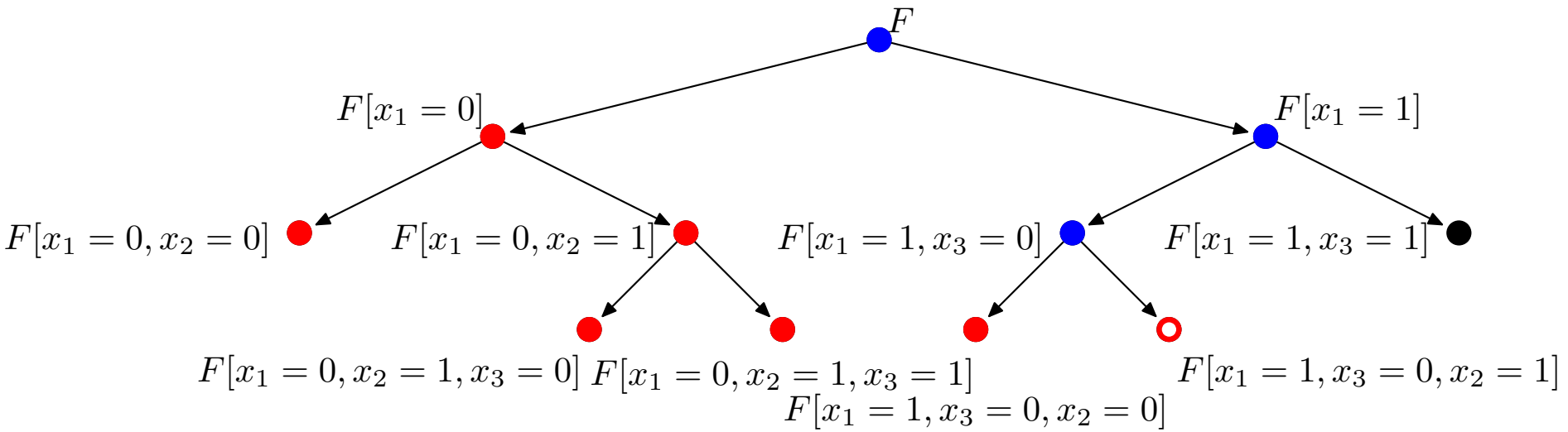


$$F[x_1 = 1, x_2 = 0, x_3 = 0] = \{\}, \{\}$$

Search Trees

$$F = \{\overline{x_2}, x_3\}, \{x_1, x_2\}, \{\overline{x_1}, x_3\}, \{\overline{x_1}, x_2, x_3\}, \{x_1, \overline{x_3}\}$$

Run of a branching algorithm can be represented as a search tree:

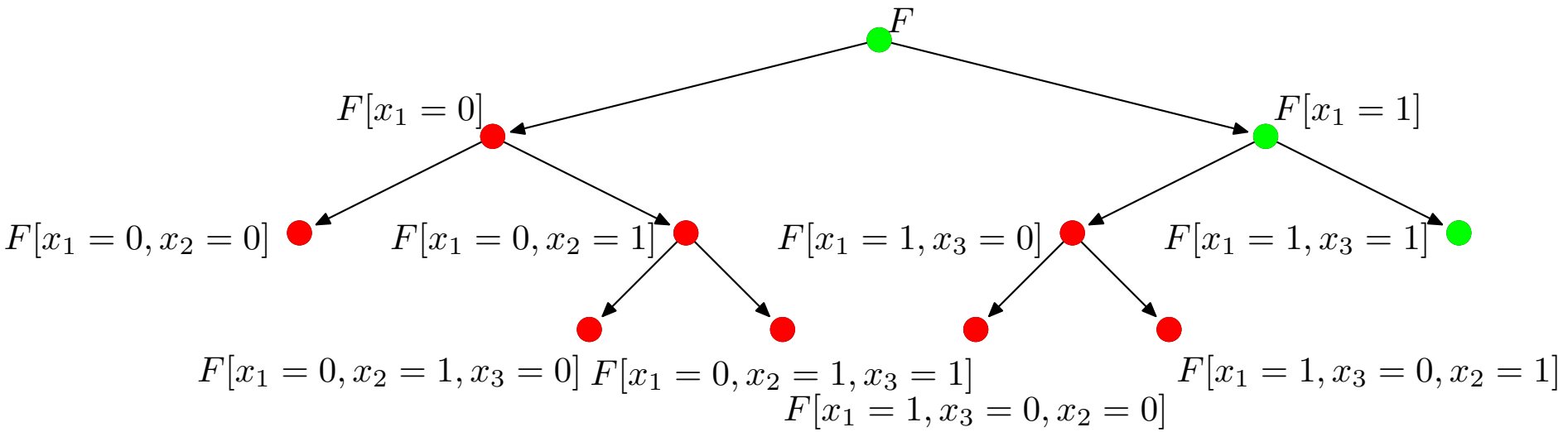


$$F[x_1 = 1, x_2 = 1, x_3 = 0] = \{\}, \{\}$$

Search Trees

$$F = \{\overline{x_2}, x_3\}, \{x_1, x_2\}, \{\overline{x_1}, x_3\}, \{\overline{x_1}, x_2, x_3\}, \{x_1, \overline{x_3}\}$$

Run of a branching algorithm can be represented as a search tree:

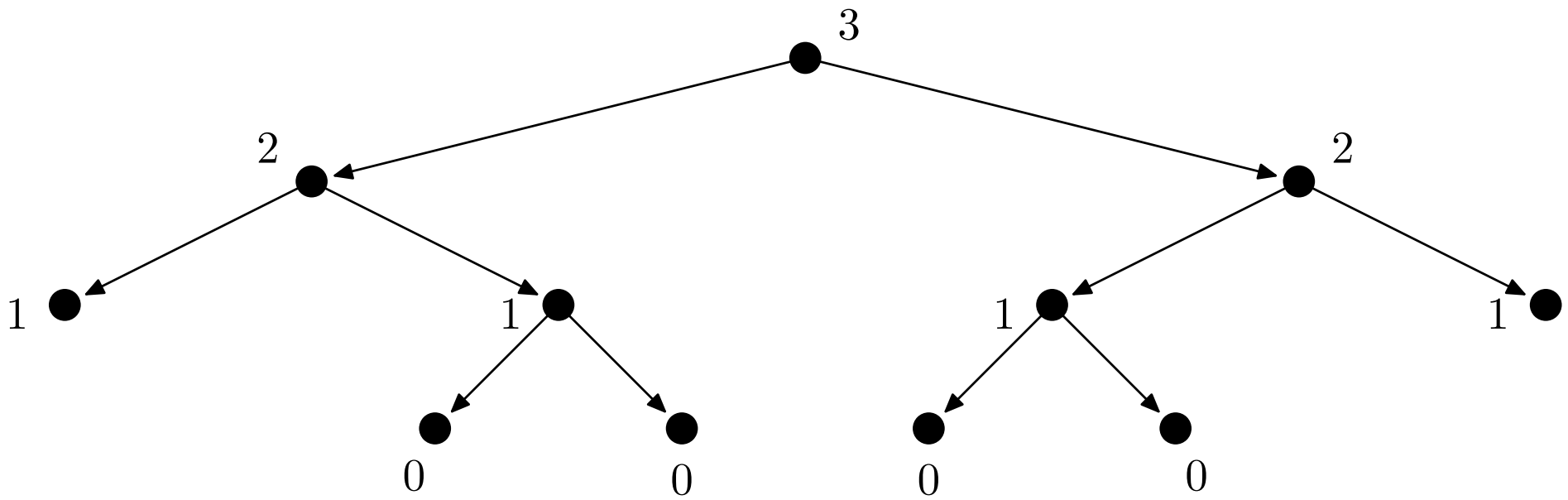


$$F[x_1 = 1, x_2 = 0, x_3 = 0] = \emptyset$$

Search Trees

$$F = \{\overline{x_2}, x_3\}, \{x_1, x_2\}, \{\overline{x_1}, x_3\}, \{\overline{x_1}, x_2, x_3\}, \{x_1, \overline{x_3}\}$$

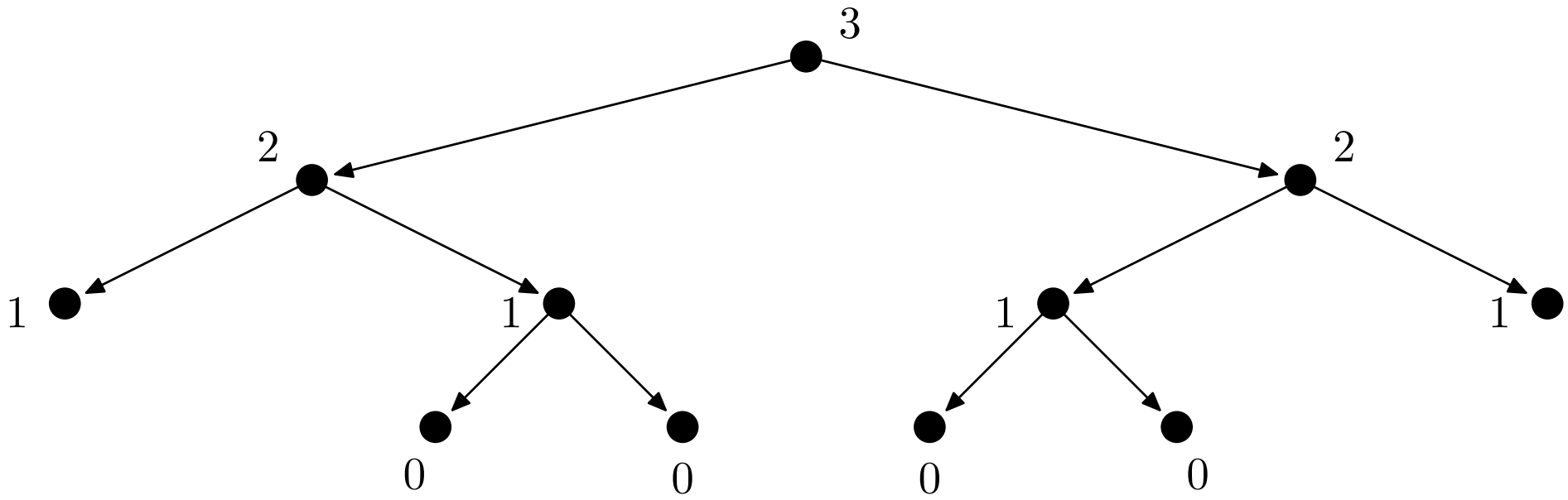
Run of a branching algorithm can be represented as a search tree:



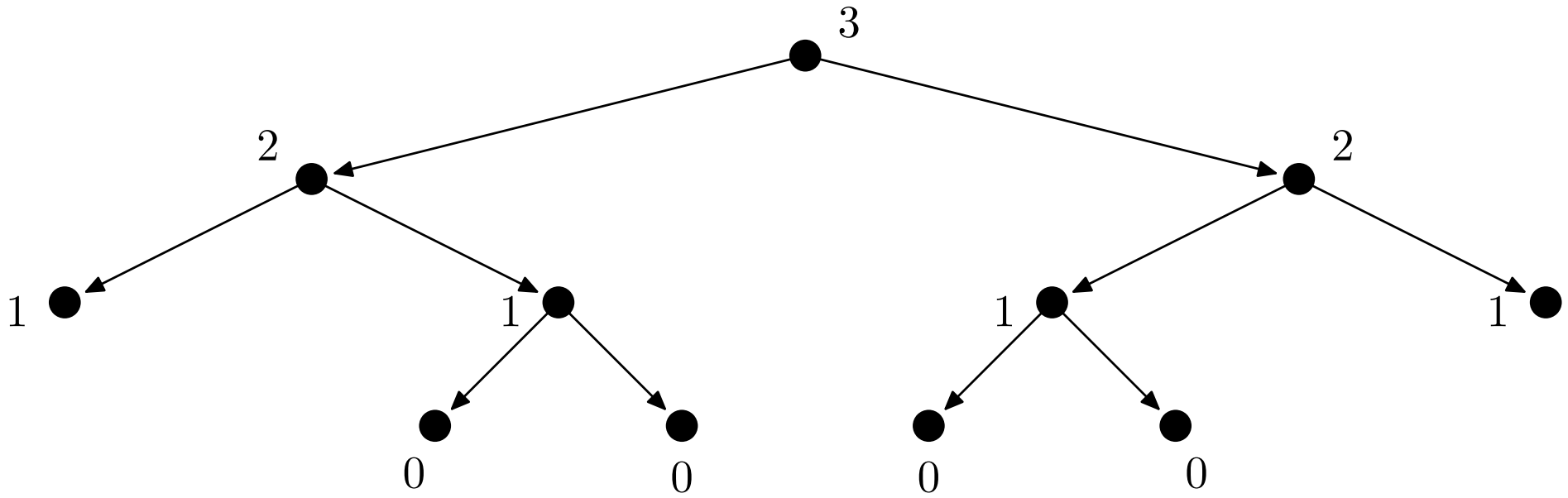
Bounded Search Trees

A Bounded Search Tree Algorithm needs to satisfy the following conditions for its search tree.

- Each node is labeled with a number at least 0
- The label of the root is a function of the input
- The number of children of each node is bounded by a function of the label
- The label of each children is smaller than the label of the parent



Example



- Each node is labeled with the number of variables $n \geq 0$
- The number of children of each node is bounded by 2
- The label of each children is by 1 smaller than the parent

Analysis of Bounded Search Trees

If

1. the root of a tree is labeled by n,
2. each node has at most two children,
3. all labels are positive,
4. the label of each child is by 1 smaller than those of the parent,

then the number of leaves in the bounded search tree is at most than 2^n.

Analysis of Bounded Search Trees

If

1. the root of a tree is labeled by n,
2. each node has at most two children,
3. all labels are positive,
4. the label of each child is by 1 smaller than those of the parent,

then the number of leaves in the bounded search tree is at most than 2^n.

How can this result be generalized?

Branching Vectors

If each node has two children and the labels of these children are by one smaller, we obtain the recurrence

$$B_n = B_{n-1} + B_{n-1}.$$

The corresponding Branching Vector is $(1, 1)$.
For a recurrence

$$B_n = B_{n-z_1} + B_{n-z_2} + \cdots + B_{n-z_m}$$

the corresponding Branching Vector is $(z_1, \ldots, z_m)$.

Branching Vectors can be used to describe Bounded Search Trees

Linear Recurrences with constant Coefficients

Definition

Let $c_1, \ldots, c_t \in \mathbb{R}$. A recurrence

$$A_n = c_1 A_{n-1} + c_2 A_{n-2} + \cdots + c_t A_{n-t} \text{ for } n \geq t.$$

is called *linear recurrence with constant coefficients*.

Here even $c_1, \ldots, c_t \in \mathbb{R}^{\geq 0}$

A linear recurrence with constant coefficients describes a set of concrete sequences.

Example ($a_n = a_{n-1} + a_{n-2}$)

$a_1 = 1$	$a_0 = 0$	$0, 1, 1, 2, 3, 5, 8, \ldots$
$a_1 = 1$	$a_0 = 1$	$1, 1, 2, 3, 5, 8, 13, \ldots$
$a_1 = 37$	$a_0 = 5$	$5, 37, 42, 79, 121, \ldots$

The Characteristic Polynomial

We need a method to solve such recurrences

The Characteristic Polynomial

We need a method to solve such recurrences

Definition

Let $A :=$

$$A_n = c_1 A_{n-1} + c_2 A_{n-2} + \cdots + c_t A_{n-t} \text{ for } n \geq t$$

be a linear recurrence with constant coefficients. The corresponding characteristic polynomial is

$$\chi(z) = z^t - c_1 z^{t-1} - c_2 z^{t-2} - \cdots - c_{t-1} z - c_t.$$

The Characteristic Polynomial

We need a method to solve such recurrences

Definition

Let $A :=$

$$A_n = c_1 A_{n-1} + c_2 A_{n-2} + \cdots + c_t A_{n-t} \text{ for } n \geq t$$

be a linear recurrence with constant coefficients. The corresponding characteristic polynomial is

$$\chi(z) = z^t - c_1 z^{t-1} - c_2 z^{t-2} - \cdots - c_{t-1} z - c_t.$$

Idea

- Roots of the characteristic polynomial can be used to solve the recurrence.
- Roots are complex numbers.

Linear Recurrences with constant Coefficients

Definition

Let $A := A_n = c_1 A_{n-1} + c_2 A_{n-2} + \cdots + c_t A_{n-t}$ for $n \geq t$ be a linear recurrence with constant coefficients. A solution of A is an infinite sequence $s_0, s_1, s_2, \ldots$ such that

$$s_n = c_1 s_{n-1} + c_2 s_{n-2} + \cdots + c_t s_{n-t} \text{ for all } n \geq t.$$

Linear Recurrences with constant Coefficients

Definition

Let $A := A_n = c_1 A_{n-1} + c_2 A_{n-2} + \cdots + c_t A_{n-t}$ for $n \geq t$ be a linear recurrence with constant coefficients. A solution of A is an infinite sequence $s_0, s_1, s_2, \ldots$ such that

$$s_n = c_1 s_{n-1} + c_2 s_{n-2} + \cdots + c_t s_{n-t} \text{ for all } n \geq t.$$

- The set of all sequences form a vector space $\mathcal{V}$ of infinite dimension.
- The set of all solutions of a recurrence A is a subspace of $\mathcal{V}$.

Linear Recurrences with constant Coefficients

Definition

Let $A := A_n = c_1 A_{n-1} + c_2 A_{n-2} + \cdots + c_t A_{n-t}$ for $n \geq t$ be a linear recurrence with constant coefficients. A solution of A is an infinite sequence $s_0, s_1, s_2, \ldots$ such that

$$s_n = c_1 s_{n-1} + c_2 s_{n-2} + \cdots + c_t s_{n-t} \text{ for all } n \geq t.$$

- The set of all sequences form a vector space $\mathcal{V}$ of infinite dimension.
- The set of all solutions of a recurrence A is a subspace of $\mathcal{V}$.

Let $A := A_n = c_1 A_{n-1} + c_2 A_{n-2} + \cdots + c_t A_{n-t}$ for $n \geq t$ be a linear recurrence with constant coefficients. The dimension of the solution subspace of A is t.

Linear Recurrences with constant Coefficients

Let us first assume that $a_n = \alpha^n$ with $\alpha \in \mathbb{C}$ is a solution to a given recurrence. For $n = t$, we obtain

$$\alpha^t = c_1\alpha^{t-1} + c_2\alpha^{t-2} + \cdots + c_{t-1}\alpha + c_t$$

which implies that α is a root of the characteristic polynomial

$$\chi(z) = z^t - c_1 z^{t-1} - c_2 z^{t-2} - \cdots - c_{t-1}z - c_t.$$

Linear Recurrences with constant Coefficients

On the other hand, if α is a root of

$$\chi(z) = z^t - c_1 z^{t-1} - c_2 z^{t-2} - \cdots - c_{t-1} z - c_t,$$

then $a_n = \alpha^n$ is also a solution to the recurrence.
To see this, we simply insert a_n into the recurrence and obtain

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_t a_{n-t}.$$

Linear Recurrences with constant Coefficients

If α is k-multiple root of χ, then $a_n = n^j \alpha^n$ for $0 \leq j < k$ is also a solution to the recurrence. We verify this again by inserting into the recurrence

$$n^j \alpha^n = \sum_{r=1}^{t} c_r (n-r)^j \alpha^{n-r} \Leftrightarrow n^j \alpha^t - \sum_{r=1}^{t} c_r (n-r)^j \alpha^{t-r} = 0.$$

Linear Recurrences with constant Coefficients

If α is k-multiple root of χ, then $a_n = n^j \alpha^n$ for $0 \leq j < k$ is also a solution to the recurrence. We verify this again by inserting into the recurrence

$$n^j \alpha^n = \sum_{r=1}^{t} c_r (n-r)^j \alpha^{n-r} \Leftrightarrow n^j \alpha^t - \sum_{r=1}^{t} c_r (n-r)^j \alpha^{t-r} = 0.$$

$$\begin{aligned}
\chi(x) &= x^t - c_1 z^{t-1} - c_2 x^{t-2} - \cdots - c_{t-1} x - c_t \\
\chi'(x) &= t x^{t-1} - (t-1) c_1 z^{t-2} - (t-2) c_2 x^{t-3} - \cdots - c_{t-1} \\
\chi''(x) &= t(t-1) x^{t-2} - (t-1)(t-2) c_1 z^{t-2} - (t-2)(t-3) c_2 x^{t-4} \\
&\quad - \cdots - (t-1) c_{t-2}
\end{aligned}$$

Linear Recurrences with constant Coefficients

If α is k-multiple root of χ, then $a_n = n^j\alpha^n$ for $0 \leq j < k$ is also a solution to the recurrence. We verify this again by inserting into the recurrence

$$n^j\alpha^n = \sum_{r=1}^{t} c_r(n-r)^j\alpha^{n-r} \Leftrightarrow n^j\alpha^t - \sum_{r=1}^{t} c_r(n-r)^j\alpha^{t-r} = 0.$$

$$\begin{aligned}
\chi(x) &= x^t - c_1z^{t-1} - c_2x^{t-2} - \cdots - c_{t-1}x - c_t \\
\chi'(x) &= tx^{t-1} - (t-1)c_1z^{t-2} - (t-2)c_2x^{t-3} - \cdots - c_{t-1} \\
\chi''(x) &= t(t-1)x^{t-2} - (t-1)(t-2)c_1z^{t-2} - (t-2)(t-3)c_2x^{t-4} \\
&\quad - \cdots - (t-1)c_{t-2}
\end{aligned}$$

Thus, $n^j\alpha^t - \sum_{r=1}^{t} c_r(n-r)^j\alpha^{t-r}$ is a linear combination of $\chi(\alpha)$, $\chi'(\alpha)$, $\chi''(\alpha)$, $\ldots$, $\chi^{(j)}(\alpha)$.

Linear Recurrences with constant Coefficients

If α is k-multiple root of χ, then $a_n = n^j\alpha^n$ for $0 \leq j < k$ is also a solution to the recurrence. We verify this again by inserting into the recurrence

$$n^j\alpha^n = \sum_{r=1}^{t} c_r(n-r)^j\alpha^{n-r} \Leftrightarrow n^j\alpha^t - \sum_{r=1}^{t} c_r(n-r)^j\alpha^{t-r} = 0.$$

$$\begin{aligned}
\chi(x) &= x^t - c_1x^{t-1} - c_2x^{t-2} - \cdots - c_{t-1}x - c_t \\
\chi'(x) &= tx^{t-1} - (t-1)c_1x^{t-2} - (t-2)c_2x^{t-3} - \cdots - c_{t-1} \\
\chi''(x) &= t(t-1)x^{t-2} - (t-1)(t-2)c_1x^{t-2} - (t-2)(t-3)c_2x^{t-4} \\
&\quad - \cdots - (t-1)c_{t-2}
\end{aligned}$$

Thus, $n^j\alpha^t - \sum_{r=1}^{t} c_r(n-r)^j\alpha^{t-r}$ is a linear combination of $\chi(\alpha)$, $\chi'(\alpha)$, $\chi''(\alpha)$, ..., $\chi^{(j)}(\alpha)$.

The first $k-1$ derivatives of χ are 0 at α, since α is a $k-$multiple root of χ.

Linear Recurrences with constant Coefficients

Theorem

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_t a_{n-t} \; for \, n \geq t$$

has the solution $a_n = n^j \alpha^n$, *for all roots* α *of the characteristic polynomial*

$$\chi(z) = z^t - c_1 z^{t-1} - c_2 z^{t-2} - \cdots - c_{t-1} z - c_t,$$

and all $j = 0, 1, \ldots, k-1$, *where* k *is the multiplicity of the root* α.

These solutions are linear independent and form a base of the solution subspace.

Linear Recurrences with constant Coefficients

Theorem

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_t a_{n-t} \; for \, n \geq t$$

has the solution $a_n = n^j \alpha^n$, *for all roots* α *of the characteristic polynomial*

$$\chi(z) = z^t - c_1 z^{t-1} - c_2 z^{t-2} - \cdots - c_{t-1} z - c_t,$$

and all $j = 0, 1, \ldots, k-1$, *where* k *is the multiplicity of the root* α.

These solutions are linear independent and form a base of the solution subspace.

Details $\Rightarrow$ Lecture Analysis of Algorithms

Linear Recurrences with constant Coefficients

Theorem

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_t a_{n-t} \ for\, n \geq t$$

has the solution $a_n = n^j \alpha^n$, *for all roots* α *of the characteristic polynomial*

$$\chi(z) = z^t - c_1 z^{t-1} - c_2 z^{t-2} - \cdots - c_{t-1} z - c_t,$$

and all $j = 0, 1, \ldots, k-1$, *where* k *is the multiplicity of the root* α.

These solutions are linear independent and form a base of the solution subspace.

Details $\Rightarrow$ Lecture Analysis of Algorithms

But what about concrete starting values?

The Size of Bounded Search Trees

Theorem

The size of a Bounded Search Tree with Branching Vector $(r_1, \ldots, r_m)$, *whose root is labeled with* n *is bounded by*

$$poly(n)\alpha^n,$$

where α *denotes the unique positive real root of the characteristic polynomial*

$$\chi(z) = z^t - z^{t-r_1} - z^{t-r_2} - \ldots - z^{t-r_m}.$$

Here, $t = \max\{r_1, \ldots, r_m\}$.

Details $\Rightarrow$ Lecture Analysis of Algorithms

We will call α branching number, also denoted by $\tau(r_1, \ldots, r_m)$.

The Simple Way

Theorem

Let $\mathcal{A}$ be an algorithm whose bounded search tree is described by the branching vector $(r_1, \ldots, r_m)$. Moreover, let $t = \max\{r_1, \ldots, r_m\}$ and let the size of each search tree whose root is labeled by at most t be 1. Then, the size of any search tree of $\mathcal{A}$ is bounded by

$$\alpha^n,$$

where α denotes the unique positive real root of the characteristic polynomial

$$\chi(z) = z^t - z^{t-r_1} - z^{t-r_2} - \ldots - z^{t-r_m}.$$

Proof:

Proof by induction over n. The size of the search tree is bounded by

$$T(n) \leq \sum_{i=1}^{t} T(n - r_i) \text{ for } n > t$$

and

$$T(n) = 1 \text{ for } 0 \leq n \leq t.$$

Since $\alpha^n \geq 1$ for all $0 \leq n \leq t$, the bound holds for all $n \leq t$.
Let $n > t$. Since α is a root of χ, we obtain

$$\sum_{i=1}^{t} T(n - r_i) \leq \sum_{i=1}^{t} \alpha^{n-r_i} \leq \alpha^n.$$

Proof:

Proof by induction over n. The size of the search tree is bounded by

$$T(n) \leq \sum_{i=1}^{t} T(n - r_i) \text{ for } n > t$$

and

$$T(n) = 1 \text{ for } 0 \leq n \leq t.$$

Since $\alpha^n \geq 1$ for all $0 \leq n \leq t$, the bound holds for all $n \leq t$.
Let $n > t$. Since α is a root of χ, we obtain

$$\sum_{i=1}^{t} T(n - r_i) \leq \sum_{i=1}^{t} \alpha^{n-r_i} \leq \alpha^n.$$

$T(n) = 1$ for all $n \leq t$ can be guaranteed by modifying the branching algorithm (at the cost of increased runtime in each call).

The Size of Bounded Search Trees

Example:
The Branching Vector $(1, 3)$ corresponds to the characteristic polynomial

$$z^3 - z^2 - 1.$$

The largest root is roughly 1.465571.
The size of the search tree is thus bounded by $O(1.465572^n)$.

The Size of Bounded Search Trees

Another example:
The Branching Vector $(1, 2, 2, 3, 6)$ corresponds to the characteristic polynomial

$$z^6 - z^5 - z^4 - z^4 - z^3 - 1.$$

The largest root is roughly 2.160912.
The size of the search tree is thus bounded by $O(2.160913^n)$.

Only one positive real root

Theorem

Let $(r_1, \ldots, r_m)$ be a branching vector and let $t = \max\{r_1, \ldots, r_m\}$. Then the characteristic polynomial

$$\chi(z) = z^t - z^{t-r_1} - z^{t-r_2} - \cdots - z^{t-r_m}.$$

has exactly one positive real root.

Only one positive real root

Theorem

Let $(r_1, \ldots, r_m)$ be a branching vector and let $t = \max\{r_1, \ldots, r_m\}$. Then the characteristic polynomial

$$\chi(z) = z^t - z^{t-r_1} - z^{t-r_2} - \cdots - z^{t-r_m}.$$

has exactly one positive real root.

Claim 1

$\chi(z) < 0$ for $z \in [0, 1)$ (and $\chi(1) \leq 0$)

$\chi(z)$ is of the form $z^t - \sum_{i=1}^{t} c_{t-i} z^{t-i}$ with $c_j \in \mathbb{N}$.

Only one positive real root

Theorem

Let $(r_1, \ldots, r_m)$ be a branching vector and let $t = \max\{r_1, \ldots, r_m\}$. Then the characteristic polynomial

$$\chi(z) = z^t - z^{t-r_1} - z^{t-r_2} - \cdots - z^{t-r_m}.$$

has exactly one positive real root.

Claim 1

$\chi(z) < 0$ for $z \in [0, 1)$ (and $\chi(1) \leq 0$)

$\chi(z)$ is of the form $z^t - c_0 - \sum_{i=1}^{t-1} c_{t-i} z^{t-i}$ with $c_j \in \mathbb{N}$.

Only one positive real root

Theorem

Let $(r_1, \ldots, r_m)$ be a branching vector and let $t = \max\{r_1, \ldots, r_m\}$. Then the characteristic polynomial

$$\chi(z) = z^t - z^{t-r_1} - z^{t-r_2} - \ldots - z^{t-r_m}.$$

has exactly one positive real root.

Claim 1

$\chi(z) < 0$ for $z \in [0, 1)$ (and $\chi(1) \leq 0$)

$\chi(z)$ is of the form $z^t - c_0 - \sum_{i=1}^{t-1} c_{t-i} z^{t-i}$ with $c_j \in \mathbb{N}$.

- $c_0 \geq 1$ by definition.
- $\sum_{i=1}^{t-1} c_{t-i} z^{t-i} \geq 0$
- $z^t < 1$ ($z^t = 1$ for $z = 1$)

Only one positive real root

Claim 2

$\chi(z) = 0$ for some $z \in [1, \infty)$.

Only one positive real root

Claim 2

$\chi(z) = 0$ for some $z \in [1, \infty)$.

$$\begin{aligned}
\sum_{i=1}^{t} c_{t-i} z^{t-i} &\leq \sum_{i=1}^{t} c_{t-i} z^{t-1} \\
&= z^{t-1} \sum_{i=1}^{t} c_{t-i} \\
&= c z^{t-1}
\end{aligned}$$

Only one positive real root

Claim 2

$\chi(z) = 0$ for some $z \in [1, \infty)$.

$$\begin{aligned}
\sum_{i=1}^{t} c_{t-i} z^{t-i} &\leq \sum_{i=1}^{t} c_{t-i} z^{t-1} \\
&= z^{t-1} \sum_{i=1}^{t} c_{t-i} \\
&= c z^{t-1}
\end{aligned}$$

$$z^t - c z^{t-1} \geq 0 \qquad \text{for } z \geq c$$

Only one positive real root

Claim 2

$\chi(z) = 0$ for some $z \in [1, \infty)$.

$$\begin{aligned}
\sum_{i=1}^{t} c_{t-i} z^{t-i} &\leq \sum_{i=1}^{t} c_{t-i} z^{t-1} \\
&= z^{t-1} \sum_{i=1}^{t} c_{t-i} \\
&= c z^{t-1}
\end{aligned}$$

$$z^t - c z^{t-1} \geq 0 \qquad \text{for } z \geq c$$

$\chi(z)$ is continuous, $\chi(1) \leq 0$, $\chi(c) \geq 0$.

Only one positive real root

Let $x \geq 1$ such that $\chi(x) \geq 0$:

$$x^t \geq \sum_{i=1}^{t} c_{t-i} x^{t-i} \geq 1 \qquad \text{Recall: } c_0 \geq 1, c_j \geq 0$$

Only one positive real root

Let $x \geq 1$ such that $\chi(x) \geq 0$:

$$x^t \geq \sum_{i=1}^{t} c_{t-i} x^{t-i} \geq 1 \qquad \text{Recall: } c_0 \geq 1, c_j \geq 0$$

$$\begin{aligned} tx^t &> (t-1) \sum_{i=1}^{t} c_{t-i} x^{t-i} \\ &> \sum_{i=1}^{t} (t-i) c_{t-i} x^{t-i} \end{aligned}$$

Only one positive real root

Let $x \geq 1$ such that $\chi(x) \geq 0$:

$$x^t \geq \sum_{i=1}^{t} c_{t-i} x^{t-i} \geq 1 \qquad \text{Recall: } c_0 \geq 1, c_j \geq 0$$

$$\begin{aligned} tx^t &> (t-1)\sum_{i=1}^{t} c_{t-i} x^{t-i} \\ &> \sum_{i=1}^{t} (t-i) c_{t-i} x^{t-i} \end{aligned}$$

$$\frac{1}{x} tx^t > \frac{1}{x} \sum_{i=1}^{t} (t-i) c_{t-i} x^{t-i}$$

Only one positive real root

Let $x \geq 1$ such that $\chi(x) \geq 0$:

$$x^t \geq \sum_{i=1}^{t} c_{t-i} x^{t-i} \geq 1 \qquad \text{Recall: } c_0 \geq 1, c_j \geq 0$$

$$\begin{aligned} tx^t &> (t-1)\sum_{i=1}^{t} c_{t-i} x^{t-i} \\ &> \sum_{i=1}^{t} (t-i) c_{t-i} x^{t-i} \end{aligned}$$

$$\frac{1}{x} tx^t > \frac{1}{x} \sum_{i=1}^{t} (t-i) c_{t-i} x^{t-i}$$

$$\chi'(x) > 0 \text{ if } \chi(x) \geq 0$$

Only one positive real root

Let α be the smallest positive real root of $\chi(z)$ and let $\beta > \alpha$ be a positive real root of $\chi(z)$ such that $\chi(x) > 0$ for all $\alpha < x < \beta$.

Then $\chi'(x) > 0$ for all $\alpha < x < \beta$.

Only one positive real root

Let α be the smallest positive real root of $\chi(z)$ and let $\beta > \alpha$ be a positive real root of $\chi(z)$ such that $\chi(x) > 0$ for all $\alpha < x < \beta$.

Then $\chi'(x) > 0$ for all $\alpha < x < \beta$.

By the Mean Value Theorem:

There exists $y \in (\alpha, \beta)$ such that

$$\chi'(y) = \frac{\chi(\beta) - \chi(\alpha)}{\beta - \alpha} = 0.$$

Only one positive real root

Let α be the smallest positive real root of $\chi(z)$ and let $\beta > \alpha$ be a positive real root of $\chi(z)$ such that $\chi(x) > 0$ for all $\alpha < x < \beta$.

Then $\chi'(x) > 0$ for all $\alpha < x < \beta$.

By the Mean Value Theorem:

There exists $y \in (\alpha, \beta)$ such that

$$\chi'(y) = \frac{\chi(\beta) - \chi(\alpha)}{\beta - \alpha} = 0.$$

$\alpha \geq 1$ is the unique positive real root of $\chi(z)$ □

The Reflected Characteristic Polynomial

Converting the Branching Vector

$$(1, 2, 2, 3, 6)$$

into the characteristic polynomial

$$z^6 - z^5 - z^4 - z^4 - z^3 - 1$$

is error-prone.
An alternative is the reflected characteristic polynomial

$$1 - z - z^2 - z^2 - z^3 - z^6.$$

The Reflected Characteristic Polynomial

Theorem

The size of a bounded search tree corresponding to a branching vector $(r_1, \ldots, r_m)$*, whose root is labeled by* n*, is bounded by*

$$poly(n)\alpha^{-n},$$

where α *is unique positive real root of the reflected characteristic polynomial*

$$\psi(z) = 1 - z^{r_1} - z^{r_2} - \cdots - z^{r_m}.$$

The Reflected Characteristic Polynomial

Theorem

α is a root of the characteristic polynomial if and only if $1/\alpha$ is a root of the reflected characteristic polynomial.

The Reflected Characteristic Polynomial

Theorem

α is a root of the characteristic polynomial if and only if $1/\alpha$ is a root of the reflected characteristic polynomial.

$$\begin{aligned}
\chi(z) &= z^t - z^{t-r_1} - z^{t-r_2} - \cdots - z^{t-r_m} \\
&= z^t \left(1 - z^{-r_1} - z^{-r_2} - \cdots - z^{-r_m}\right) \\
\psi(z) &= 1 - z^{r_1} - z^{r_2} - \cdots - z^{r_m}
\end{aligned}$$

The Reflected Characteristic Polynomial

Theorem

α is a root of the characteristic polynomial if and only if $1/\alpha$ is a root of the reflected characteristic polynomial.

$$\begin{aligned}
\chi(z) &= z^t - z^{t-r_1} - z^{t-r_2} - \cdots - z^{t-r_m} \\
&= z^t \left(1 - z^{-r_1} - z^{-r_2} - \cdots - z^{-r_m}\right) \\
\psi(z) &= 1 - z^{r_1} - z^{r_2} - \cdots - z^{r_m}
\end{aligned}$$

$$\chi(z) = z^t \psi(1/z)$$

The Reflected Characteristic Polynomial

Theorem

α is a root of the characteristic polynomial if and only if $1/\alpha$ is a root of the reflected characteristic polynomial.

$$\begin{aligned}
\chi(z) &= z^t - z^{t-r_1} - z^{t-r_2} - \ldots - z^{t-r_m} \\
&= z^t \left(1 - z^{-r_1} - z^{-r_2} - \ldots - z^{-r_m}\right) \\
\psi(z) &= 1 - z^{r_1} - z^{r_2} - \ldots - z^{r_m}
\end{aligned}$$

$$\chi(z) = z^t \psi(1/z)$$

Theorem

If the branching number α is a simple root of the characteristic polynomial, the size of the corresponding bounded search tree is bounded by $O(\alpha^n)$.

Branching Numbers — Example 1

- Consider the simple algorithm for SAT.
- The branching vector is $(1, 1)$.
- The characteristic polynomial is $z - 2$.
- The reflected characteristic polynomial is $1 - 2z$.
- The branching number is 2.
- The size of the search tree is bounded by $O(2^n)$.

Branching Vectors

If the two Branching Vectors $(1, 1)$ and $(2, 2, 3)$ occur in a search tree, we obtain the recurrence

$$B_n = \max\{2B_{n-1}, 2B_{n-2} + B_{n-3}\}.$$

We need methods to analyze Bounded Search Trees, where several different Branching Vectors occur. Hence, we need to solve such recurrences.

Several Branching Vectors

Theorem

Let M be a set of branching vectors. A bounded search tree whose recursive branches correspond to branching vectors in M and whose root is labeled by n is of size at most

$$n^{O(1)}\alpha^n,$$

where α denotes the largest branching number of all branching vectors in M.

Branching Numbers — Example 2

- Consider again the algorithm for SAT, but analyzed in the number of clauses m.
- SAT can be solved in polynomial time if each variable occurs at most twice.
- We can thus branch on a variable that occurs in at least three clauses.
- The branching vectors are (a, b) with $a, b \geq 1$, $a + b \geq 3$.
- The reflected characteristic polynomials are $1 - z^a - z^b$.

$$\text{For all } z \in [0, 1]: \qquad 1 - z^a - z^b \geq 1 - z^1 - z^2$$

Hence, the size of the bounded search tree is bounded by $O(1.618034^m)$, since 1.618034 is the branching number of $(1, 2)$.

Useful Properties

Lemma

Let $(r_1, \ldots, r_k)$ be a branching vector, and π a permutation. Then

$$\tau(r_1, \ldots, r_k) = \tau(r_{\pi(1)}, \ldots, r_{\pi(k)})$$

Proof: Both have the same characteristic polynomial. □

Useful Properties

Lemma

Let $(r_1, \ldots, r_k)$ be a branching vector, and π a permutation. Then

$$\tau(r_1, \ldots, r_k) = \tau(r_{\pi(1)}, \ldots, r_{\pi(k)})$$

Proof: Both have the same characteristic polynomial. □

Lemma

Let $(r_1, \ldots, r_k)$ be a branching vector. Then

$$\tau(r_1, \ldots, r_k) \geq 1$$

Proof: $1 - \sum_{i=1}^{k} 0^{r_i} = 1$ and $1 - \sum_{i=1}^{k} 1^{r_i} \leq 0$.
Hence, there is some $0 < \beta \leq 1$, such that $1 - \sum_{i=1}^{k} \beta^{r_i} = 0$ □

Useful Properties

Lemma

Let $(r_1, \ldots, r_k)$ and $(s_1, \ldots, s_k)$ be branching vectors, and $r_i \geq s_i$ for all $1 \leq i \leq k$. Then

$$\tau(r_1, \ldots, r_k) \leq \tau(s_1, \ldots, s_k)$$

$$1 - \sum_{i=1}^{k} z^{r_i} \geq 1 - \sum_{i=1}^{k} z^{s_i} \text{ for all } z \in [0, 1]$$

Let α be the unique positive real root of $1 - \sum_{i=1}^{k} z^{s_i}$. Then

$$1 - \sum_{i=1}^{k} z^{s_i} > 0 \text{ for all } z \in [0, \alpha)$$

and thus

$$1 - \sum_{i=1}^{k} z^{r_i} > 0 \text{ for all } z \in [0, \alpha)$$

Useful Properties

Lemma

Let $(r_1, \ldots, r_k)$ *and* $(s_1, \ldots, s_k)$ *be branching vectors, and* $r_i \geq s_i$ *for all* $1 \leq i \leq k$. *Then*

$$\tau(r_1, \ldots, r_k) \leq \tau(s_1, \ldots, s_k)$$

Lemma

Let $(r_1, \ldots, r_k)$ *and* $(r_1, \ldots, r_{k-1})$ *be branching vectors. Then*

$$\tau(r_1, \ldots, r_k) \geq \tau(r_1, \ldots, r_{k-1})$$

Proof: Analogously.

3-SAT

Pick a clause with variables x_1, x_2, x_3. Branch on all variables.
Either: x_1: $(1, 1)$ and then x_2: $(1, 1)$ and again x_3: $(1, 1)$
or x_1, x_2, x_3: $(3, 3, 3, 3, 3, 3, 3, 3)$

3-SAT

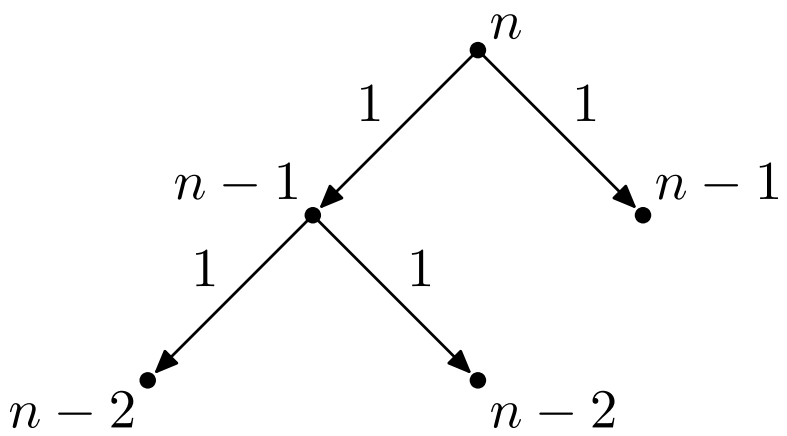
Pick a clause with variables x_1, x_2, x_3. Branch on all variables.
Either: x_1: $(1, 1)$ and then x_2: $(1, 1)$ and again x_3: $(1, 1)$
or x_1, x_2, x_3: $(3, 3, 3, 3, 3, 3, 3, 3)$

$$\tau(1, 1) = \tau(3, 3, 3, 3, 3, 3, 3, 3) = 2$$

3-SAT

Pick a clause with variables x_1, x_2, x_3. Branch on all variables.
Either: x_1: $(1, 1)$ and then x_2: $(1, 1)$ and again x_3: $(1, 1)$
or x_1, x_2, x_3: $(3, 3, 3, 3, 3, 3, 3, 3)$

$$\tau(1, 1) = \tau(3, 3, 3, 3, 3, 3, 3, 3) = 2$$



combining branching vectors

3-SAT

$$\{x_1, x_2, x_3\}$$

Only 7 out of 8 possibilities

$$\tau(3, 3, 3, 3, 3, 3, 3) \leq 1.92$$

3-SAT

$$\{x_1, x_2, x_3\}$$

Only 7 out of 8 possibilities

$$\tau(3, 3, 3, 3, 3, 3, 3) \leq 1.92$$

Better Idea

- $x_1 = 1$
- $x_1 = 0,\ x_2 = 1$
- $x_1 = 0,\ x_2 = 0,\ x_3 = 1$

$$\tau(1, 2, 3) \leq 1.84$$

$$\{x_1, x_2, x_3\}$$

Only 7 out of 8 possibilities

$$\tau(3,3,3,3,3,3,3) \leq 1.92$$

Better Idea

- $x_1 = 1$
- $x_1 = 0,\ x_2 = 1$
- $x_1 = 0,\ x_2 = 0,\ x_3 = 1$

$$\tau(1,2,3) \leq 1.84$$

x_1 also occurs negated $\{\overline{x_1}, y_1, y_2\}$

$x_1 = 1$ creates a clause of size two. $\Rightarrow$ Next branching $(1,2)$

$$\tau(2,3,2,3) \leq 1.77$$

Branching on a clause $\{x_1, x_2, \ldots, x_k\}$

- $x_1 = 1$
- $x_1 = 0,\ x_2 = 1$
- $\ldots$
- $x_1 = 0,\ x_2 = 0,\ \ldots,\ x_k = 1$

Branching vector: $(1, 2, \ldots, k)$

Find root of the characteristic polynomial:

$$\begin{aligned} x^k - x^{k-1} - \ldots - x - 1 &= 0 \\ x^{k+1} - x^k - \ldots - x^2 - x &= 0 \end{aligned}$$

$$x^{k+1} - 2x^k + 1 = 0$$

$k = 3$: 1.84, $k = 4$: 1.93, $\ldots$ $k = 10$: 1.9991

3-SAT

$(1, 2, 3)$ is followed by $(1, 2)$. And then?

3-SAT

$(1, 2, 3)$ is followed by $(1, 2)$. And then?

- As long as we have clause of size two, we obtain a better branching number
- How many good cases?

3-SAT

$(1, 2, 3)$ is followed by $(1, 2)$. And then?

- As long as we have clause of size two, we obtain a better branching number
- How many good cases?

Example

$$\varphi = \{\overline{x_1}, x_2, x_3\}, \{x_1, \overline{x_2}, x_3\}, \{x_1, \overline{x_2}, x_5\}, \{x_3, x_4, x_5\}, \ldots$$

3-SAT

$(1, 2, 3)$ is followed by $(1, 2)$. And then?

- As long as we have clause of size two, we obtain a better branching number
- How many good cases?

Example

$$\varphi = \{\overline{x_1}, x_2, x_3\}, \{x_1, \overline{x_2}, x_3\}, \{x_1, \overline{x_2}, x_5\}, \{x_3, x_4, x_5\}, \ldots$$
$$\varphi[x_1 = 0, x_2 = 0] = \{x_3, x_4, x_5\}, \ldots$$

3-SAT

$(1, 2, 3)$ is followed by $(1, 2)$. And then?

- As long as we have clause of size two, we obtain a better branching number
- How many good cases?

Example

$$\varphi = \{\overline{x_1}, x_2, x_3\}, \{x_1, \overline{x_2}, x_3\}, \{x_1, \overline{x_2}, x_5\}, \{x_3, x_4, x_5\}, \ldots$$
$$\varphi[x_1 = 0, x_2 = 0] = \{x_3, x_4, x_5\}, \ldots$$

No guarantee to have a good branch, but we gain something at the end.

3-SAT

$(1, 2, 3)$ is followed by $(1, 2)$. And then?

- As long as we have clause of size two, we obtain a better branching number
- How many good cases?

Example

$$\varphi = \{\overline{x_1}, x_2, x_3\}, \{x_1, \overline{x_2}, x_3\}, \{x_1, \overline{x_2}, x_5\}, \{x_3, x_4, x_5\}, \ldots$$
$$\varphi[x_1 = 0, x_2 = 0] = \{x_3, x_4, x_5\}, \ldots$$

No guarantee to have a good branch, but we gain something at the end.

But can we exploit this structure?

Partial Assignments

Theorem

Let φ be a formula in CNF over variables $x_1, \ldots, x_n$ and $\varphi = \psi \cup \varphi'$ such that φ' contains no variables in $x_1, \ldots, x_i$.
Moreover, let there be an assignment A to the variables $x_1, \ldots, x_i$ that satisfies all clauses in ψ.

Then, φ is satisfiable if and only if φ' is satisfiable.

Partial Assignments

Theorem

Let φ be a formula in CNF over variables $x_1, \ldots, x_n$ and $\varphi = \psi \cup \varphi'$ such that φ' contains no variables in $x_1, \ldots, x_i$.
Moreover, let there be an assignment A to the variables $x_1, \ldots, x_i$ that satisfies all clauses in ψ.

Then, φ is satisfiable if and only if φ' is satisfiable.

Clearly, if φ is satisfiable, then φ' is also satisfiable.

Partial Assignments

Theorem

Let φ be a formula in CNF over variables $x_1, \ldots, x_n$ and $\varphi = \psi \cup \varphi'$ such that φ' contains no variables in $x_1, \ldots, x_i$.
Moreover, let there be an assignment A to the variables $x_1, \ldots, x_i$ that satisfies all clauses in ψ.

Then, φ is satisfiable if and only if φ' is satisfiable.

Clearly, if φ is satisfiable, then φ' is also satisfiable.

- Let I' satisfy φ'.
- φ' does not contain $x_1, \ldots, x_i$
- Together I and I' define assignment for $x_1, \ldots, x_n$

Partial Assignments

Theorem

Let φ be a formula in CNF over variables $x_1, \ldots, x_n$ and $\varphi = \psi \cup \varphi'$ such that φ' contains no variables in $x_1, \ldots, x_i$.
Moreover, let there be an assignment A to the variables $x_1, \ldots, x_i$ that satisfies all clauses in ψ.

Then, φ is satisfiable if and only if φ' is satisfiable.

Clearly, if φ is satisfiable, then φ' is also satisfiable.

- Let I' satisfy φ'.
- φ' does not contain $x_1, \ldots, x_i$
- Together I and I' define assignment for $x_1, \ldots, x_n$

Now, all clauses in φ' are satisfied and all clauses in ψ are satisfied by I.

Definition

φ' is called autark (for I).

Idea:

- Either we branch according to $(1, 2, \ldots, k-1)$ or
- We do not have to branch at all

First branch is always $(1, 2, \ldots, k)$

We need to ensure that a clause of size at most $k-1$ always exists

Idea:

- Either we branch according to $(1, 2, \ldots, k-1)$ or
- We do not have to branch at all

First branch is always $(1, 2, \ldots, k)$

But only polynomial overhead, occurs at most once.

We need to ensure that a clause of size at most $k-1$ always exists

Idea:

- Either we branch according to $(1, 2, \ldots, k-1)$ or
- We do not have to branch at all

First branch is always $(1, 2, \ldots, k)$

But only polynomial overhead, occurs at most once.

We need to ensure that a clause of size at most $k-1$ always exists

Before we do any branch, we need to check whether one branch corresponds to an autark assignment

Algorithm $\mathcal{SAT}$

If (F contains $\{\}$) **return false**;
If ($F = \emptyset$) **return true**;
choose a shortest clause $\{l_1, \ldots, l_s\}$;
Compute $F_i := F[l_1 = 0, \ldots, l_{i-1} = 0, l_i = 1]$;
If (F_i is autark **for** $l_1 = 0, \ldots, l_{i-1} = 0, l_i = 1$)
  return $\mathcal{SAT}(F_i)$;
for $i = 1, \ldots, s$
  if ($\mathcal{SAT}(F_i)$) **return true**;
return false;

Algorithm $\mathcal{SAT}$

If (F contains $\{\}$) **return false**;
If ($F = \emptyset$) **return true**;
choose a shortest clause $\{l_1, \ldots, l_s\}$;
Compute $F_i := F[l_1 = 0, \ldots, l_{i-1} = 0, l_i = 1]$;
If (F_i is autark **for** $l_1 = 0, \ldots, l_{i-1} = 0, l_i = 1$)
  return $\mathcal{SAT}(F_i)$;
for $i = 1, \ldots, s$
  if ($\mathcal{SAT}(F_i)$) **return true**;
return false;

Theorem

Algorithm $\mathcal{SAT}$ is correct.

Running Time of Algorithm $\mathcal{SAT}$

Theorem

The running time of Algorithm $\mathcal{SAT}$ is bounded by $O^*(\tau(1, 2 \ldots, k-1))$.

Claim

Whenever Algorithm $\mathcal{SAT}$ reaches the recursive calls, we have

- $s < k$, or
- it is the very first call of Algorithm $\mathcal{SAT}$, or
- the call before found an autark formula

Running Time of Algorithm $\mathcal{SAT}$

Theorem

The running time of Algorithm $\mathcal{SAT}$ is bounded by $O^*(\tau(1, 2 \ldots, k-1))$.

Claim

Whenever Algorithm $\mathcal{SAT}$ reaches the recursive calls, we have

- $s < k$, or
- it is the very first call of Algorithm $\mathcal{SAT}$, or
- the call before found an autark formula

Runtime bound

- $s < k$ yields at most $(1, 2, \ldots, k-1)$
- If an autark formula is found, the number of variables decreases by at least one.
- Combined with $(1, \ldots, k)$, this yields $(2, 3, \ldots, k+1)$.

Running Time of Algorithm $\mathcal{SAT}$

Runtime bound $\tau(1, 2, \ldots, k-1)$

- $s < k$ yields at most $(1, 2, \ldots, k-1)$
- If an autark formula is found, the number of variables decreases by at least one.
- Combined with $(1, \ldots, k)$, this yields $(2, 3, \ldots, k+1)$.

$$\chi(z) = z^{k+1} - z^{k-1} - \ldots - 1 \quad = \quad z^2(z^{k-1} - z^{k-2} - \ldots - 1) + z^k - z - 1$$

Running Time of Algorithm $\mathcal{SAT}$

Runtime bound $\tau(1, 2, \ldots, k-1)$

- $s < k$ yields at most $(1, 2, \ldots, k-1)$
- If an autark formula is found, the number of variables decreases by at least one.
- Combined with $(1, \ldots, k)$, this yields $(2, 3, \ldots, k+1)$.

$$\chi(z) = z^{k+1} - z^{k-1} - \ldots - 1 \quad = \quad z^2(z^{k-1} - z^{k-2} - \ldots - 1) + z^k - z - 1$$

Let α be the root of $z^{k-1} - z^{k-2} - \ldots - 1$. Then

$$\chi(\alpha) = \alpha^k - \alpha - 1$$

Running Time of Algorithm $\mathcal{SAT}$

Runtime bound $\tau(1, 2, \ldots, k-1)$

- $s < k$ yields at most $(1, 2, \ldots, k-1)$
- If an autark formula is found, the number of variables decreases by at least one.
- Combined with $(1, \ldots, k)$, this yields $(2, 3, \ldots, k+1)$.

$$\chi(z) = z^{k+1} - z^{k-1} - \ldots - 1 \quad = \quad z^2(z^{k-1} - z^{k-2} - \ldots - 1) + z^k - z - 1$$

Let α be the root of $z^{k-1} - z^{k-2} - \ldots - 1$. Then

$$\chi(\alpha) = \alpha^k - \alpha - 1$$

Since $\alpha = \tau(1, 2, \ldots, k-1) \geq \tau(1, 2) \geq 1.6$, $\chi(\alpha) \geq 0$ for $k \geq 3$.

Designing a fast exact algorithm for Independent Set

Given a graph $G = (V, E)$, $V' \subseteq V$ is called independent, if $G[V']$ has no edges.

Definition (Independent Set)

Input: A graph $G = (V, E)$, a number $k \in \mathbb{N}$
Question: Is there an independent set in G of size k?

Definition

The size of a maximum independent set $I \subseteq V$ in $G = (V, E)$ is denoted by $\alpha(G) = |I|$.

Designing a fast exact algorithm for Independent Set

Given a graph $G = (V, E)$, $V' \subseteq V$ is called independent, if $G[V']$ has no edges.

Definition (Alternative)

Input: A graph $G = (V, E)$
Output: The size of a maximum independent set.

Definition

The size of a maximum independent set $I \subseteq V$ in $G = (V, E)$ is denoted by $\alpha(G) = |I|$.

Designing a fast exact algorithm for Independent Set

Given a graph $G = (V, E)$, $V' \subseteq V$ is called independent, if $G[V']$ has no edges.

Definition (Independent Set)

Input: A graph $G = (V, E)$, a number $k \in \mathbb{N}$
Question: Is there an independent set in G of size k?

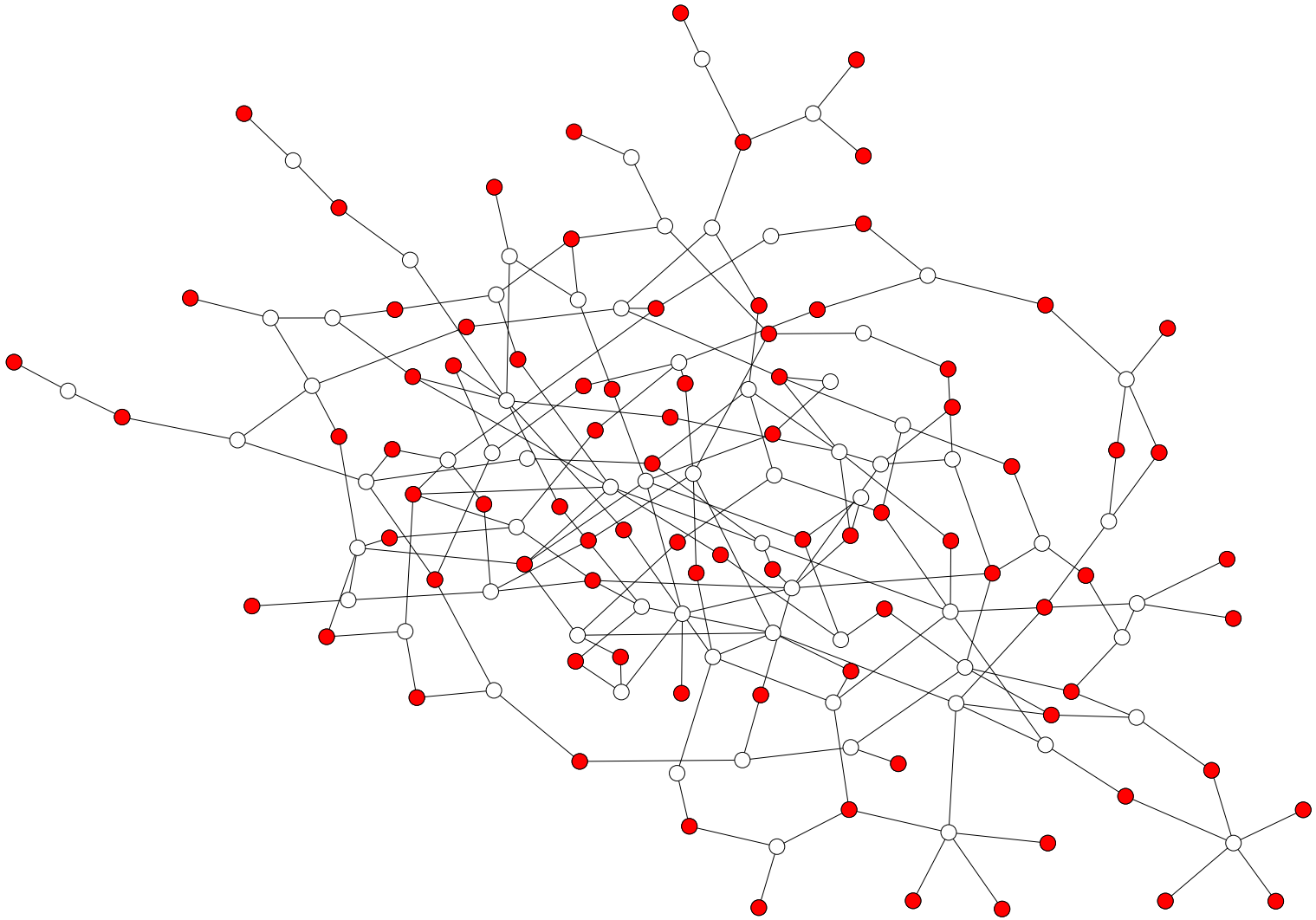
Definition

The size of a maximum independent set $I \subseteq V$ in $G = (V, E)$ is denoted by $\alpha(G) = |I|$.

Starting with a simple idea:

- Pick a node $v \in V$.
- Branch whether v belongs to an independent set or not.
- Clearly branching vector $(1, 1)$, hence $O^*(2^n)$

Independent Set



Algorithm $\mathcal{IS}$

If $(V = \emptyset)$ **return** 0;
Choose $u \in V$;
return $\max(\mathcal{IS}(G \setminus \{u\}), \mathcal{IS}(G \setminus N[u]) + 1)$;

Correctness: Obvious, since either $u \in I$ or $u \notin I$ for an optimal independent set I

Runtime: Clearly $O^*(2^n)$

Algorithm $\mathcal{IS}$

If $(V = \emptyset)$ **return** 0;
Choose $u \in V$;
return $\max(\mathcal{IS}(G \setminus \{u\}), \mathcal{IS}(G \setminus N[u]) + 1)$;

Correctness: Obvious, since either $u \in I$ or $u \notin I$ for an optimal independent set I

Runtime: Clearly $O^*(2^n)$

Lemma

Let $G = (V, E)$ be a graph, $u \in V$ and $\deg(u) = 0$.
Then $\alpha(G) = \alpha(G \setminus \{u\}) + 1$.

Algorithm $\mathcal{IS}$

```
if (V = ∅) return 0;
if (∃u ∈ V with deg(u) = 0)
   return IS(G \ {u}) + 1;
Choose u ∈ V;
return max(IS(G \ {u}), IS(G \ N[u]) + 1);
```

Algorithm $\mathcal{IS}$

if $(V = \emptyset)$ **return** 0;
if $(\exists u \in V$ with $\deg(u) = 0)$
  return $\mathcal{IS}(G \setminus \{u\}) + 1$;
Choose $u \in V$;
return $\max(\mathcal{IS}(G \setminus \{u\}), \mathcal{IS}(G \setminus N[u]) + 1)$;

Correctness: By previous lemma and exhaustive search

Runtime: $\tau(1, 2) \Rightarrow O^*(1.62^n)$

A Different Approach

Lemma

Let I be an independent set. If $v \in I$, then $N(v) \cap I = \emptyset$.

Lemma

Let I be a maximal independent set and $v \in V$. Then $N[v] \cap I \neq \emptyset$.

A Different Approach

Lemma

Let I be an independent set. If $v \in I$, then $N(v) \cap I = \emptyset$.

Lemma

Let I be a maximal independent set and $v \in V$. Then $N[v] \cap I \neq \emptyset$.

Algorithm Simple-IS

```
If (V = ∅) return 0;
Choose u ∈ V;
max =0;
for each v ∈ N[u]
   if (Simple−IS(G \ N[v])+1>max)
      max = Simple−IS(G \ N[v])+1;
return max;
```

A Different Approach

Let $N(u) = \{u_1, \ldots, u_l)$. Then the branching vector is

$$(\deg(u) + 1, \deg(u_1) + 1, \ldots, \deg(u_l) + 1).$$

Example ($\deg(u) = 4$, all neighbors of degree two)

$$\tau(5, 3, 3, 3, 3) > 1.63$$

A Different Approach

Let $N(u) = \{u_1, \ldots, u_l)$. Then the branching vector is

$$(\deg(u) + 1, \deg(u_1) + 1, \ldots, \deg(u_l) + 1).$$

Example ($\deg(u) = 4$, all neighbors of degree two)

$$\tau(5, 3, 3, 3, 3) > 1.63$$

Algorithm Simple-IS

```
If (V = ∅) return 0;
Choose u ∈ V of minimum degree;
max =0;
for each v ∈ N[u]
   if (Simple−IS(G \ N[v])+1>max)
      max = Simple−IS(G \ N[v])+1;
return max;
```

Different Branching

Let $N(u) = \{u_1, \ldots, u_l)$. Then the branching vector is

$$(\deg(u) + 1, \deg(u_1) + 1, \ldots, \deg(u_l) + 1),$$

where $\deg(u) \leq \deg(u_i)$ and $\deg(u) = l$.

Different Branching

Let $N(u) = \{u_1, \ldots, u_l)$. Then the branching vector is

$$(\deg(u) + 1, \deg(u_1) + 1, \ldots, \deg(u_l) + 1),$$

where $\deg(u) \leq \deg(u_i)$ and $\deg(u) = l$.

Worst Case:

$$\underbrace{(\deg(u) + 1, \ldots, \deg(u) + 1)}_{(\deg(u)+1)\mathit{times}}$$

Different Branching

Let $N(u) = \{u_1, \ldots, u_l)$. Then the branching vector is

$$(\deg(u) + 1, \deg(u_1) + 1, \ldots, \deg(u_l) + 1),$$

where $\deg(u) \leq \deg(u_i)$ and $\deg(u) = l$.

Worst Case:

$$\underbrace{(\deg(u) + 1, \ldots, \deg(u) + 1)}_{(\deg(u)+1)\,times}$$

Characteristic polynomial:

$$z^{\deg(u)+1} - (\deg(u) + 1)$$

Root: $\quad (\deg(u) + 1)^{\frac{1}{\deg(u)+1}}$

Different Branching

Let $N(u) = \{u_1, \ldots, u_l)$. Then the branching vector is

$$(\deg(u) + 1, \deg(u_1) + 1, \ldots, \deg(u_l) + 1),$$

where $\deg(u) \leq \deg(u_i)$ and $\deg(u) = l$.

Worst Case:

$$\underbrace{(\deg(u) + 1, \ldots, \deg(u) + 1)}_{(\deg(u)+1)\,times}$$

Characteristic polynomial:

$$z^{\deg(u)+1} - (\deg(u) + 1)$$

Root: $\quad (\deg(u) + 1)^{\frac{1}{\deg(u)+1}}$

Worst Case:

$$\deg(u) = 2,\ \tau(3, 3, 3) \approx 1.45$$

Two Simple Algorithms

- Runtime of Algorithm $\mathcal{IS}$ is bad, if maximum degree is small
- Runtime of Algorithm $Simple - \mathcal{IS}$ is bad, if minimum degree is small

Reductions

- Solve problem in $poly(n)$ if some condition is satisfied
  Example: If $(V = \emptyset)$ return 0;
- Something holds in any successful branch
  Example: if $(\exists u \in V$ with $\deg(u) = 0)$ return $\mathcal{IS}(G \setminus \{u\}) + 1$;

Two Simple Algorithms

- Runtime of Algorithm $\mathcal{IS}$ is bad, if maximum degree is small
- Runtime of Algorithm $Simple - \mathcal{IS}$ is bad, if minimum degree is small

Reductions

- Solve problem in $poly(n)$ if some condition is satisfied
  Example: If $(V = \emptyset)$ return 0;
- Something holds in any successful branch
  Example: if $(\exists u \in V$ with $\deg(u) = 0)$ return $\mathcal{IS}(G \setminus \{u\}) + 1$;
- Something holds in at least one successful branch

Two Simple Algorithms

- Runtime of Algorithm $\mathcal{IS}$ is bad, if maximum degree is small
- Runtime of Algorithm $Simple - \mathcal{IS}$ is bad, if minimum degree is small

Reductions

- Solve problem in $poly(n)$ if some condition is satisfied
  Example: If $(V = \emptyset)$ return 0;
- Something holds in any successful branch
  Example: if $(\exists u \in V$ with $\deg(u) = 0)$ return $\mathcal{IS}(G \setminus \{u\}) + 1$;
- Something holds in at least one successful branch

Lemma

Let $G = (V, E)$ *be a graph,* $u \in V$ *and* $\deg(u) = 1$.
Then $\alpha(G) = \alpha(G \setminus \{u\}) + 1$.

The Power of Reduction Rules

Reduction Rules

- Used to get rid of bad cases without branching
- Positive side effect: Analysis can become easier

Lemma

Let $G = (V, E)$ and $\Delta(G) \leq 2$. Then $\alpha(G)$ can be computed in polynomial time.

The Power of Reduction Rules

Reduction Rules

- Used to get rid of bad cases without branching
- Positive side effect: Analysis can become easier

Lemma

Let $G = (V, E)$ and $\Delta(G) \leq 2$. Then $\alpha(G)$ can be computed in polynomial time.

G consists of cycles and paths.

The Power of Reduction Rules

Reduction Rules

- Used to get rid of bad cases without branching
- Positive side effect: Analysis can become easier

Lemma

Let $G = (V, E)$ and $\Delta(G) \leq 2$. Then $\alpha(G)$ can be computed in polynomial time.

G consists of cycles and paths.

Lemma

Let $G = (V, E)$ and let G consist of connected components $G_1, \ldots, G_l$. Then

$$\alpha(G) = \sum_{i=1}^{l} \alpha(G_i).$$

The Power of Reduction Rules

Reduction Rules

- Used to get rid of bad cases without branching
- Positive side effect: Analysis can become easier

Lemma

Let $G = (V, E)$ and $\Delta(G) \leq 2$. Then $\alpha(G)$ can be computed in polynomial time.

G consists of cycles and paths.

Lemma

Let $G = (V, E)$ and let G consist of connected components $G_1, \ldots, G_l$. Then

$$\alpha(G) = \sum_{i=1}^{l} \alpha(G_i).$$

An optimal solution for G_i does not affect G_j.

The Power of Reduction Rules

Lemma

Let $G = (V, E)$ and $\Delta(G) \leq 2$. Then $\alpha(G)$ can be computed in polynomial time.

implies

Lemma

The running time of Algorithm $\mathcal{IS}$ is bounded by $O^*(\tau(1,4)) = O^*(1.39^n)$

Algorithm $\mathcal{IS}$

if $(\Delta(G) \leq 2)$ solve problem in $poly(n)$
Choose $u \in V$ of maximum degree;
return $\max(\mathcal{IS}(G \setminus \{u\}), \mathcal{IS}(G \setminus N[u]) + 1)$;

The Power of Reduction Rules

Lemma

The running time of Algorithm $\mathcal{IS}$ is bounded by $O^*(\tau(1,4)) = O^*(1.39^n)$

Algorithm $\mathcal{IS}$

if $(\Delta(G) \leq 2)$ solve problem in $poly(n)$
Choose $u \in V$ of maximum degree;
return $\max(\mathcal{IS}(G \setminus \{u\}), \mathcal{IS}(G \setminus N[u]) + 1)$;

Not needed anymore

if $(V = \emptyset)$ **return** 0;
if $(\exists u \in V$ with $\deg(u) = 0)$
  return $\mathcal{IS}(G \setminus \{u\}) + 1$;

And Simple-$\mathcal{IS}$?

Lemma

Let $G = (V, E)$ *be a graph,* $u \in V$ *and* $\deg(u) \leq 1$.
Then $\alpha(G) = \alpha(G \setminus N[u]) + 1$.

Does not improve runtime bound, since worst case is $\deg(u) = 2$.

And Simple-$\mathcal{IS}$?

Lemma

Let $G = (V, E)$ be a graph, $u \in V$ and $\deg(u) \leq 1$.
Then $\alpha(G) = \alpha(G \setminus N[u]) + 1$.

Does not improve runtime bound, since worst case is $\deg(u) = 2$.

Lemma

Let $G = (V, E)$ and $\Delta(G) \leq 2$. Then $\alpha(G)$ can be computed in polynomial time.

Does not improve runtime bound,
since we branch on a node of minimum degree.

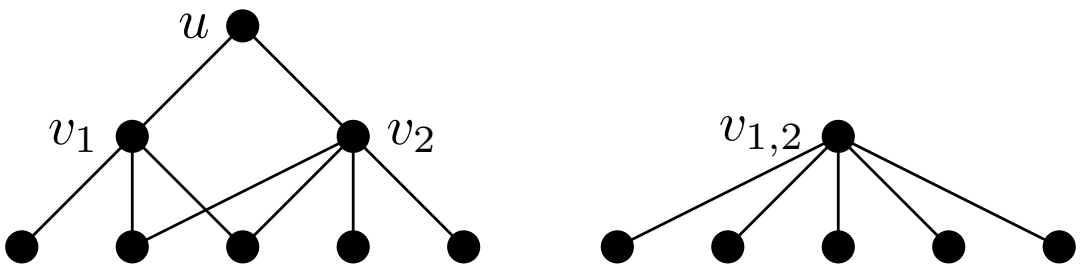
Nodes of degree two

Definition

Let $G = (V, E)$ and $u \in V$ with $N(u) = \{v_1, v_2\}$. Then $fold_u(G)$ is defined by

- Removing u
- Add $v_{1,2}$ to V and connect each node in $N(v_1, v_2) \setminus \{u\}$ to $v_{1,2}$
- Remove $\{v_1, v_2\}$

This operation is called *folding*.



Folding

Lemma

Let $G = (V, E)$ *and* $u \in V$ *with* $N(u) = \{v_1, v_2\}$ *such that* $\{v_1, v_2\} \notin E$. *Then*

$$\alpha(G) = \alpha(fold_u(G)) + 1$$

Let I be an independent set in $fold_u(G)$.

- If $v_{1,2} \in I$, then $I \cup \{v_1, v_2\} \setminus \{v_{1,2}\}$ is an independent set in G of size $|I| + 1$.
- If $v_{1,2} \notin I$, then $I \cup \{u\}$ is an independent set in G of size $|I| + 1$.

Folding

Lemma

Let $G = (V, E)$ and $u \in V$ with $N(u) = \{v_1, v_2\}$ such that $\{v_1, v_2\} \notin E$. Then

$$\alpha(G) = \alpha(fold_u(G)) + 1$$

Let I be an independent set in $fold_u(G)$.

- If $v_{1,2} \in I$, then $I \cup \{v_1, v_2\} \setminus \{v_{1,2}\}$ is an independent set in G of size $|I| + 1$.
- If $v_{1,2} \notin I$, then $I \cup \{u\}$ is an independent set in G of size $|I| + 1$.

Let I be an independent set in G.

- If $\{v_1, v_2\} \subseteq I$, then $I \cup \{v_{1,2}\} \setminus \{v_1, v_2\}$ is an independent set in $fold_u(G)$ of size $|I| - 1$.
- If $\{v_1, v_2\} \not\subseteq I$, there exists an independent set I' with $u \in I'$ and $|I| = |I'|$.

Folding

Lemma

Let $G = (V, E)$ and $u \in V$ with $N(u) = \{v_1, v_2\}$ such that $\{v_1, v_2\} \notin E$. Then

$$\alpha(G) = \alpha(fold_u(G)) + 1$$

Let I be an independent set in $fold_u(G)$.

- If $v_{1,2} \in I$, then $I \cup \{v_1, v_2\} \setminus \{v_{1,2}\}$ is an independent set in G of size $|I| + 1$.
- If $v_{1,2} \notin I$, then $I \cup \{u\}$ is an independent set in G of size $|I| + 1$.

Let I be an independent set in G.

- If $\{v_1, v_2\} \subseteq I$, then $I \cup \{v_{1,2}\} \setminus \{v_1, v_2\}$ is an independent set in $fold_u(G)$ of size $|I| - 1$.
- If $\{v_1, v_2\} \not\subseteq I$, there exists an independent set I' with $u \in I'$ and $|I| = |I'|$.
  $I' \setminus \{u\}$ is an independent set in $fold_u(G)$ of size $|I| - 1$.

Domination

What if $\{v_1, v_2\} \in E$?

Domination

What if $\{v_1, v_2\} \in E$?

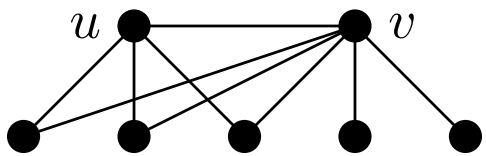
u, v_1, v_2 is a triangle, and $\deg(u) = 2$. Does this help?

Lemma

Let $G = (V, E)$ be a graph and let $u, v \in V$ such that $N[u] \subseteq N[v]$. Then

$$\alpha(G) = \alpha(G) \setminus \{v\}.$$

We say v dominates u.



Domination

What if $\{v_1, v_2\} \in E$?

u, v_1, v_2 is a triangle, and $\deg(u) = 2$. Does this help?

Lemma

Let $G = (V, E)$ be a graph and let $u, v \in V$ such that $N[u] \subseteq N[v]$. Then

$$\alpha(G) = \alpha(G) \setminus \{v\}.$$

We say v dominates u.

Let I be an independent set in G with $v \in I$.

- $u \notin I$, since $N[u] \subseteq N[v]$. This implies $|I \cup \{u\} \setminus \{v\}| = |I|$.
- $N(v) \cap I = \emptyset$. Hence, $N(u) \cap I = \emptyset$. Thus $I \cup \{u\} \setminus \{v\}$ is an independent set.

Runtime of Algorithm Simple-$\mathcal{IS}$

Algorithm Simple-IS

Apply reduction rules; If $(V = \emptyset)$ **return** 0;
Choose $u \in V$ of minimum degree;
max =0;
for each $v \in N[u]$
  if $(S - \mathcal{IS}(G \setminus N[v]) + 1 > max)$ $max = S - \mathcal{IS}(G \setminus N[v]) + 1$;
return max;

$\delta(G) \geq 3$

Branching on neighbors of u yields

$$\underbrace{\tau(a+1, \ldots, a+1)}_{(a+1)times}$$

with $a \geq 3$. Worst case is $\tau(4,4,4,4) = 1.42$.

Overview

Runtime Bounds

- Algorithm $\mathcal{IS}$: $\tau(1,4) = 1.39$
- Algorithm Simple-$\mathcal{IS}$: $\tau(4,4,4,4) = 1.42$

Overview

Runtime Bounds

- Algorithm $\mathcal{IS}$: $\tau(1,4) = 1.39$
- Algorithm Simple-$\mathcal{IS}$: $\tau(4,4,4,4) = 1.42$

- Analyzing only one branching step
- In k-SAT: Multiple steps together to obtain better bounds

Overview

Runtime Bounds

- Algorithm $\mathcal{IS}$: $\tau(1,4) = 1.39$
- Algorithm Simple-$\mathcal{IS}$: $\tau(4,4,4,4) = 1.42$

- Analyzing only one branching step
- In k-SAT: Multiple steps together to obtain better bounds

Worst-Case: 3-regular graphs in both algorithms

Analyzing more than one step

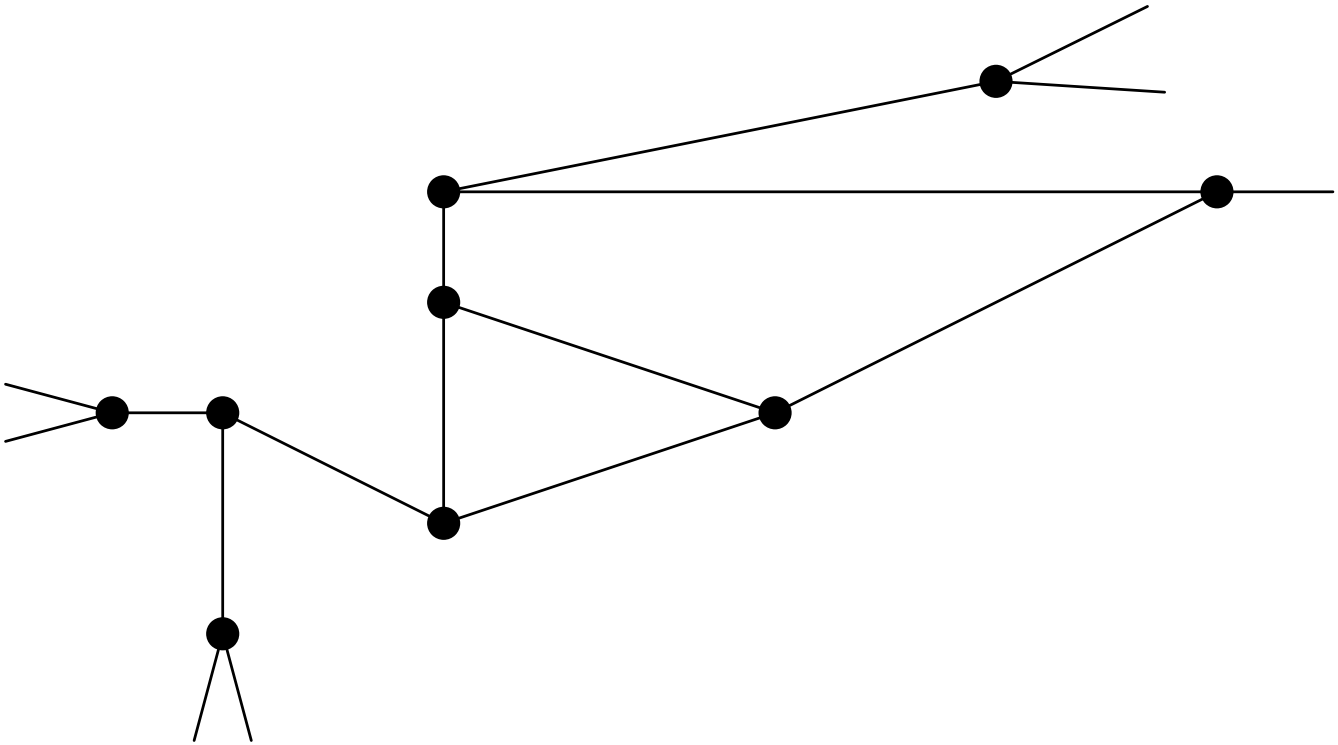
Algorithm Simple-IS

```
...
for each v ∈ N[u]
    if (S − IS(G \ N[v]) + 1 > max) max = S − IS(G \ N[v]) + 1;
return max;
```

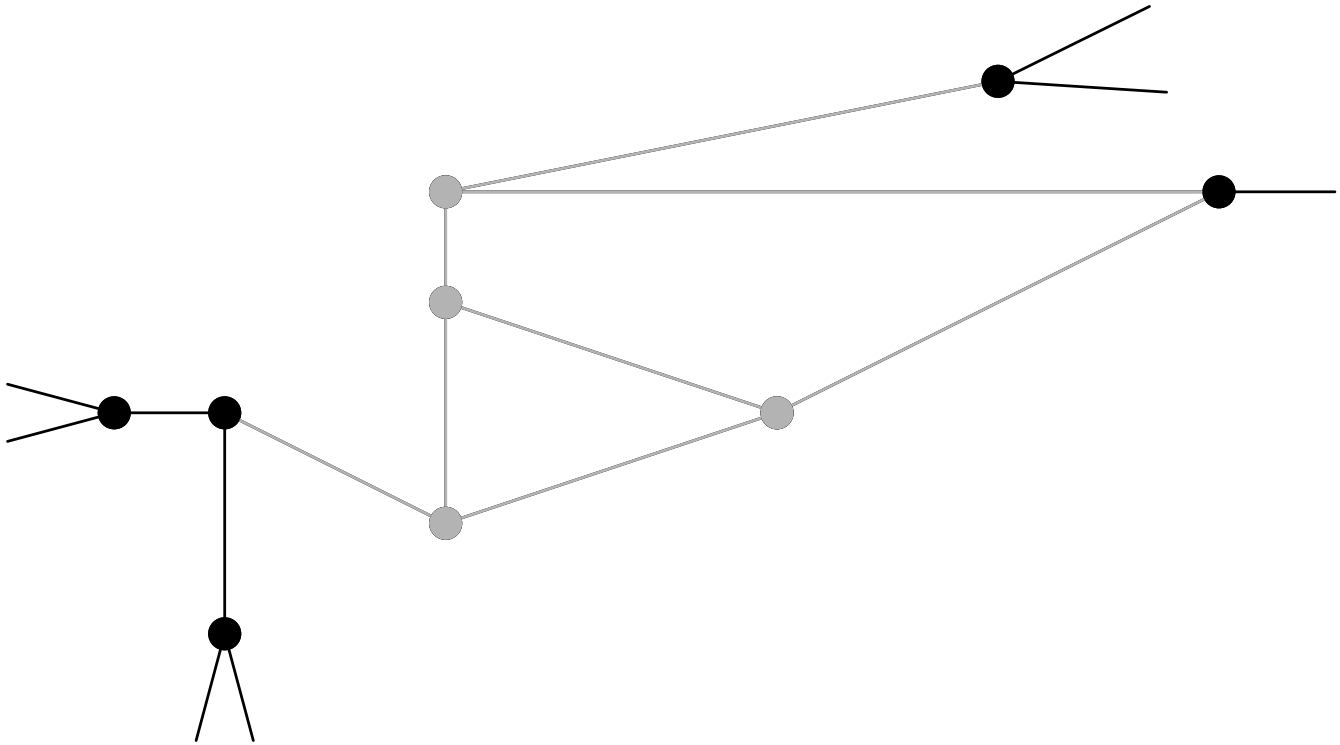
Assume G is 3-regular.

- We remove $N[v]$ for some $v \in V$ in each step
- What happens in the next step?

Analyzing more than one step



Analyzing more than one step



3-regular graphs

- Branching in 3-regular graphs decreases the degree of other nodes below 3
- These nodes are affected by some reduction rules

3-regular graphs

- Branching in 3-regular graphs decreases the degree of other nodes below 3
- These nodes are affected by some reduction rules

But how many? And by which reduction rule?

3-regular graphs

- Branching in 3-regular graphs decreases the degree of other nodes below 3
- These nodes are affected by some reduction rules

But how many? And by which reduction rule?

- Isolated Nodes are removed (affects one node)
- Nodes of degree one are added to the independent set (affects two nodes)
- Folding removes three nodes and adds a new one

3-regular graphs

- Branching in 3-regular graphs decreases the degree of other nodes below 3
- These nodes are affected by some reduction rules

But how many? And by which reduction rule?

- Isolated Nodes are removed (affects one node)
- Nodes of degree one are added to the independent set (affects two nodes)
- Folding removes three nodes and adds a new one

After a reduction rule is applied, other nodes might now be reducible!

3-regular graphs

Lemma

Let $G = (V, E)$ be a graph, such that no node is dominated. Then

$$N(v) \setminus N[u] \neq \emptyset$$

for all $v \in N(u)$.

3-regular graphs

Lemma

Let $G = (V, E)$ be a graph, such that no node is dominated. Then

$$N(v) \setminus N[u] \neq \emptyset$$

for all $v \in N(u)$.

- If $N(v) \setminus N[u] = \emptyset$, we have $N(v) \subseteq N[u]$
- $v \in N(u)$ implies $N[v] \subseteq N[u]$
- v is dominated by u

3-regular graphs

Lemma

Let $G = (V, E)$ be a graph, such that no node is dominated. Then

$$N(v) \setminus N[u] \neq \emptyset$$

for all $v \in N(u)$.

- If $N(v) \setminus N[u] = \emptyset$, we have $N(v) \subseteq N[u]$
- $v \in N(u)$ implies $N[v] \subseteq N[u]$
- v is dominated by u

- There are at least 3 edges to the remaining graph if $N[u]$ is removed

3-regular graphs

Lemma

Let $G = (V, E)$ be a connected, 3-regular graph with $|V| \geq 6$. Moreover, assume that no node is dominated. Then $|N^2(v)| \geq 2$ for $v \in V$.

3-regular graphs

Lemma

Let $G = (V, E)$ be a connected, 3-regular graph with $|V| \geq 6$. Moreover, assume that no node is dominated. Then $|N^2(v)| \geq 2$ for $v \in V$.

- Each node in $N(v)$ has at least one edge to $N^2(v)$
- If $|N^2(v)| = 1$, the single node $u \in N^2(v)$ has three neighbors in $N(v)$
- u has no other neighbors
- $N^2[v]$ is a connected component with 5 nodes

3-regular graphs

Lemma

Let $G = (V, E)$ be a connected, 3-regular graph with $|V| \geq 6$ and $v \in V$. Moreover, assume that no node is dominated. Let $G' = G \setminus N[v]$ and $G'' = (V'', E'')$ be the graph obtained from G' by applying folding and domination rules.

Then

$$|V''| \leq |V| - 6$$

3-regular graphs

Lemma

Let $G = (V, E)$ be a connected, 3-regular graph with $|V| \geq 6$ and $v \in V$. Moreover, assume that no node is dominated. Let $G' = G \setminus N[v]$ and $G'' = (V'', E'')$ be the graph obtained from G' by applying folding and domination rules.
Then

$$|V''| \leq |V| - 6$$

Each reduction rules decreases the number of nodes.

- $\deg(u) \leq 2$ for all $u \in N^2(v)$ in $G \setminus N[v]$.
- If $\deg(u) = 2$ for some u, two nodes are removed by folding
- If $\deg(u) = 1$ for some u, two nodes are removed by domination
- Otherwise all nodes are isolated, but there are at least two nodes

3-regular graphs

Algorithm Simple-IS

...
for each $v \in N[u]$
  if $(S - \mathcal{IS}(G \setminus N[v]) + 1 > max)$ $max = S - \mathcal{IS}(G \setminus N[v]) + 1$;
return max;

Worst-Case: $\tau(4, 4, 4, 4)$

3-regular graphs

Algorithm Simple-IS

...
for each $v \in N[u]$
  if $(S - \mathcal{IS}(G \setminus N[v]) + 1 > max)$ $max = S - \mathcal{IS}(G \setminus N[v]) + 1$;
return max;

Worst-Case: $\tau(4,4,4,4)$

Runtime on 3-regular graph

- Number of nodes decreases by 6 due to reduction rules
- $\tau(6,6,6,6) = 1.26$

3-regular graphs

Algorithm Simple-IS

...

for each $v \in N[u]$

  if $(S - \mathcal{IS}(G \setminus N[v]) + 1 > max)$ $max = S - \mathcal{IS}(G \setminus N[v]) + 1$;

return max;

Worst-Case: $\tau(4, 4, 4, 4)$

Runtime on 3-regular graph

- Number of nodes decreases by 6 due to reduction rules
- $\tau(6, 6, 6, 6) = 1.26$

Is this the new runtime bound?

Graphs with $\Delta(G) > 3$, $\delta(G) = 3$

Graph not regular

- Removing edges to nodes of higher degree
- Reduction rules cannot be applied
- We do not gain additional nodes

Graphs with $\Delta(G) > 3$, $\delta(G) = 3$

Graph not regular

- Removing edges to nodes of higher degree
- Reduction rules cannot be applied
- We do not gain additional nodes

But the degree of some nodes is higher

If $\delta(G) = 3$ and $\Delta(G) > 3$, there is node with a high-degree neighbor.

$$\tau(4, 4, 4, 5) = 1.39$$

Probably other reduction rules possible or better branching vector

But hard to prove!

Back to Algorithm $\mathcal{IS}$

Worst case: $\Delta(G) = 3$, $\tau(1, 4)$

$\delta(G) = \Delta(G) = 3$

By lemma: $\tau(1, 6) = 1.29$.

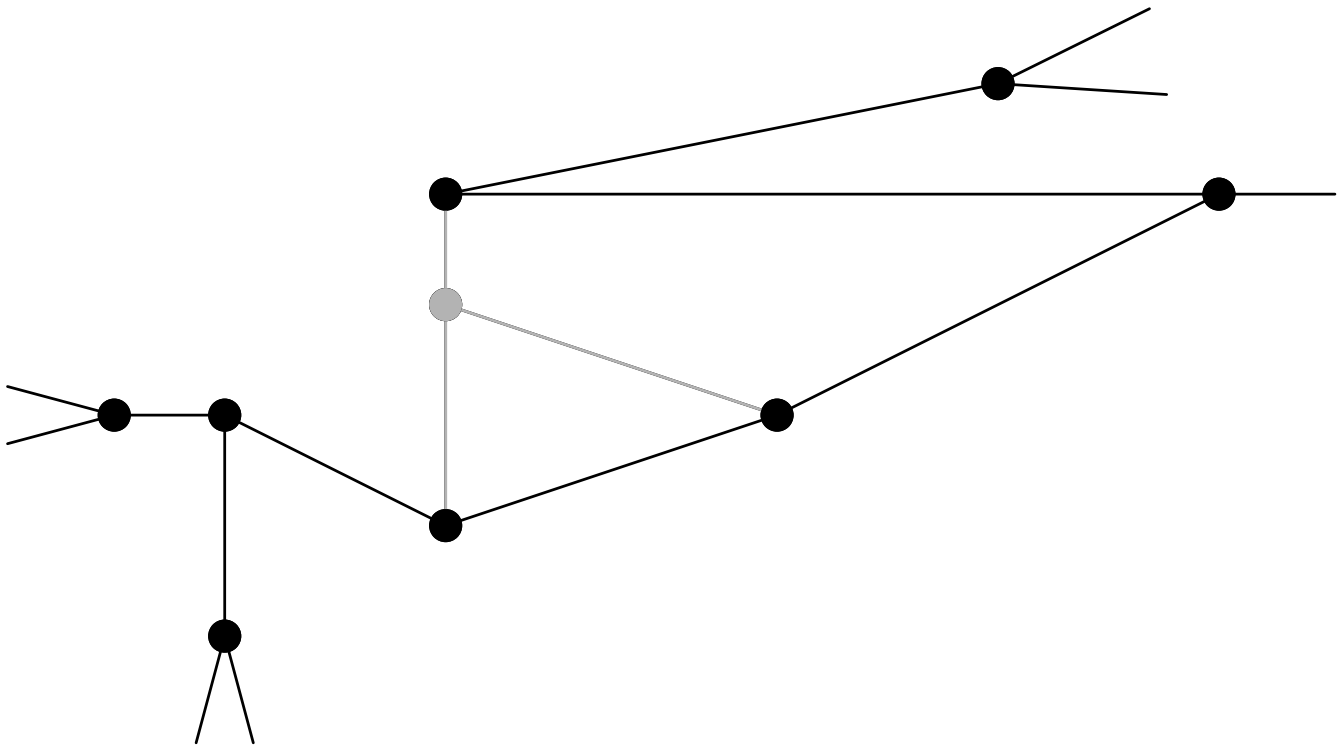
Back to Algorithm $\mathcal{IS}$

Worst case: $\Delta(G) = 3$, $\tau(1, 4)$

$\delta(G) = \Delta(G) = 3$

By lemma: $\tau(1, 6) = 1.29$.

Even more:



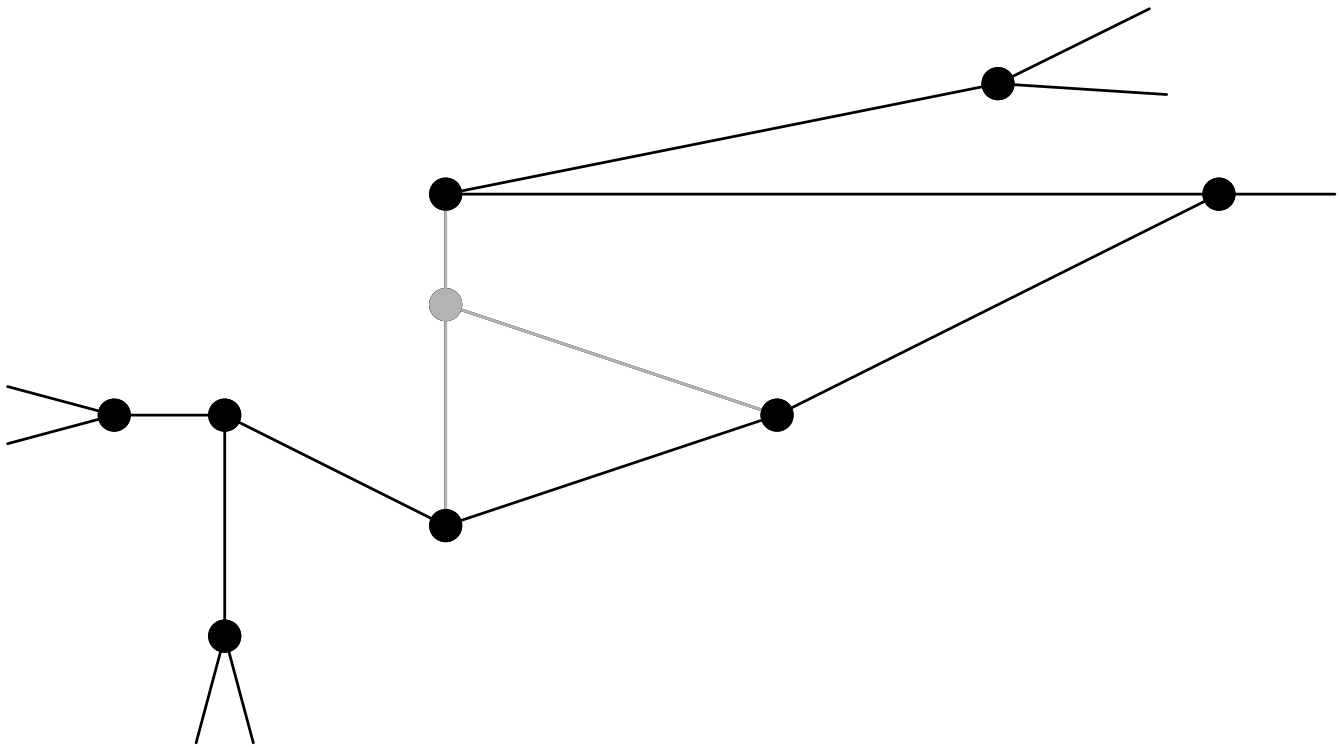
Back to Algorithm $\mathcal{IS}$

Worst case: $\Delta(G) = 3$, $\tau(1, 4)$

$\delta(G) = \Delta(G) = 3$

By lemma: $\tau((1 + 2 + 1), 6) = 1.15$.

Even more:



Back to Algorithm $\mathcal{IS}$

Algorithm $\mathcal{IS}$

Apply folding **and** domination rules;
If $(\deg(v) \leq 1$ **for** some $v \in V)$ **return** $\alpha(G \setminus N[v]) + 1$;
if $(V = \emptyset)$ **return** 0;
Choose $u \in V$ of maximum degree;
return $\max(\mathcal{IS}(G \setminus \{u\}), \mathcal{IS}(G \setminus N[u]) + 1)$;

Runtime bound

- Good if degree is high
- Very good if $\Delta(G) = 3$
- Worst case $\Delta(G) = 4$, but $\Delta(G) = 4$ is not changed by branching in worst-case

Back to Algorithm $\mathcal{IS}$

Algorithm $\mathcal{IS}$

Apply folding **and** domination rules;
If $(\deg(v) \leq 1$ **for** some $v \in V)$ **return** $\alpha(G \setminus N[v]) + 1$;
if $(V = \emptyset)$ **return** 0;
Choose $u \in V$ of maximum degree;
return $\max(\mathcal{IS}(G \setminus \{u\}), \mathcal{IS}(G \setminus N[u]) + 1)$;

Runtime bound

- Good if degree is high
- Very good if $\Delta(G) = 3$
- Worst case $\Delta(G) = 4$, but $\Delta(G) = 4$ is not changed by branching in worst-case

Runtime bound: $O^*(\tau(1,5)^n) = O^*(1.33^n)$

Dominating Set

Let $G = (V, E)$. $V' \subseteq V$ is called *dominating* if $N[u] \cap V' \neq \emptyset$ for all $u \in V$.

Dominating Set

Let $G = (V, E)$. $V' \subseteq V$ is called *dominating* if $N[u] \cap V' \neq \emptyset$ for all $u \in V$.

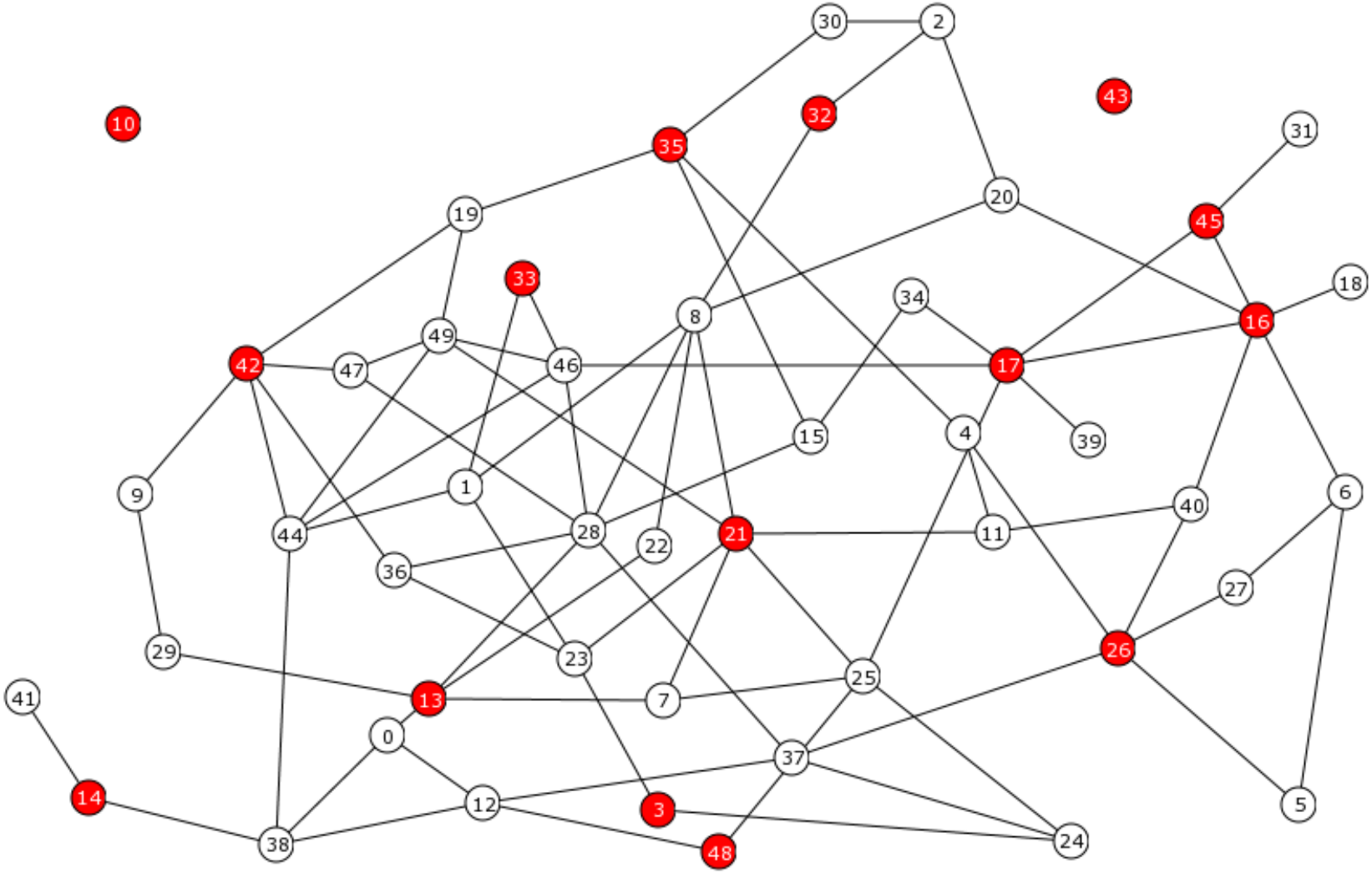
Definition (Dominating Set)

Input: A graph $G = (V, E)$, a number $k \in \mathbb{N}$
Question: Is there a dominating set in G of size k?

Definition

Let $G = (V, E)$. The size of a minimum dominating set in G is denoted by $\gamma(G)$.

Example


30
2
43
10
32
31
35
20
19
45
33
18
34
8
16
49
42
47
46
17
15
4
39
9
1
40
6
44
28
21
11
22
36
27
29
23
26
25
41
13
7
0
37
14
12
3
5
38
24
48

Reduction Rules for Dominating Set

Lemma

Let $G = (V, E)$ be a graph and $\Delta(G) \leq 2$. Then $\gamma(G)$ can be computed in polynomial time.

Lemma

Let $G = (V, E)$ be a graph and $v \in V$ with $\deg(v) = 0$. Then

$$\gamma(G) = \gamma(G \setminus \{v\}) + 1.$$

Reduction Rules for Dominating Set

Lemma

Let $G = (V, E)$ be a graph and $\Delta(G) \leq 2$. Then $\gamma(G)$ can be computed in polynomial time.

Lemma

Let $G = (V, E)$ be a graph and $v \in V$ with $\deg(v) = 0$. Then

$$\gamma(G) = \gamma(G \setminus \{v\}) + 1.$$

Lemma

Let $G = (V, E)$ be a graph and $v \in V$ with $\deg(v) = 1$. Then there is a minimum dominating set V' in G such that $v \notin V'$

Reduction Rules for Dominating Set

Lemma

Let $G = (V, E)$ be a graph and $\Delta(G) \leq 2$. Then $\gamma(G)$ can be computed in polynomial time.

Lemma

Let $G = (V, E)$ be a graph and $v \in V$ with $\deg(v) = 0$. Then

$$\gamma(G) = \gamma(G \setminus \{v\}) + 1.$$

Lemma

Let $G = (V, E)$ be a graph and $v \in V$ with $\deg(v) = 1$. Then there is a minimum dominating set V' in G such that $v \notin V'$

Can we remove v from G?

Problems with Simple Branching Rules

Let V' be a minimum dominating set.

Analysis in $|V|$

- Dominated Nodes cannot be removed
- If we know, $v \in V'$ or $V \notin V'$, v still cannot be removed

Analysis in number of decided nodes does not help.

Problems with Simple Branching Rules

Let V' be a minimum dominating set.

Analysis in $|V|$

- Dominated Nodes cannot be removed
- If we know, $v \in V'$ or $V \notin V'$, v still cannot be removed

Analysis in number of decided nodes does not help.

Analysis in $|N[V']|$

- What if $v \notin V'$?
- Branching Vector $(0, a)$?

Same for an analysis in $|V'|$.

Set Cover

Definition

Let $\mathcal{U}$ be a universe, $S_1, \ldots, S_m \subseteq \mathcal{U}$, and $\mathcal{S} = \{S_1, \ldots, S_m\}$. $C \subseteq \mathcal{S}$ is called set cover if

$$\bigcup_{S \in C} S = \mathcal{U}.$$

Set Cover

Definition

Let $\mathcal{U}$ be a universe, $S_1, \ldots, S_m \subseteq \mathcal{U}$, and $\mathcal{S} = \{S_1, \ldots, S_m\}$. $C \subseteq \mathcal{S}$ is called set cover if

$$\bigcup_{S \in C} S = \mathcal{U}.$$

Definition (Set Cover)

Input:	A family $\mathcal{S} = \{S_1, \ldots, S_m\}$ of subsets of a universe $\mathcal{U}$.
Question:	Is there a set cover of size k for $\mathcal{S}$?

Set Cover

Definition

Let $\mathcal{U}$ be a universe, $S_1, \ldots, S_m \subseteq \mathcal{U}$, and $\mathcal{S} = \{S_1, \ldots, S_m\}$. $C \subseteq \mathcal{S}$ is called set cover if

$$\bigcup_{S \in C} S = \mathcal{U}.$$

Definition (Set Cover)

Input: A family $\mathcal{S} = \{S_1, \ldots, S_m\}$ of subsets of a universe $\mathcal{U}$.
Question: Is there a set cover of size k for $\mathcal{S}$?

Set Cover is a generalization of Dominating Set

$$\mathcal{U} = V,\ S_i = N[v_i].$$

Example

Example

$$\mathcal{U} = \{1, 2, 3 \ldots, 9\}$$

$$\mathcal{S} = \begin{matrix} \{1,2,3\} & \{1,4,7\} & \{5,8,9\} & \{2,3,7,9\} \\ \{1,4,6,8,9\} & \{3,6,7\} & \{7,8\} & \{1\} \end{matrix}$$

Example

Example

$$\mathcal{U} = \{1, 2, 3 \ldots, 9\}$$

$$\mathcal{S} = \begin{matrix} \{1,2,3\} & \{1,4,7\} & \{5,8,9\} & \{2,3,7,9\} \\ \{1,4,6,8,9\} & \{3,6,7\} & \{7,8\} & \{1\} \end{matrix}$$

A solution:

$$\{1,2,3\}, \{5,8,9\}, \{1,4,6,8,9\}, \{3,6,7\}$$

Example

Example

$$\mathcal{U} = \{1, 2, 3 \ldots, 9\}$$

$$\mathcal{S} = \begin{matrix} \{1,2,3\} & \{1,4,7\} & \{5,8,9\} & \{2,3,7,9\} \\ \{1,4,6,8,9\} & \{3,6,7\} & \{7,8\} & \{1\} \end{matrix}$$

A solution:

$$\{1,2,3\}, \{5,8,9\}, \{1,4,6,8,9\}, \{3,6,7\}$$

Optimal solution:

$$\{2,5,8,9\}, \{2,3,7,9\}, \{1,4,6,8,9\}$$

Using Set Cover to solve Dominating Set

Lemma

Dominating Set $\leq$ Set Cover

For each $v \in V$

- $v \in \mathcal{U}$ (v must be dominated)
- $N[v] \in \mathcal{S}$ (v dominates $N[v]$)

Using Set Cover to solve Dominating Set

Lemma

Dominating Set $\leq$ Set Cover

For each $v \in V$

- $v \in \mathcal{U}$ (v must be dominated)
- $N[v] \in \mathcal{S}$ (v dominates $N[v]$)

Lemma

Let $G = (V, E)$ be a graph. There is a instance $(\mathcal{U}, \mathcal{S})$ for Set Cover such that

- *G has a dominating set of size k iff $(\mathcal{U}, \mathcal{S})$ has a set cover of size k*
- $|\mathcal{U}| \leq |V|$
- $|\mathcal{S}| \leq |V|$
- *$(\mathcal{U}, \mathcal{S})$ can be computed in polynomial time*

Reduction Rules

Lemma

Let $(\mathcal{U}, \mathcal{S})$ be a set cover instance and $u \in \mathcal{U}$ such that $\{S \in \mathcal{S} \mid u \in S\} = \{S_j\}$.
Then S_j is contained in any set cover for $\mathcal{S}$.

Reduction Rules

Lemma

Let $(\mathcal{U}, \mathcal{S})$ be a set cover instance and $u \in \mathcal{U}$ such that $\{S \in \mathcal{S} \mid u \in S\} = \{S_j\}$.
Then S_j is contained in any set cover for $\mathcal{S}$.

Lemma

Let $(\mathcal{U}, \mathcal{S})$ be a set cover instance and $S_1, S_2 \in \mathcal{S}$ such that $S_1 \subseteq S_2$.
Then $(\mathcal{U}, \mathcal{S})$ has a set cover of size $k < |\mathcal{S}|$ if and only if $(\mathcal{U}, \mathcal{S} \setminus \{S_1\})$ has a set cover of size k.

Equivalence to low Degree?

Very useful:

Lemma

Let $(\mathcal{U}, \mathcal{S})$ be a set cover instance such that $|S_i| \leq 2$ for all $S_i \in \mathcal{S}$. Then Set Cover *can be solved in polynomial time.*

Equivalence to low Degree?

Very useful:

Lemma

Let $(\mathcal{U}, \mathcal{S})$ be a set cover instance such that $|S_i| \leq 2$ for all $S_i \in \mathcal{S}$. Then Set Cover *can be solved in polynomial time.*

Proof not obvious, later ...

Simple Branching?

$S_i \in C$ or $S_i \notin C$ for some solution.

Analysis in?

Analysis in $|\mathcal{S}|$

Only yields $(1, 1)$

Analysis in $|\mathcal{U}|$

Only yields $(0, a)$?

Just like Dominating Set.

Analysis in?

Analysis in $|\mathcal{S}|$

Only yields $(1, 1)$

Analysis in $|\mathcal{U}|$

Only yields $(0, a)$?

Just like Dominating Set.

Solution:

Analysis in $|\mathcal{U}| + |\mathcal{S}|$.

A simple Algorithm for Set Cover

Wlog: $\mathcal{U} = \bigcup_{S \in \mathcal{S}} S$. Input: $\mathcal{S}$

Algorithm $\mathcal{SC}$

If $(\mathcal{U} = \emptyset)$ **return** 0;
Remove all $s \in \bigcup_{S \in \mathcal{S}} S \setminus \mathcal{U}$;
Apply reduction rules;
Choose a $S \in \mathcal{S}$ of maximum cardinality;
If $|S| = 2$ then solve instance in polynomial time;
else return $\min(\mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U}), \mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U} \setminus S) + 1)$;

A simple Algorithm for Set Cover

Wlog: $\mathcal{U} = \bigcup_{S \in \mathcal{S}} S$. Input: $\mathcal{S}$

Algorithm $\mathcal{SC}$

If $(\mathcal{U} = \emptyset)$ **return** 0;
Remove all $s \in \bigcup_{S \in \mathcal{S}} S \setminus \mathcal{U}$;
Apply reduction rules;
Choose a $S \in \mathcal{S}$ of maximum cardinality;
If $|S| = 2$ then solve instance in polynomial time;
else return $\min(\mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U}), \mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U} \setminus S) + 1)$;

Reduction Rules (Reminder)

- Pick any $S \in \mathcal{S}$ with $\{S_i \in \mathcal{S} \mid u \in S_i\} = \{S\}$ for some u
- Iteratively remove any $S \in \mathcal{S}$ with $S \subseteq S'$ for some $S' \in \mathcal{S}$

Runtime Analysis

return $\min(\mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U}), \mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U} \setminus S) + 1)$;

$$|S| \geq 3 \Rightarrow \tau(1,3) = 1.39$$

Runtime Bound

$$O^*(1.39^{|\mathcal{S}|+|\mathcal{U}|})$$

Runtime Analysis

return $\min(\mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U}), \mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U} \setminus S) + 1)$;

$$|S| \geq 3 \Rightarrow \tau(1,3) = 1.39$$

Runtime Bound

$$O^*(1.39^{|\mathcal{S}|+|\mathcal{U}|})$$

For Dominating Set: $|V| \leq |\mathcal{U}|$, $|V| \leq |\mathcal{S}|$

Theorem

Dominating Set *can be solved in time* $O^*(1.39^{2n}) = O^*(1.93^n)$.

Missing Lemma

Lemma

Let $(\mathcal{U}, \mathcal{S})$ be a set cover instance such that $|S_i| \leq 2$ for all $S_i \in \mathcal{S}$. Then Set Cover *can be solved in polynomial time.*

Construct graph $G = (V, E)$ as follows:

- $V = \mathcal{U}$
- $E = \mathcal{S}$

Covering all nodes with edges $\Leftrightarrow$ finding a set cover

Edge Cover

Lemma

Minimum Edge Cover *can be solved in polynomial time*

```
Compute maximum matching M;
for each v ∉ M
    add arbitrary edge e to M with v ∈ e
```

- Let E' be a minimum edge cover and M' a maximum matching in E'.
- Let M be a maximum matching in G.
- $|M'| \leq |M|$

- Size of E': $|V| - 2|M'| + |M'| = |V| - |M'|$
- Algorithm: $|V| - 2|M| + |M| = |V| - |M|$

Branching Algorithms in Practice

Implementation of Branching Algorithms

Algorithm $\mathcal{SC}$

If $(\mathcal{U} = \emptyset)$ **return** 0;
Remove all $s \in \bigcup_{S \in \mathcal{S}} S \setminus \mathcal{U}$;
Apply reduction rules;
Choose a $S \in \mathcal{S}$ of maximum cardinality;
If $|S| = 2$ then solve instance in polynomial time;
else return $\min(\mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U}), \mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U} \setminus S) + 1)$;

Implementation straight forward?

Obvious Problems

No duplicated calls

If $(S - \mathcal{IS}(G \setminus N[v]) + 1 > max)$ $max = S - \mathcal{IS}(G \setminus N[v]) + 1$;

Order of Reduction Rules

Apply folding **and** domination rules;
If $(\deg(v) = 1$ **for** some $v \in V)$ **return** $\alpha(G \setminus N[v]) + 1$;

Fast Data Structures

Choose a $S \in \mathcal{S}$ of maximum cardinality;

Recursive Calls

return $\min(\mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U}), \mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U} \setminus S) + 1)$;

return $\max(\mathcal{IS}(G \setminus \{u\}), \mathcal{IS}(G \setminus N[u]) + 1)$;

for each $v \in N[u]$
  if $(S - \mathcal{IS}(G \setminus N[v]) + 1 > max)$ $max = S - \mathcal{IS}(G \setminus N[v]) + 1$;

- At least two new instances
- Obtained by local operations

A Typical Implementation

```
int Algorithms::solve(graph oldG){
        if (G.number_of_nodes()==0) return 0;
        graph g=G;
        node u = G.first_node();
        list<node> first, second;
        first.push(u);
        second = neighborhood(g,u);
        removeNodes(G,first);
        removeNodes(g,second);
        int alpha1=solve(G);
        int alpha2=solve(g);
        if (alpha1> alpha2+1) return alpha1;
        else  return alpha2+1;
}
```

A Typical Implementation

```
int Algorithms::solve(graph oldG){
        if (G.number_of_nodes()==0) return 0;
        graph g=G;
        node u = G.first_node();
        list<node> first, second;
        first.push(u);
        second = neighborhood(g,u);
        removeNodes(G,first);
        removeNodes(g,second);
        int alpha1=solve(G);
        int alpha2=solve(g);
        if (alpha1> alpha2+1) return alpha1;
        else  return alpha2+1;
}
```

What is expensive?

A Typical Implementation

```
int Algorithms::solve(graph oldG){
        if (G.number_of_nodes()==0) return 0;
        graph g=G;
        node u = G.first_node();
        list<node> first, second;
        first.push(u);
        second = neighborhood(g,u);
        removeNodes(G,first);
        removeNodes(g,second);
        int alpha1=solve(G);
        int alpha2=solve(g);
        if (alpha1> alpha2+1) return alpha1;
        else  return alpha2+1;
}
```

What is expensive?

- removeNodes(G,first)? second = neighborhood(g,v)?

A Typical Implementation

```
int Algorithms::solve(graph oldG){
        if (G.number_of_nodes()==0) return 0;
        graph g=G;
        node u = G.first_node();
        list<node> first, second;
        first.push(u);
        second = neighborhood(g,u);
        removeNodes(G,first);
        removeNodes(g,second);
        int alpha1=solve(G);
        int alpha2=solve(g);
        if (alpha1> alpha2+1) return alpha1;
        else  return alpha2+1;
}
```

What is expensive?

- removeNodes(G,first)? second = neighborhood(g,v)?
- g=G!

Using Pointers

```
int Algorithms::solve(graph &oldG){
        if (G.number_of_nodes()==0) return 0;
        node u = G.first_node();
        list<node> first, second;
        first.push(u);
        list<noder> U=removeNodes(G, first);
        int alpha1=solve(G);
        restoreNodes(G,U);
        second = neighborhood(G,u);
        removeNodes(G, second);
        int alpha2=solve(G);
        if (alpha1> alpha2+1) return alpha1;
        else  return alpha2+1;
}
```

- Small overhead
- Many changes $\Leftrightarrow$ good branching
- Reduction Rules difficult

Not all Reduction Rules Decrease the Running Time

Lemma

Let $G = (V, E)$ and let G consist of connected components $G_1, \ldots, G_l$. Then

$$\alpha(G) = \sum_{i=1}^{l} \alpha(G_i).$$

- Extremely powerful if components are big
- Computing $G_1, \ldots, G_l$ takes $O(|E|)$.
- How often does G decompose?

Not all Reduction Rules Decrease the Running Time

Lemma

Let $G = (V, E)$ and let G consist of connected components $G_1, \ldots, G_l$. Then

$$\alpha(G) = \sum_{i=1}^{l} \alpha(G_i).$$

- Extremely powerful if components are big
- Computing $G_1, \ldots, G_l$ takes $O(|E|)$.
- How often does G decompose?

- On random graphs: Only after many nodes are removed
- On special graph classes: Maybe early and often

Not all Reduction Rules Decrease the Running Time

Lemma

Let $G = (V, E)$ be a graph and let $u, v \in V$ such that $N[u] \subseteq N[v]$. Then

$$\alpha(G) = \alpha(G) \setminus \{v\}.$$

- Compare all pairs of nodes?

Not all Reduction Rules Decrease the Running Time

Lemma

Let $G = (V, E)$ be a graph and let $u, v \in V$ such that $N[u] \subseteq N[v]$. Then

$$\alpha(G) = \alpha(G) \setminus \{v\}.$$

- Compare all pairs of nodes?
- Check for all neighbors

Not all Reduction Rules Decrease the Running Time

Lemma

Let $G = (V, E)$ be a graph and let $u, v \in V$ such that $N[u] \subseteq N[v]$. Then

$$\alpha(G) = \alpha(G) \setminus \{v\}.$$

- Compare all pairs of nodes?
- Check for all neighbors

- Nodes of high degree are harder to dominate
- Check only for small degree nodes?

Branch & Bound

Mostly for optimization problems

- Compute bound on remaining instance
- Ignore branch completely if bound cannot improve known solution

Extremely powerful in practice

Branch & Bound

Mostly for optimization problems

- Compute bound on remaining instance
- Ignore branch completely if bound cannot improve known solution

Extremely powerful in practice

Example (Vertex Cover)

Given: Graph G, size c of already computed vertex cover, number k of already selected nodes

```
if V = ∅ then
   if k < c then c = k;
   return c;
Compute 2−Approximation A;
if k + 1/2|A| ≥ |c| return c
select {u, v} ∈ E;
else return min{VC(G \ {u}, k + 1), VC(G \ {v}, k + 1)}
```

Branch & Bound

- Works good if many solutions are worse than the optimum
- Many branches are stopped early

Good Bounds

- Fast to compute
- Often close to optimum
- No guarantee on quality needed

Examples

Independent Set

- $|V| - 1/2|V'|$ for a 2-approximation of a vertex cover

Lower bounds not helpful for stopping, but as starting value

- $|V| - |V'|$ for some vertex cover
- Greedily constructed independent set

Dominating Set

- $|V|/(\Delta(V) + 1)$
- Greedily select nodes of distance 3

Overview

1. Introduction
2. Branching
3. Measure&Conquer
4. Dynamic Programming
5. Local Search
6. Split & List
7. Lower Bounds
8. Inclusion-exclusion
9. Subset Convolution

Measure&Conquer

- Pure analytic technique
- No need to change algorithms
- Introduced for branching algorithms

Basic Idea

- Even out good and bad cases
- Works even if only bad cases occur

In contrast to:

Amortized Analysis

- Potential Φ
- Good case deposits in Φ
- Bad case with draws from Φ
- Order not important, but needs guarantee that a lot good cases occur

Changing Branching Vectors

Even out branching vectors

Just one branching vector

$(1, 5) \approx 1.33$

Several branching vectors for different cases

Case 1 $(4, 4)$
Case 2 $(2, 2)$

Changing Branching Vectors

Even out branching vectors

Just one branching vector

$(1, 5) \approx 1.33$ $(3, 3) \approx 1.26$

Several branching vectors for different cases

Case 1 $(4, 4)$
Case 2 $(2, 2)$

Changing Branching Vectors

Even out branching vectors

Just one branching vector

$(1, 5) \approx 1.33$ $(3, 3) \approx 1.26$

Several branching vectors for different cases

Case 1	$(4, 4)$	Case 1	$(3, 3)$
Case 2	$(2, 2)$	Case 2	$(3, 3)$

Algorithms Revisited

Algorithm $\mathcal{IS}$

Apply folding **and** domination rules;
If $(\deg(v) \leq 1$ **for** some $v \in V)$ **return** $\alpha(G \setminus N[v]) + 1$;
if $(V = \emptyset)$ **return** 0;
Choose $u \in V$ of maximum degree;
return $\max(\mathcal{IS}(G \setminus \{u\}), \mathcal{IS}(G \setminus N[u]) + 1)$;

Worst Case:

- $(1, 5)$ if $\Delta(G) = 4$

Recall:

Definition

Let $c_1, \ldots, c_t \in \mathbb{R}$. A recurrence

$$A_n = c_1 A_{n-1} + c_2 A_{n-2} + \cdots + c_t A_{n-t} \text{ for } n \geq t.$$

is called *linear recurrence with constant coefficients*.

Example

$$A_n = A_{n-2} + 3A_{n-4}$$

Recall:

Definition

Let $c_1, \ldots, c_t \in \mathbb{R}$. A recurrence

$$A_n = c_1 A_{n-1} + c_2 A_{n-2} + \cdots + c_t A_{n-t} \text{ for } n \geq t.$$

is called *linear recurrence with constant coefficients*.

Example

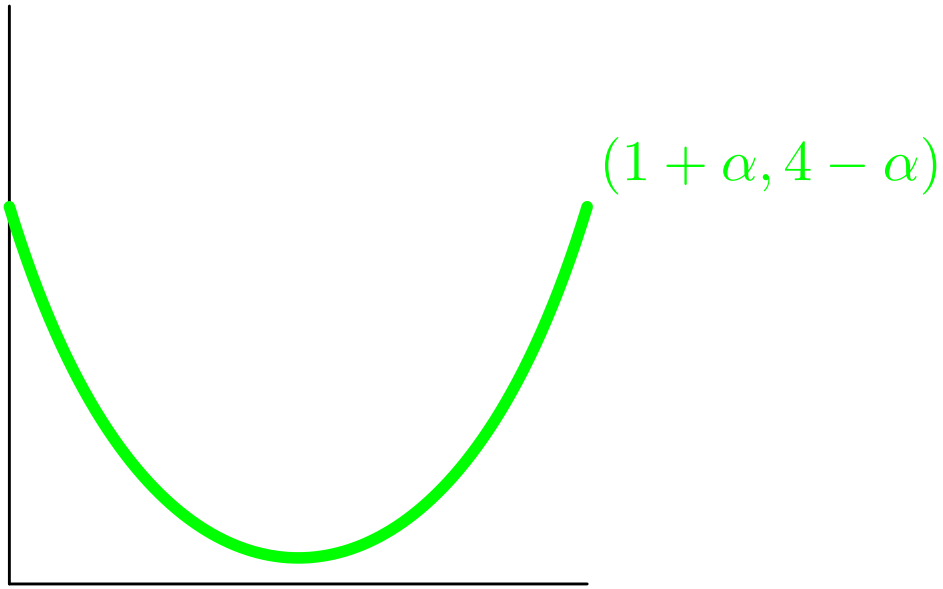
$$A_n = A_{n-2} + 3A_{n-4}$$

But what about this?

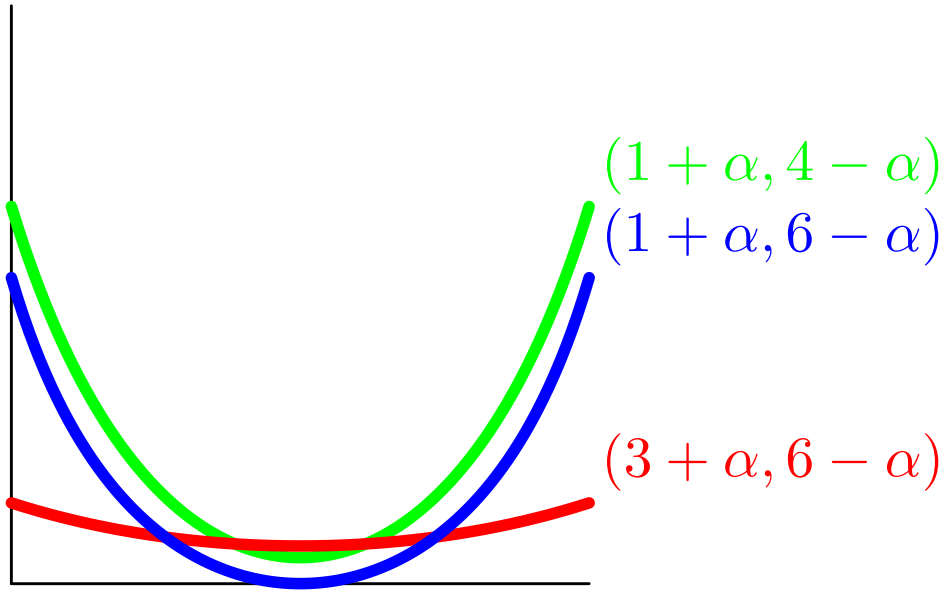
Example

$$A_n = A_{n-2.5} + 3A_{n-3.8}$$

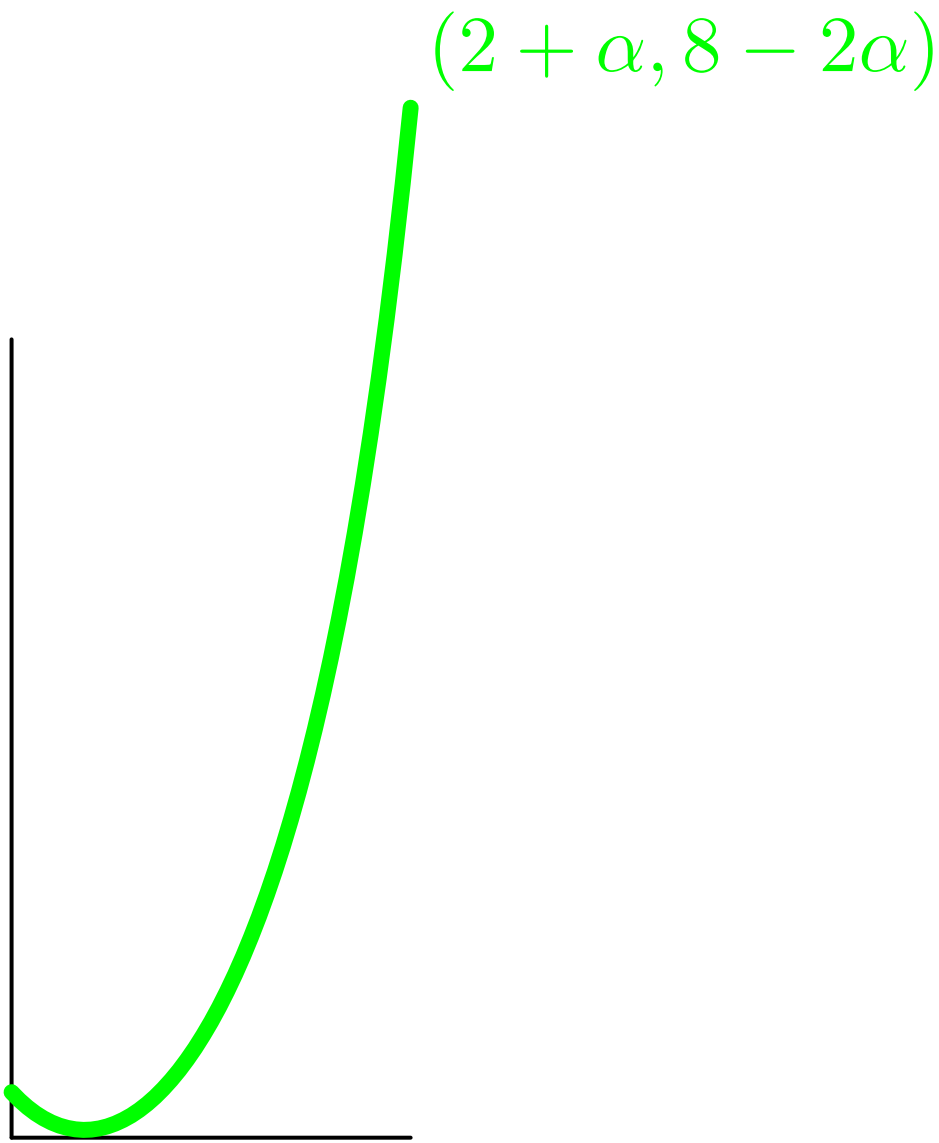
Runtime Bounds



Runtime Bounds



Runtime Bounds



Algorithms Revisited

Algorithm $\mathcal{IS}$

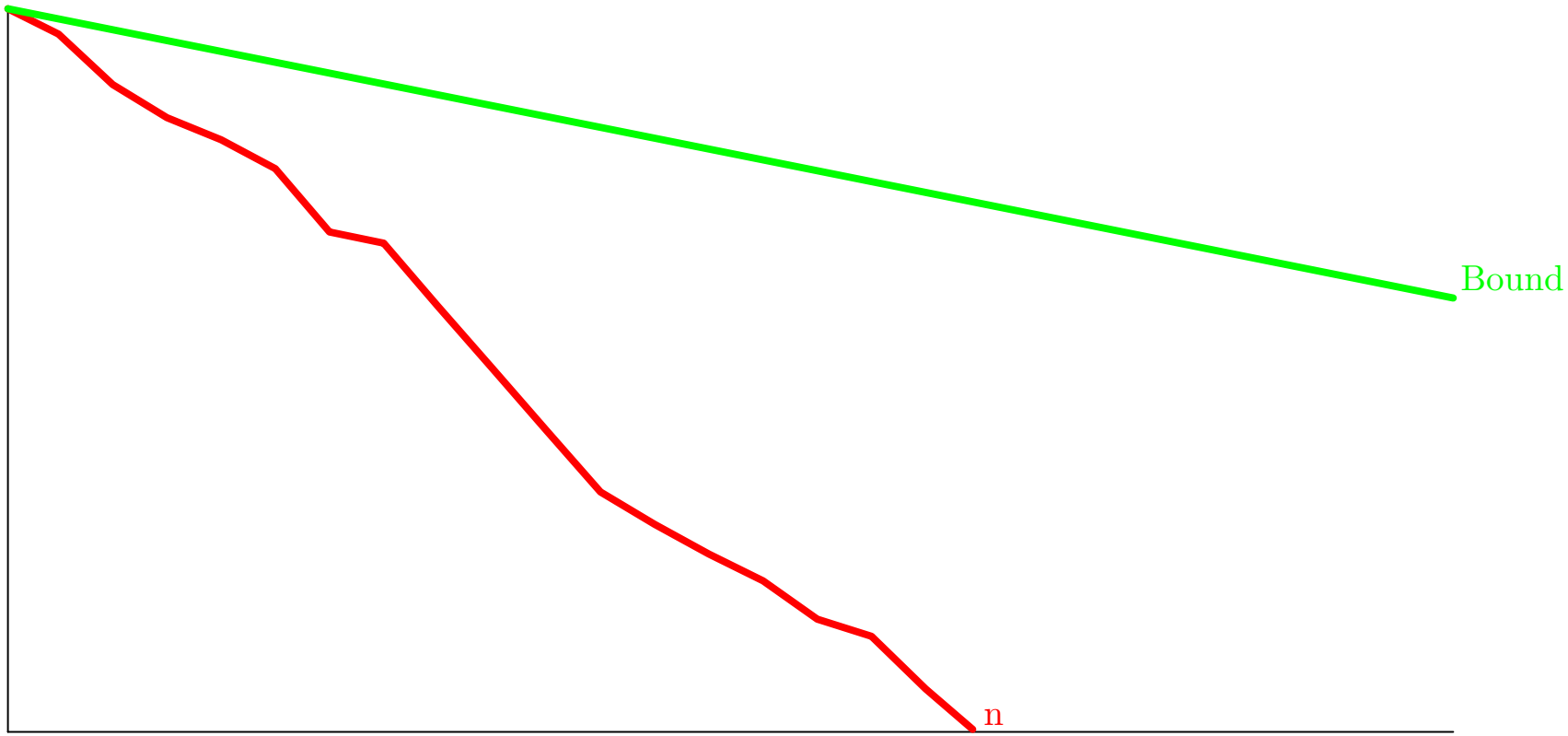
Apply folding **and** domination rules;
If $(\deg(v) \leq 1$ **for** some $v \in V)$ **return** $\alpha(G \setminus N[v]) + 1$;
if $(V = \emptyset)$ **return** 0;
Choose $u \in V$ of maximum degree;
return $\max(\mathcal{IS}(G \setminus \{u\}), \mathcal{IS}(G \setminus N[u]) + 1)$;

Branching Vectors:

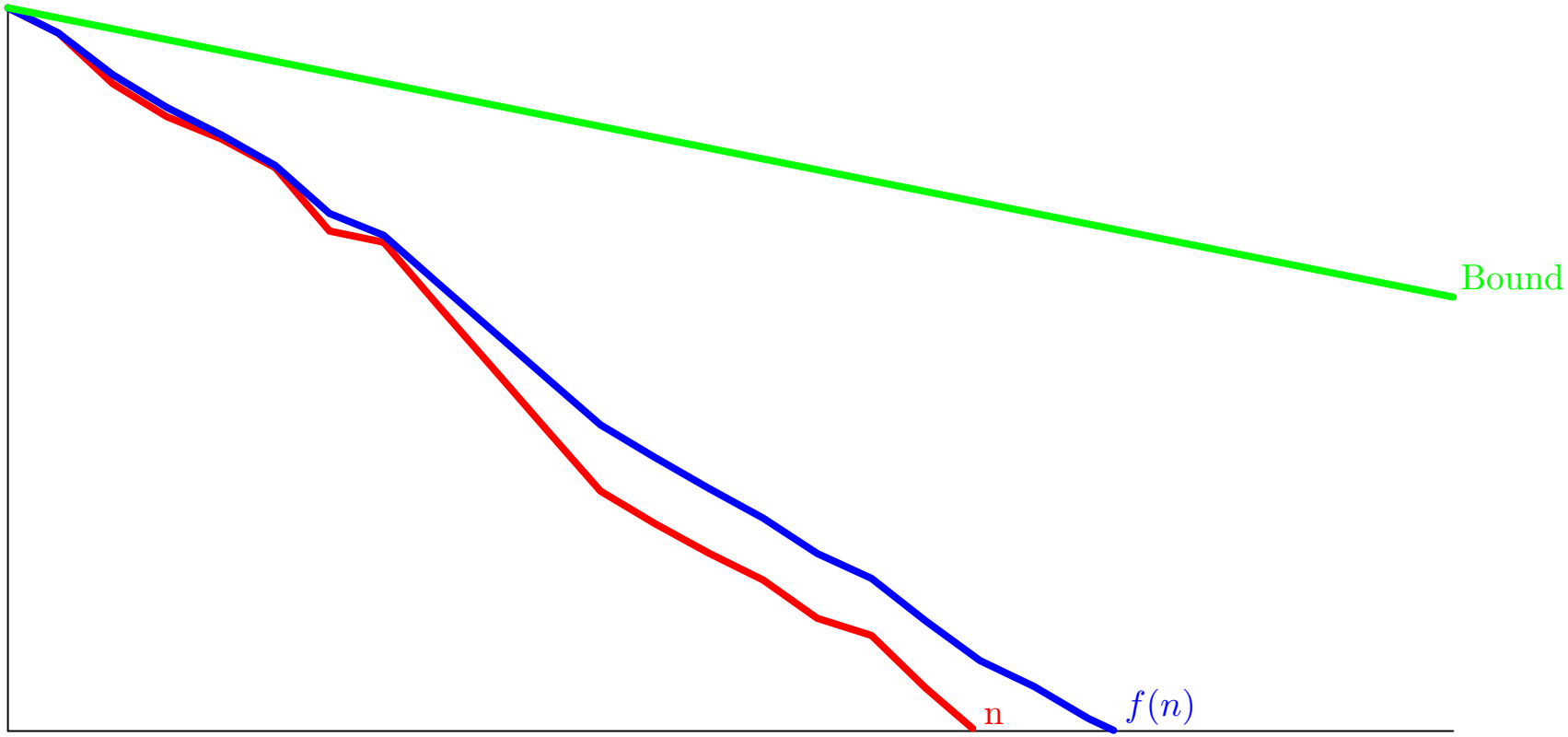
- $(4, 6)$ if $\Delta(G) = 3$
- $(1, 6)$ if $\Delta(G) \geq 5$
- $(1, 5)$ if $\Delta(G) = 4$

Precise analysis of reduction rules necessary

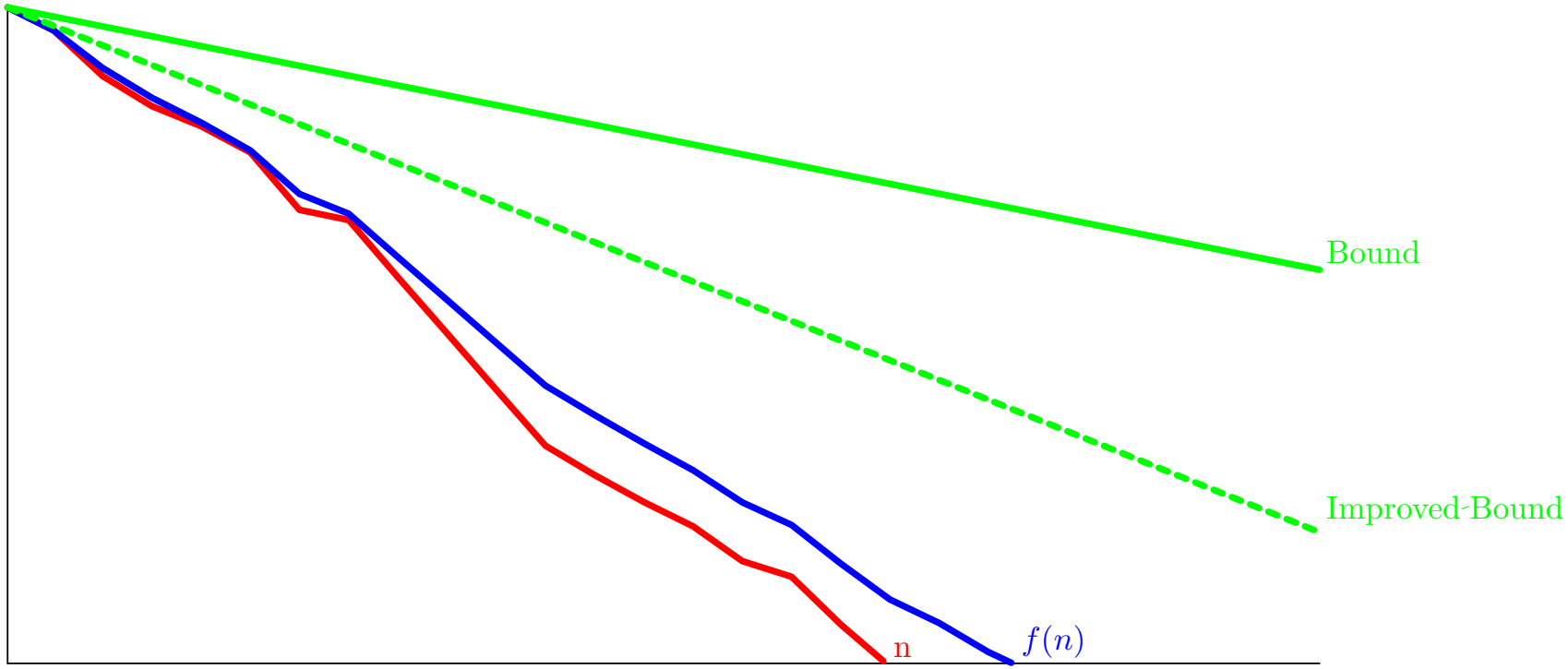
Runtime Bounds



Runtime Bounds



Runtime Bounds



Smoother functions can yield better bounds

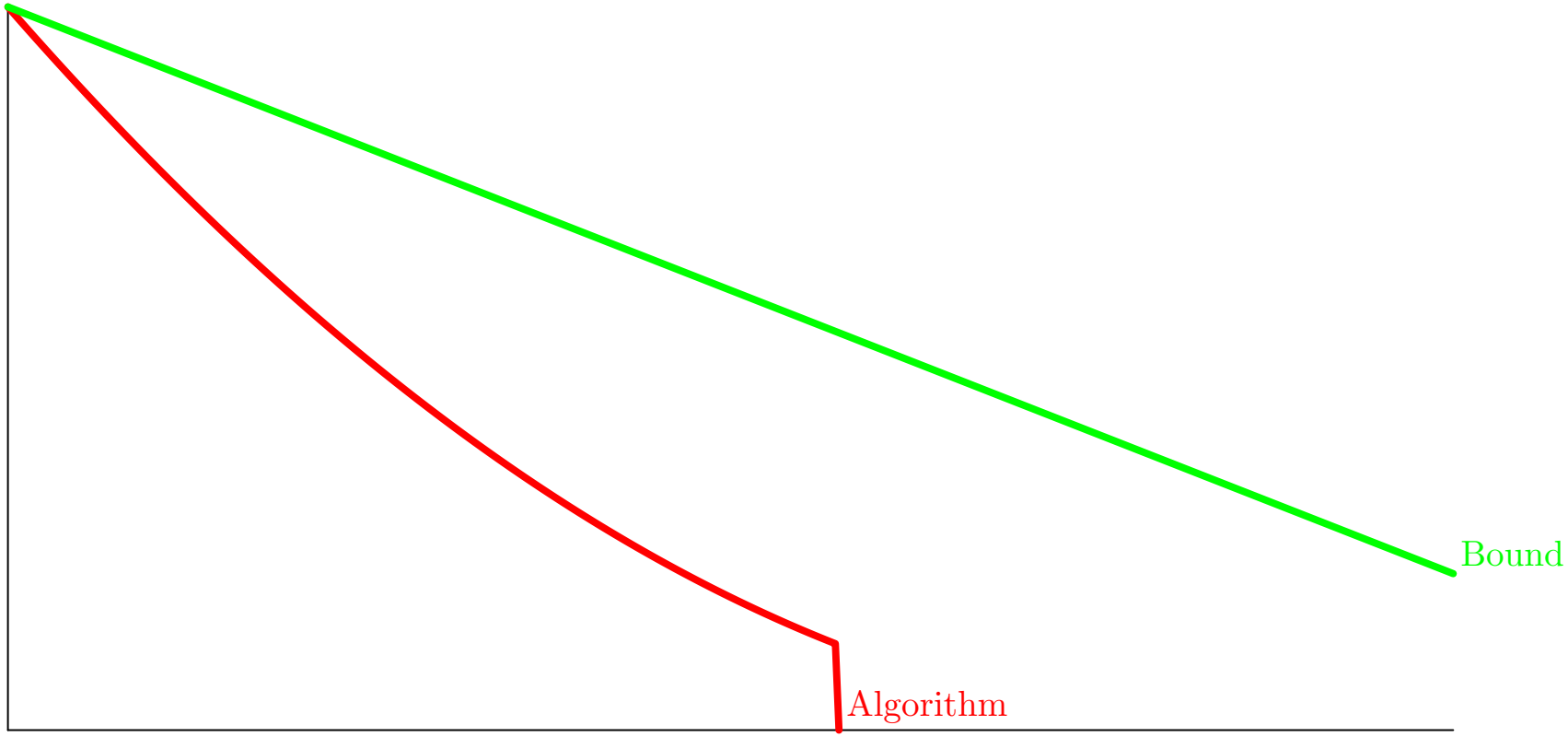
Algorithms Revisited

Algorithm $\mathcal{SC}$

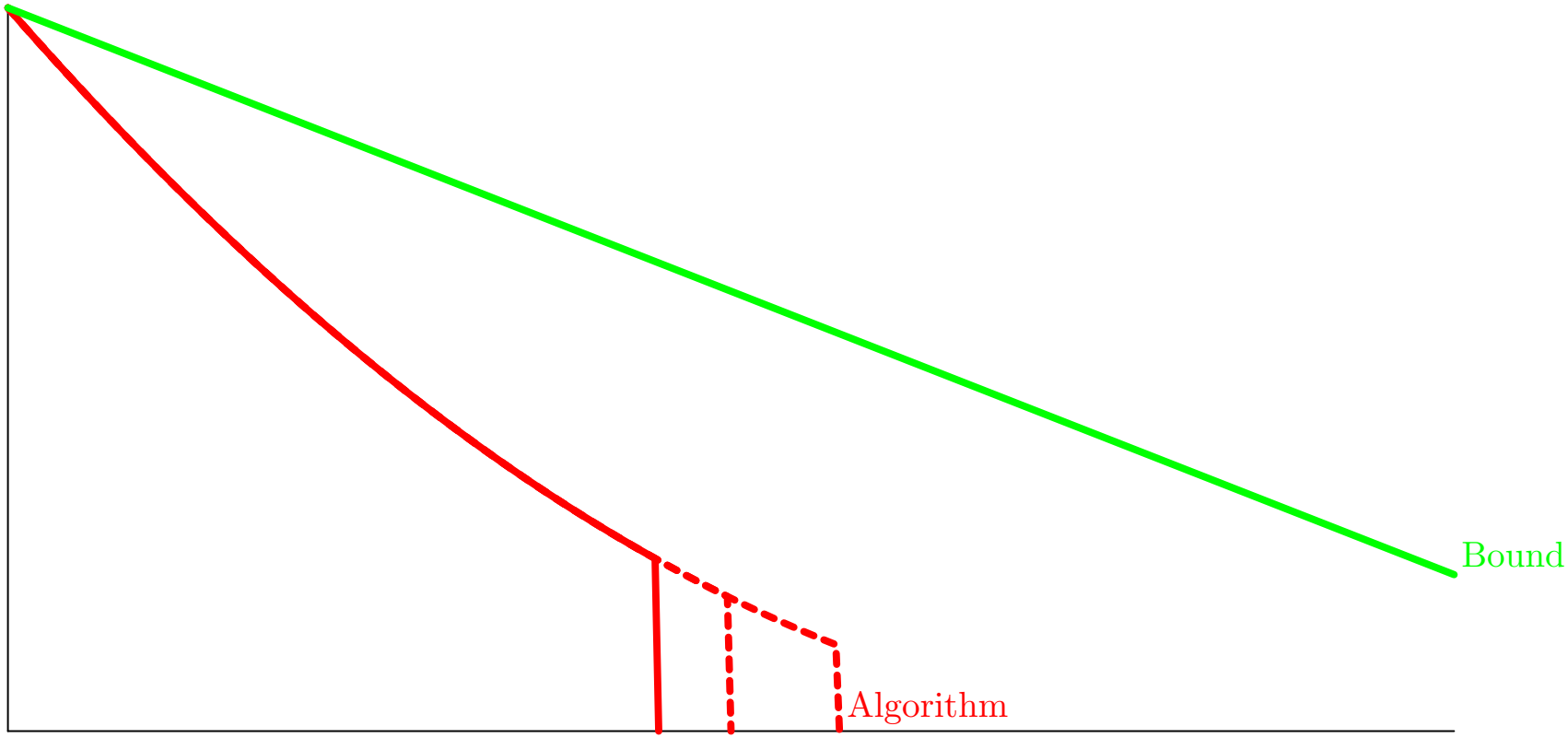
If $(\mathcal{U} = \emptyset)$ **return** 0;
Remove all $s \in \bigcup_{S \in \mathcal{S}} S \setminus \mathcal{U}$;
Apply reduction rules;
Choose a $S \in \mathcal{S}$ of maximum cardinality;
If $|S| = 2$ then solve instance in polynomial time;
else return $\min(\mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U}), \mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U} \setminus S) + 1)$;

- If one S_i is large, branching is good
- If all S_i are small, they are close to reduction rules

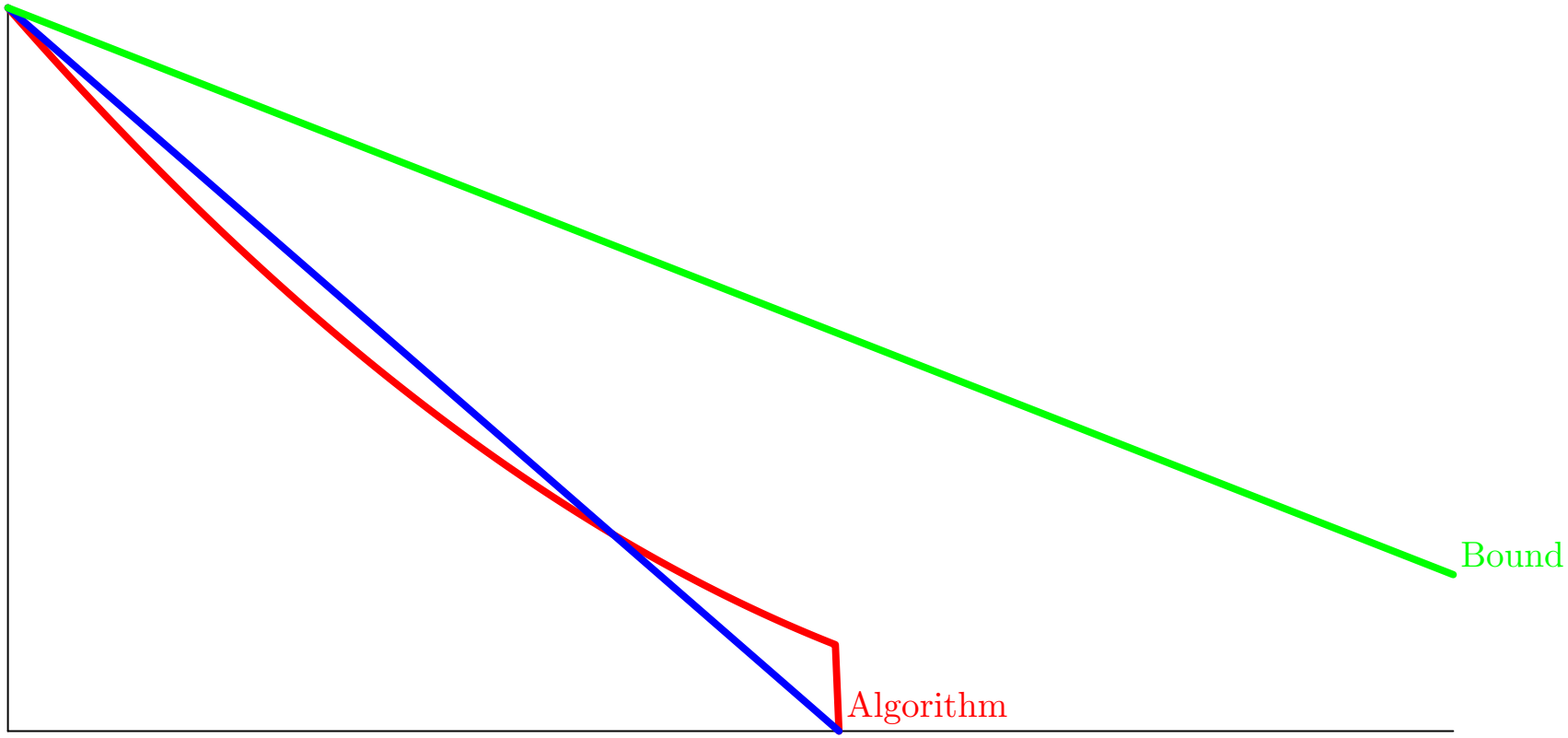
Runtime Bounds



Runtime Bounds



Runtime Bounds



Measure & Conquer

For the set of all input instances I :

- Define measure $\Phi : I \rightarrow \mathcal{R}^{\geq 0}$
- Find bound $\Phi(I) \leq f(I)$
- Analyze branching in Φ

Runtime bound: $c^{\Phi(I)} \leq c^{f(I)}$

- Branching in Φ better or better to analyze
- Never increase Φ!
- Careful with reduction rules

A first Example

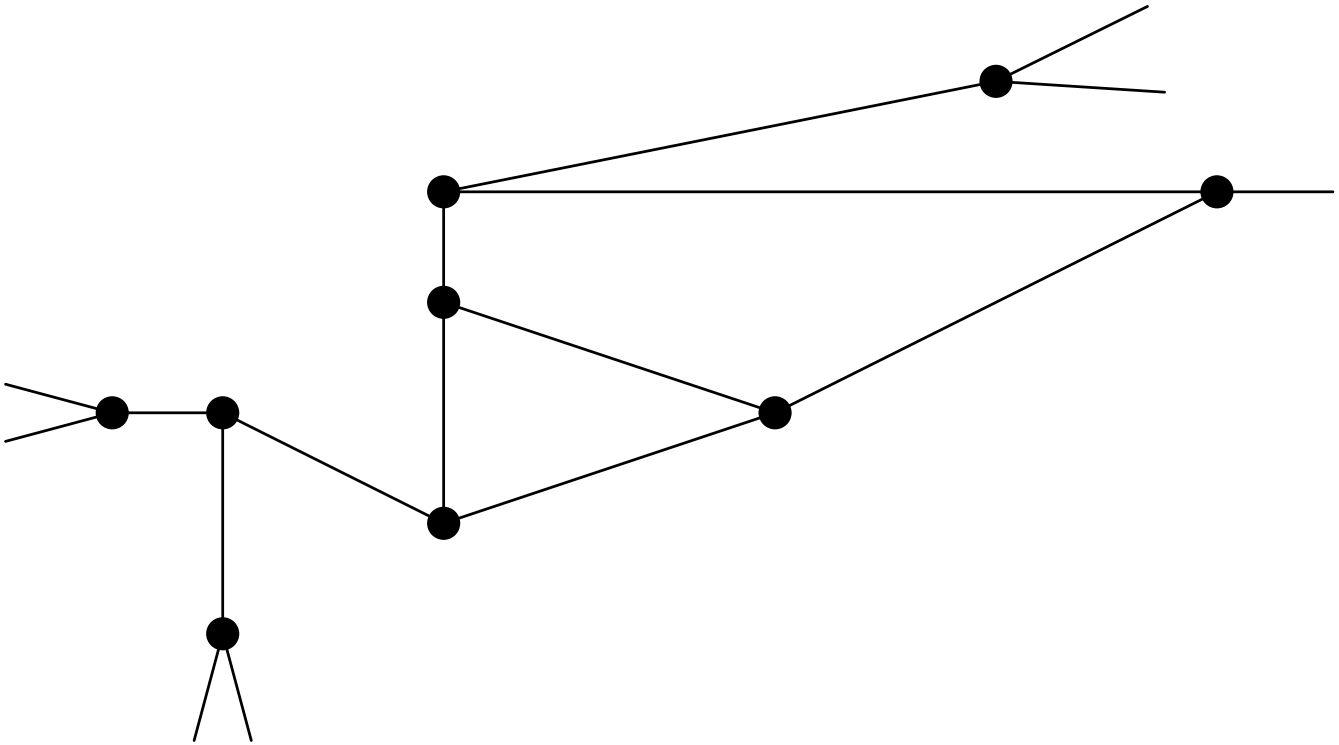
Apply folding **and** domination rules;
If $(\deg(v) \leq 1$ **for** some $v \in V)$ **return** $\alpha(G \setminus N[v]) + 1$;
if $(V = \emptyset)$ **return** 0;
Choose $u \in V$ of maximum degree;
return $\max(\mathcal{IS}(G \setminus \{u\}), \mathcal{IS}(G \setminus N[u]) + 1)$;

Example

$$\Phi(G) = |\{v \in V \mid \deg(v) \geq 3\}|$$

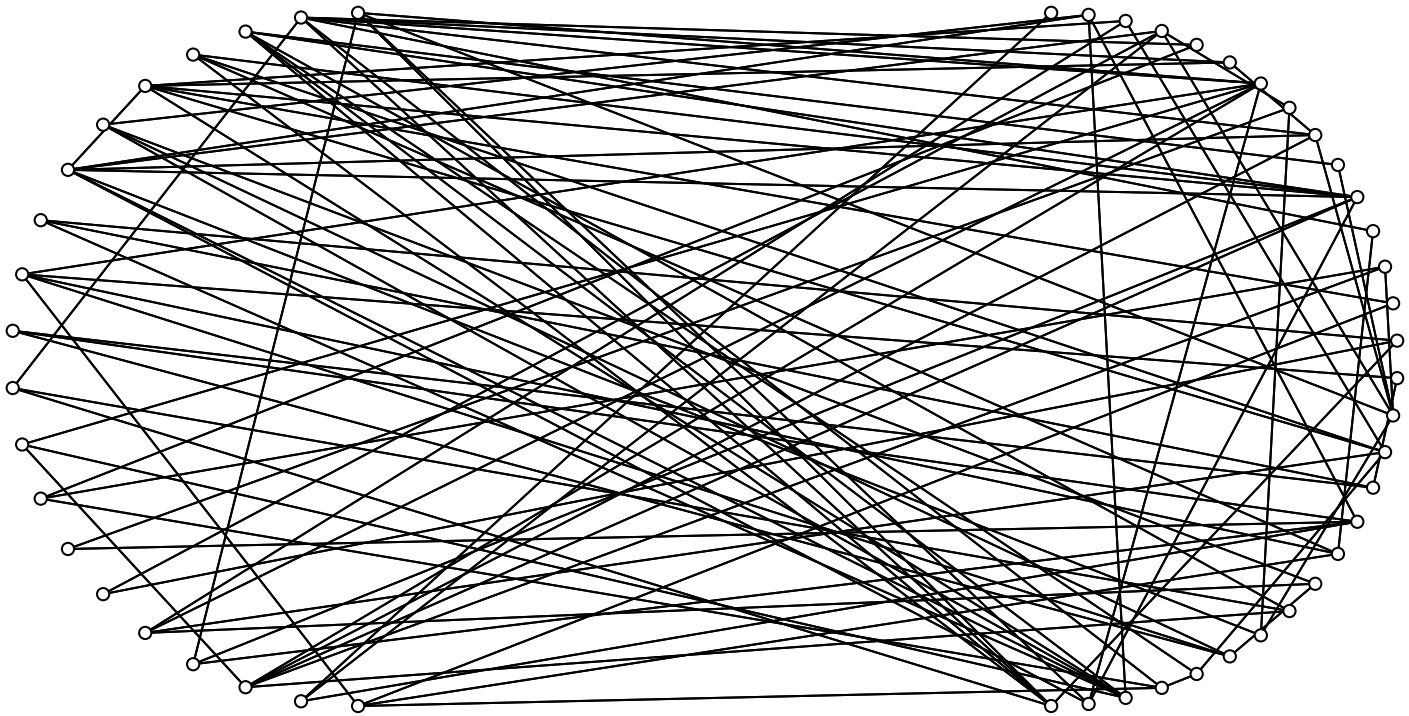
- Reduction rules do not increase Φ (esp. folding)
- If $\Delta(G) \geq 4$, we get $(1, 1 + \Delta(G))$
- If $\Delta(G) = 3$, first branch yields 3 nodes of degree two, second branch at least 2 $\Rightarrow (4, 6)$

$\Delta(G) = 3$



Definition (MaxCut)

Input:	A graph $G = (V, E)$.
Feasible solutions:	Any bipartition $V = V_1 \cup V_2$.
Goal:	Maximize the number of edges that are *cut* by the bipartition.



Branching for MaxCut

Goal: Analysis in $|E|$.

Branching on an edge

- Edge is cut or not?
- How an edge lies in V_1, V_2?

Branching for MaxCut

Goal: Analysis in $|E|$.

Branching on an edge

- Edge is cut or not?
- How an edge lies in V_1, V_2?

Seems not to be better than 2^m

Branching for MaxCut

Goal: Analysis in $|E|$.

Branching on an edge

- Edge is cut or not?
- How an edge lies in V_1, V_2?

Seems not to be better than 2^m

Alternative

- Branch on nodes

Branching for MaxCut

Goal: Analysis in $|E|$.

Branching on an edge

- Edge is cut or not?
- How an edge lies in V_1, V_2?

Seems not to be better than 2^m

Alternative

- Branch on nodes
- $v \in V_1$ or $v \in V_2$

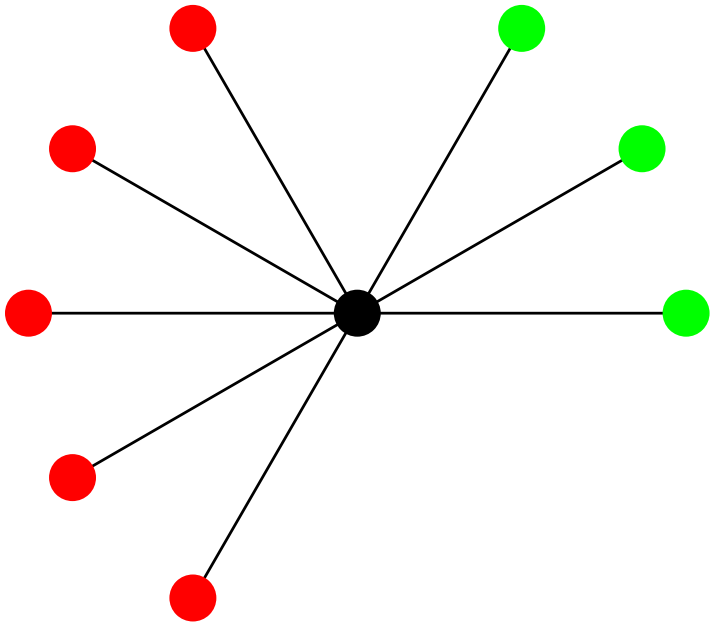
But how do we get an analysis in $|E|$?

- In edges that are completely decided?
- In edges that have on fixed endpoint?

Branching for MaxCut

Partially decided edges, because we gain something in every step

- Branching always (a, a)
- High degree gives good branching



Only partially decided edges $\Rightarrow$ choose greedily

Some Notations

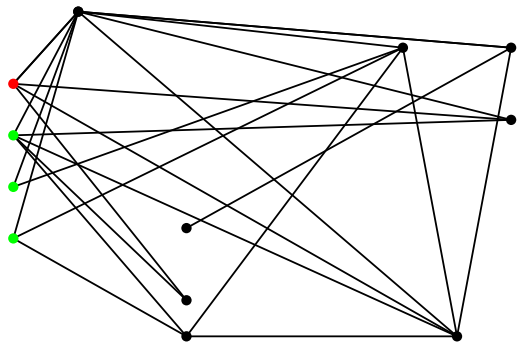
Definition

Let $G = (V, E)$ be a graph and $V_1, V_2 \subseteq V$. If $V_1 \cap V_2 = \emptyset$, we call (V_1, V_2) a (partial) coloring of G.

Definition

Let $G = (V, E)$ be a graph and (V_1, V_2) a (partial) coloring of G. The degree $\deg(v)$ of $v \in V$ is defined as

$$\deg(v) := |\{u \in N(v) \mid u \notin V_1 \cup V_2\}|$$



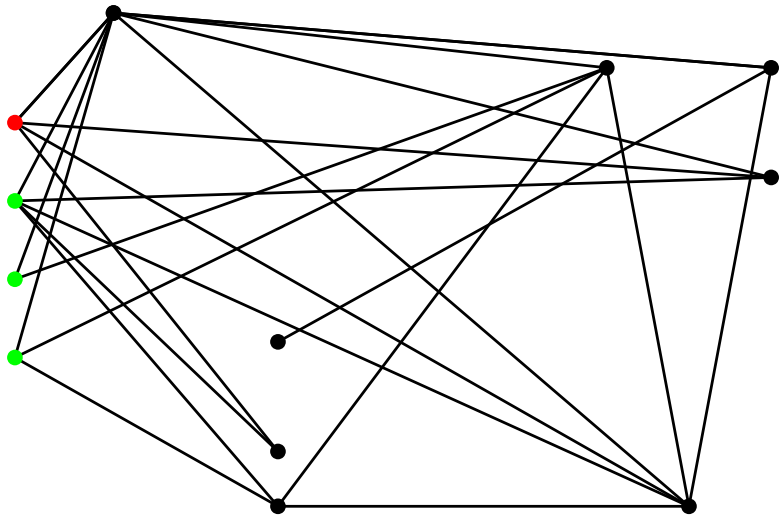
More Notations

Definition

Let $G = (V, E)$ be a graph. A partial coloring (V_1', V_2') is good if there exists a maximum cut (V_1, V_2) of G such that $V_1' \subseteq V_1$ and $V_2' \subseteq V_2$.

Definition

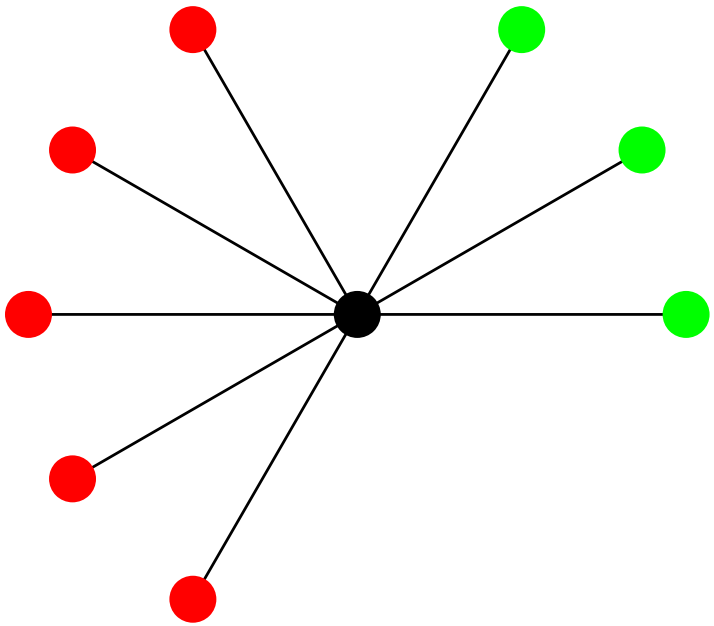
Let $G = (V, E)$ be a graph and (V_1', V_2') a (partial) coloring of G. Then $maxcut(G, V_1', V_2')$ is the size of maximum cut in G that respects (V_1', V_2').



First Reduction Rules

Lemma

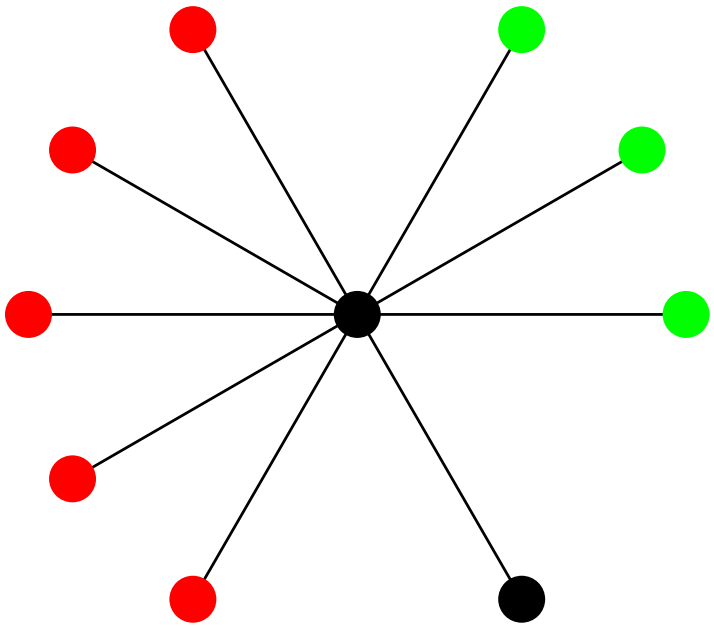
Let $G = (V, E)$ be a graph and (V_1, V_2) a good partial coloring of G. Let $v \in V \setminus V_1 \cup V_2$ with $\deg(v) = 0$.
If $|N(v) \cap V_1| \geq |N(v) \cap V_2|$, then $(V_1, V_2 \cup \{v\})$ is also good.



First Reduction Rules

Lemma

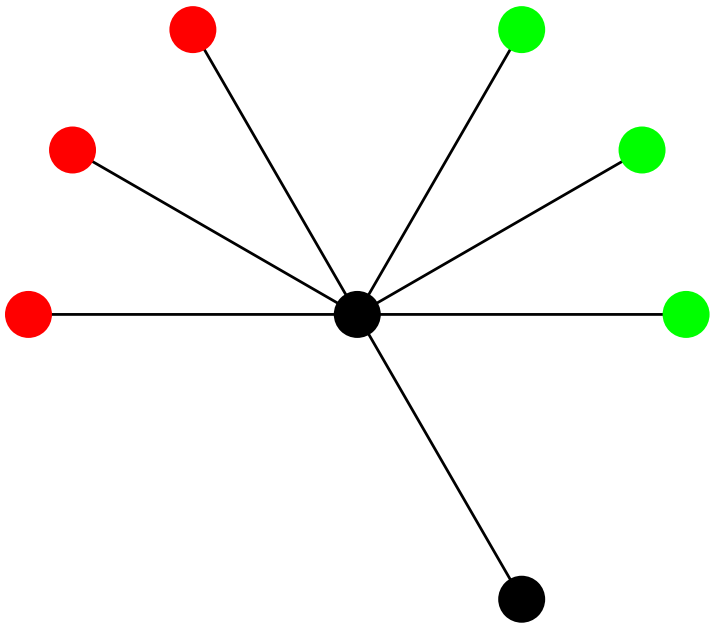
Let $G = (V, E)$ be a graph and (V_1, V_2) a good partial coloring of G. Let $v \in V \setminus V_1 \cup V_2$ with $\deg(v) = 1$.
If $|N(v) \cap V_1| > |N(v) \cap V_2|$, then $(V_1, V_2 \cup \{v\})$ is also good.



First Reduction Rules

Lemma

Let $G = (V, E)$ *be a graph and* (V_1, V_2) *a good partial coloring of* G*. Let* $v \in V \setminus V_1 \cup V_2$ *with* $\deg(v) = 1$*. If* $|N(v) \cap V_1| = |N(v) \cap V_2|$*, then* $maxcut(G, V_1, V_2) = maxcut(G \setminus \{v\}, V_1, V_2) + 1$.



The Algorithm

Algorithm MaxCut(G, V_1, V_2)

```
Apply reduction rules;
Pick a node u of maximum degree;
if deg(u) = 0
   return |{e ∈ E | e ⊈ V1 and e ⊈ V2}|
else
 return max(MaxCut(G, V1 ∪ {u}, V2), MaxCut(G, V1, V2 ∪ {u}))
```

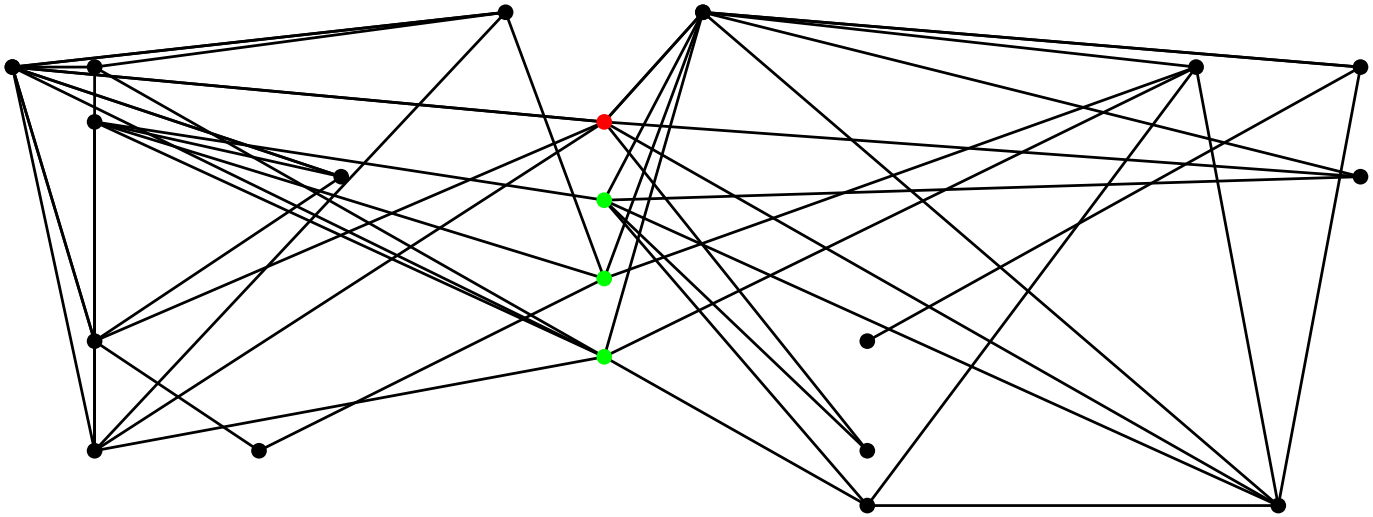
Correctness:

By previous lemmas and complete branching

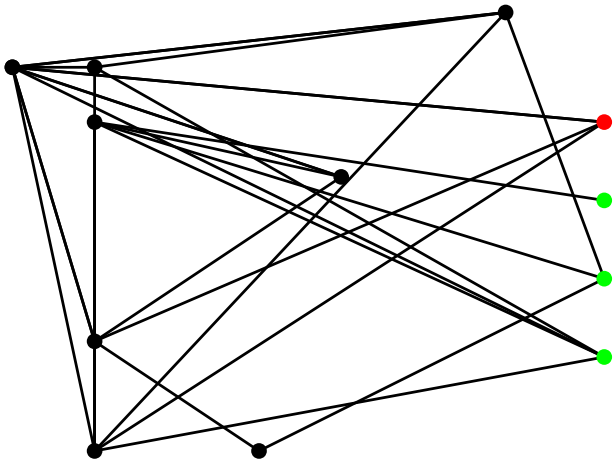
Running time:

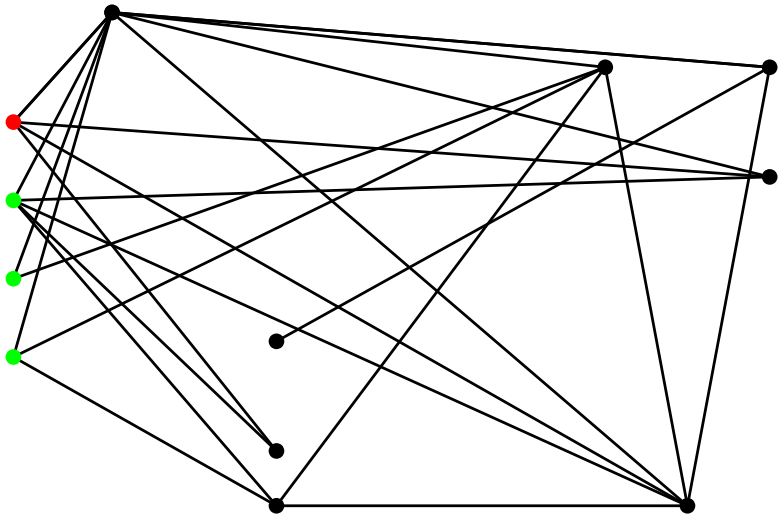
- Trivial measure: $|\{e \in E \mid e \cap (V_1 \cup V_2) \neq \emptyset\}|$
- Branching vector: $(2, 2)$ by reduction rules

More Reductions

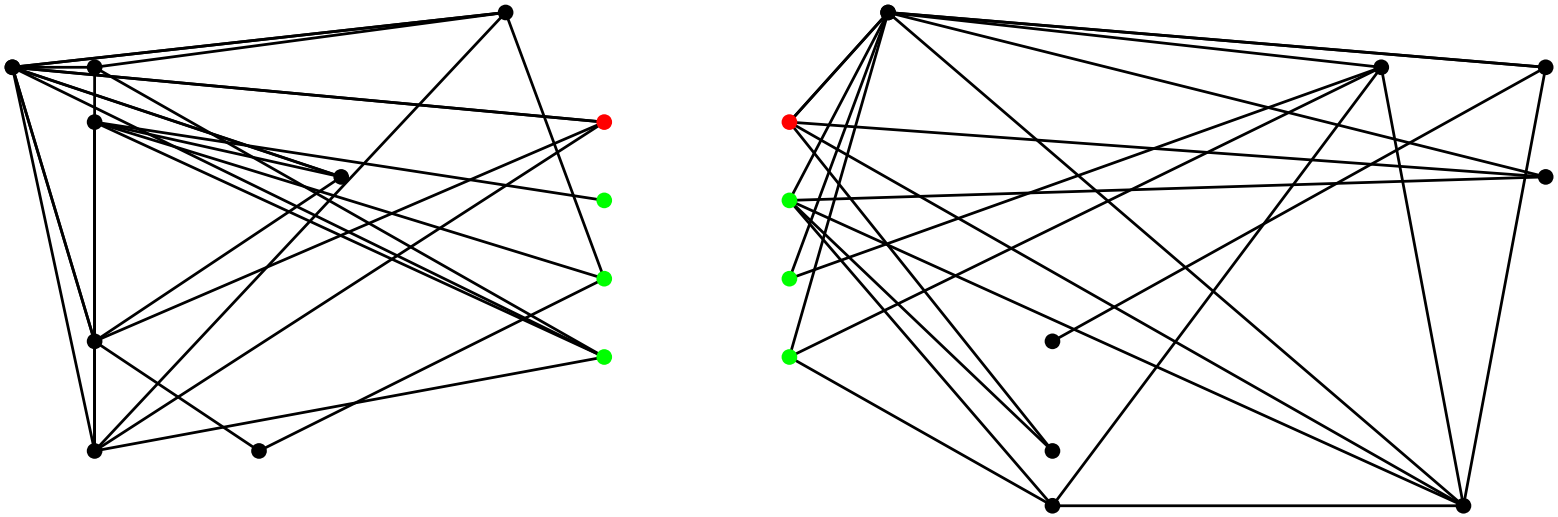


More Reductions





More Reductions



Lemma

Let $G = (V, E)$ be a graph, (V_1, V_2) a good partial coloring of G. If $G \setminus (V_1 \cup V_2)$ consists of components $C_1, \ldots, C_k$, then

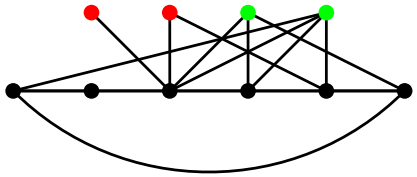
$$maxcut(G, V_1, V_2) = maxcut(G_1, V_1, V_2) + \ldots + maxcut(G_1, V_1, V_2),$$

where $G_i = G[C_i \cup V_1 \cup V_2]$ is the subgraph of G induced by $C_i \cup V_1 \cup V_2$.

Lemma

Let $G = (V, E)$ be a graph, (V_1, V_2) a good partial coloring of G. If $\deg(v) \leq 2$ for all $v \in V \setminus (V_1 \cup V_2)$, then maxcut$(G, V_1, V_2)$ can be computed in polynomial time.

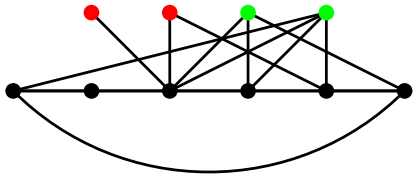
- $G \setminus (V_1 \cup V_2)$ connected
- $\delta(v) \geq 2$ and $\Delta(v) \leq 2$



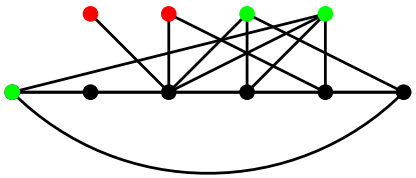
Lemma

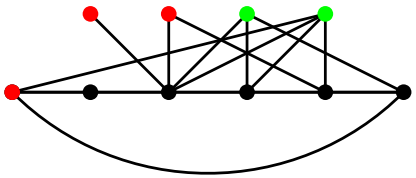
Let $G = (V, E)$ be a graph, (V_1, V_2) a good partial coloring of G. If $\deg(v) \leq 2$ for all $v \in V \setminus (V_1 \cup V_2)$, then maxcut$(G, V_1, V_2)$ can be computed in polynomial time.

- $G \setminus (V_1 \cup V_2)$ connected
- $\delta(v) \geq 2$ and $\Delta(v) \leq 2$



- Branch on one node
- Apply reduction rules





Algorithm MaxCut(G, V_1, V_2)

Apply reduction rules;
Pick a node u of maximum degree;
if $\deg(u) \leq 2$
  compute $maxcut(G, V_1, V_2)$ in polynomial time;
return $\max(\text{MaxCut}(G, V_1 \cup \{u\}, V_2), \text{MaxCut}(G, V_1, V_2 \cup \{u\}))$

Runtime Analysis

- Number of uncolored edges
$$|\{e \in E \mid e \cap (V_1 \cup V_2) = \emptyset\}|$$
- $\deg(u) \geq 3 \Rightarrow$ branching vector $(3, 3)$
- Runtime bounded by $O^*(1.26^{|E|})$

Note:
$$|\{e \in E \mid e \cap (V_1 \cup V_2) = \emptyset\}| = 0.5 \sum_{v \in V} \deg(v)$$

Improvements?

Independent Set

- 3-regular case very good
- Folding removes nodes of degree 2

Measure: $\Phi(G) = |\{v \in V \mid \deg(v) \geq 3\}|$

We never branch on nodes of degree 2. Can we use

$$0.5 \sum_{v \in V, \deg(v) \geq 3} \deg(v)?$$

Improvements?

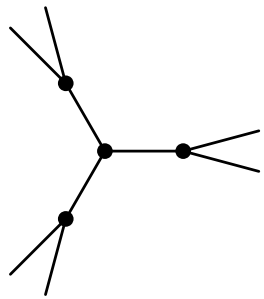
Independent Set

- 3-regular case very good
- Folding removes nodes of degree 2

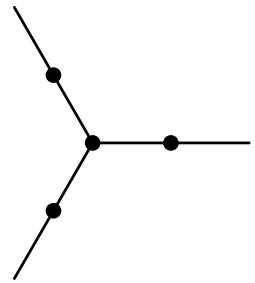
Measure: $\Phi(G) = |\{v \in V \mid \deg(v) \geq 3\}|$

We never branch on nodes of degree 2. Can we use

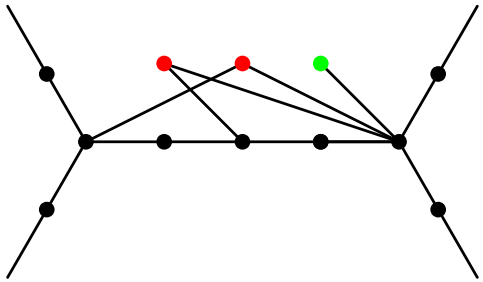
$$0.5 \sum_{v \in V, \deg(v) \geq 3} \deg(v)?$$



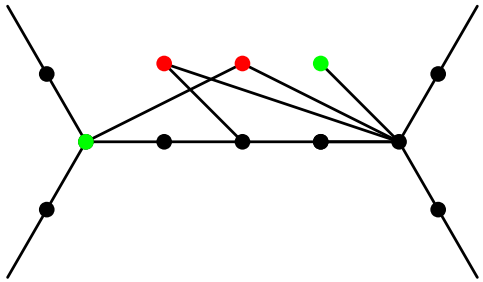
$(6, 6)$ vs $(1.5, 1.5)$



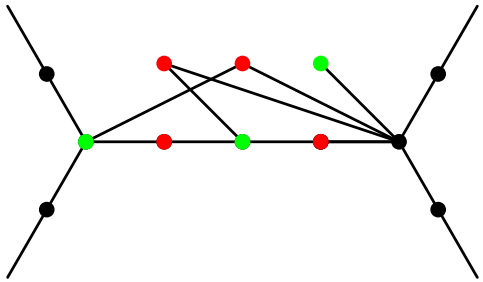
But Nodes of Degree Two are good



But Nodes of Degree Two are good



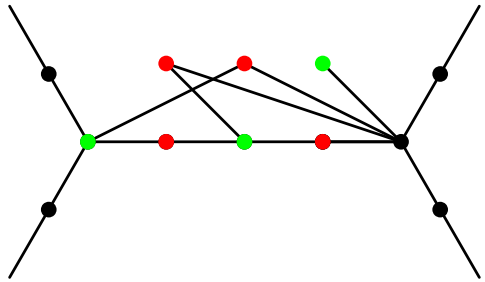
But Nodes of Degree Two are good



After branching on v:

- Neighbors of previous degree 2 are reduced
- Neighbors of previous degree at least 3 remain

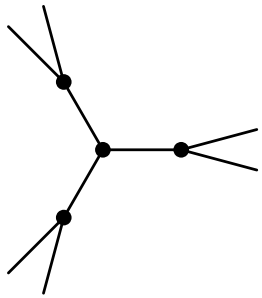
But Nodes of Degree Two are good

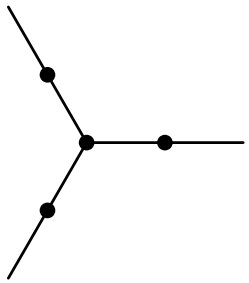


After branching on v:

- Neighbors of previous degree 2 are reduced
- Neighbors of previous degree at least 3 remain

$$0.5 \sum_{v \in V} \deg(v)$$



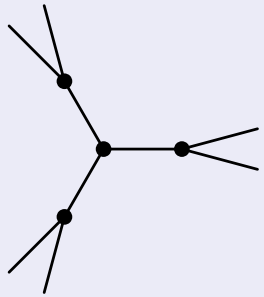


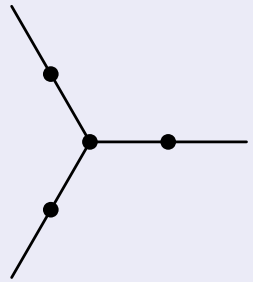
$(3, 3)$ vs $(6, 6)$

If the edges are different!

Different Measures

Depending on measure:





$(3,3)$ vs $(6,6)$

or

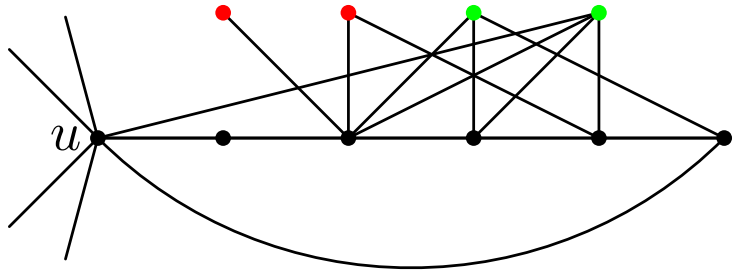
$(6,6)$ vs $(1.5,1.5)$

Use Measure&Conquer to even out

But first:

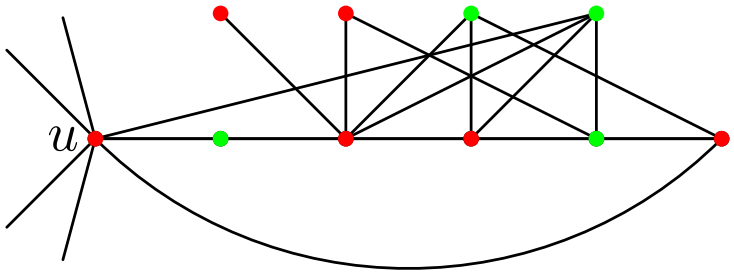
Take care of loops

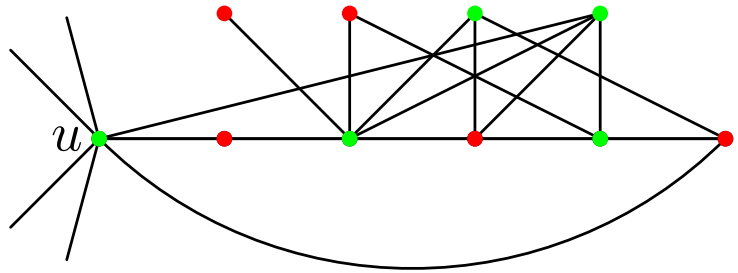
Loops



Observation

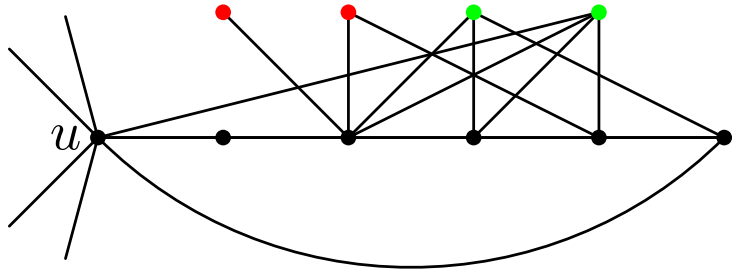
Depending on the side of u, the loop adds a fixed number of cut edges





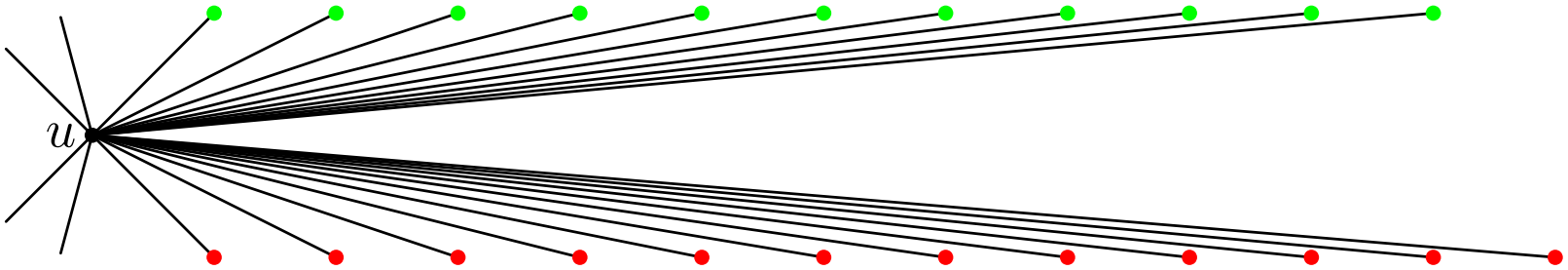
- u: 11 edges
- u: 12 edges

Loops



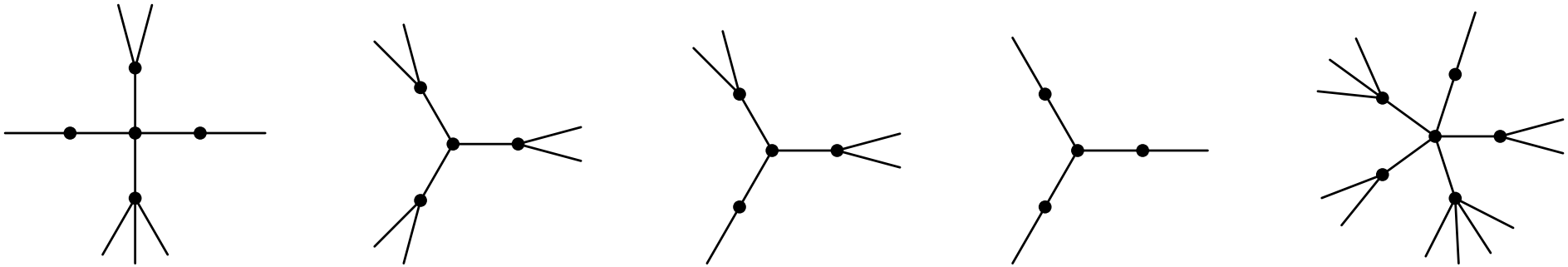
Observation

Depending on the side of u, the loop adds a fixed number of cut edges



- u: 11 edges
- u: 12 edges

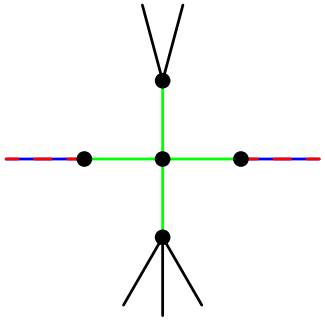
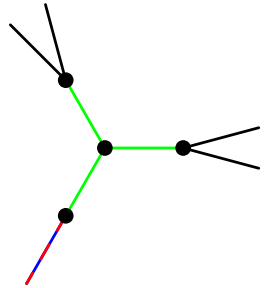
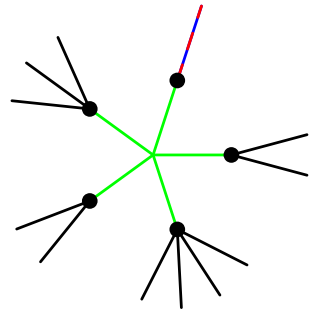
Optimizing the Measure



What do we achieve in each case?

$$\Psi(G) = 0.5 \sum_{v \in V} \deg(v)$$

Optimizing the Measure

What do we achieve in each case?

$$\Psi(G) = 0.5 \sum_{v \in V} \deg(v)$$

G' obtained from G by branching on u and applying reduction rules

$$\Psi(G) - \Psi(G') \quad = \quad 0.5 \sum_{v \in V} \deg(v) - 0.5 \sum_{v \in V'} \deg(v)$$

Until now: Just a complicated way of counting edges

Note

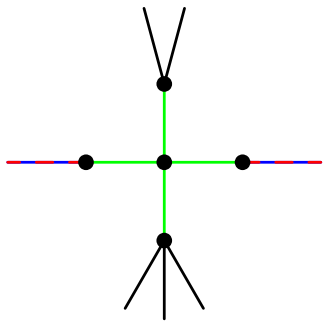
red-blue paths end at nodes of degree at least 3

Optimizing the Measure

$$\Psi(G) - \Psi(G') \quad = \quad 0.5 \sum_{v \in V} \deg(v) - 0.5 \sum_{v \in V'} \deg(v)$$

What do we gain at each node?

- u : $0.5 \deg(u)$
- $v \in N(u)$, $\deg(v) \geq 3$: $0.5(\deg(v) - (\deg(v) - 1))$
- $v \in N(u)$, $\deg(v) = 2$: $0.5(2 - 0)$
- For an endpoint v of a red-blue path:
  a paths ending at v: $0.5(\deg(v) - (\deg(v) - a))$



Optimizing the Measure

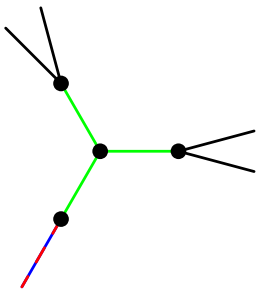
Let $c_i := |\{v \in N(u) \mid \deg(v) = i\}|$. Then $\sum c_i = \deg(u)$.

$$\Psi(G) - \Psi(G') = 0.5\deg(u) + 0.5\sum_{i=3}^{\Delta(G)} c_i + (0.5 + 0.5)c_2 + 0.5c_2$$

Critical Cases

$\Delta(G) = 3$ and $\deg(u) = 3 = c_3 + c_2$

c_2	0	1	2	3
Δ_Ψ	3	4	5	6



Remember: Just count number of colored edges

Optimizing the Measure

Measure

$$\Phi(G) = 0.5 \sum_{v \in V, \deg(v) \geq 3} \deg(v) + 0.5 \sum_{v \in V, \deg(v)=2} \alpha_2$$

with $\alpha_2 \in [0, 2]$.

- u : $0.5 \deg(u)$
- $v \in N(u)$, $\deg(v) \geq 4$: $0.5(\deg(v) - (\deg(v) - 1))$
- $v \in N(u)$, $\deg(v) = 3$: $0.5(3 - \alpha_2)$
- $v \in N(u)$, $\deg(v) = 2$: $0.5\alpha_2$

Optimizing the Measure

But c red-blue paths can end at v

$d = \deg(v) - c$ be the new degree of v.

- $d > 2$: We gain $0.5(\deg(v) - d) = 0.5c$
- $d = 2$: We gain $0.5(\deg(v) - \alpha_2) \geq 0.5(\deg(v) - 2) = 0.5c$
- $d = 1$: We gain $0.5 \deg(v) \geq 0.5c$
- $d = 0$: We gain $0.5 \deg(v) = 0.5c$

At least 0.5 per path

Why does it help?

In total:

$$\Phi(G) - \Phi(G') = 0.5\deg(u) + 0.5\sum_{i=4}^{\Delta(G)} c_i + 0.5(3-\alpha_2)c_i + 0.5\alpha_2 c_2 + 0.5c_2$$

Why does it help?

In total:

$$\Phi(G) - \Phi(G') = 0.5\deg(u) + 0.5\sum_{i=4}^{\Delta(G)} c_i + 0.5(3-\alpha_2)c_i$$
$$+0.5\alpha_2 c_2 + 0.5c_2$$

$\deg(u) = 3,\ c_3 = 3$

$$\Delta_\Phi = 1.5 + 3 \cdot 0.5(3 - \alpha_2)$$

$\deg(u) = 3,\ c_2 = 3$

$$\Delta_\Phi = 1.5 + 3 \cdot 0.5(\alpha_2 + 1)$$

Why does it help?

In total:

$$\Phi(G) - \Phi(G') = 0.5\deg(u) + 0.5\sum_{i=4}^{\Delta(G)} c_i + 0.5(3-\alpha_2)c_i + 0.5\alpha_2 c_2 + 0.5c_2$$

$\deg(u) = 3,\ c_3 = 3$

$$\Delta_\Phi = 1.5 + 3 \cdot 0.5(3 - \alpha_2)$$

$\deg(u) = 3,\ c_2 = 3$

$$\Delta_\Phi = 1.5 + 3 \cdot 0.5(\alpha_2 + 1)$$

For $\alpha_2 = 1$, both yield $\Phi(G) - \Phi(G') = 4.5$
Also for all $c_2 + c_3 = 3 = \deg(u)$:

$$1.5 + c_3 \cdot 0.5(3 - \alpha_2) + c_2 \cdot 0.5(\alpha_2 + 1)$$

General Case

Still missing: higher degree!

$\Delta(G) = 4$

$$0.5 \deg(u) + c_4 \cdot 0.5(4-3) + c_3 \cdot 0.5(3-\alpha_2) + c_2 \cdot 0.5(\alpha_2 + 1)$$

Unfortunately: $c_4 = 4 \Rightarrow 2 + 4 \cdot 0.5 = 4$ and independent of α_2.

$\Delta(G) \geq 5$

Always better than $\Delta(G) = 4$:

- $\deg(u)$ increases
- Gain on higher degree the same: $0.5(4-3)$ vs $0.5(5-4)$
- Gain on degree 2 and 3 remains unchanged
- $\sum c_i$ increases

Runtime Bound

Theorem

MaxCut *can be solved in* $O^*(\tau(4,4)^{|E|})$

For $\alpha_2 = 1$, the worst case of

$$0.5\deg(u) + c_4 \cdot 0.5(4-3) + c_3 \cdot 0.5(3-\alpha_2) + c_2 \cdot 0.5(\alpha_2+1)$$

is $c_4 = 4$. All other cases are clearly better.

- At least better than $\tau(3,3)$.
- Used measure to smooth runtime.

Runtime Bound

Theorem

MaxCut *can be solved in* $O^*(\tau(4,4)^{|E|})$

For $\alpha_2 = 1$, the worst case of

$$0.5\deg(u) + c_4 \cdot 0.5(4-3) + c_3 \cdot 0.5(3-\alpha_2) + c_2 \cdot 0.5(\alpha_2+1)$$

is $c_4 = 4$. All other cases are clearly better.

- At least better than $\tau(3,3)$.
- Used measure to smooth runtime.

But why apply this only to nodes of degree 2?

General Case

New measure:

$$\Phi(G) = \sum_{v \in V} c_i \alpha_i$$

We gain:

$$\alpha_{\deg(u)} + \sum_{i=2}^{\deg(u)} c_i(\alpha_i - \alpha_{i-1}) + c_2 X$$

- u is completely removed
- Neighbors degree decreases by one
- End of red-blue paths yield something X

Note that we can set $\alpha_0 = \alpha_1$ since nodes of degree at most 1 are removed.
This implies that we completely gain α_2 on nodes of degree 2 as before.

General Case

red-blue paths : X?

$$\min\{\alpha_i - \alpha_{i-1}, \alpha_3/3 \mid i \geq 3\}$$

Guaranteed per edge

- If deg ≥ 4, we gain clearly $\alpha_i - \alpha_{i-1}$.
- If deg $= 3$, the first edge gains $\alpha_3 - \alpha_2$
- If there are more, the second gains α_2, the third 0.
  But then, at least $\alpha_3/3$ per edge.
- Minimum per edge:

$$\min\{\alpha_i - \alpha_{i-1}, \alpha_3/3 \mid i \geq 3\}$$

$$\alpha_{\deg(u)} + \sum_{i=2}^{\deg(u)} c_i(\alpha_i - \alpha_{i-1}) + c_2 \min\{\alpha_i - \alpha_{i-1}, \alpha_3/3 \mid i \geq 3\}$$

Computing α_i

Conditions

- $\alpha_0 = \alpha_1 = 0$
- $\alpha_i \leq i/2$, otherwise $\Psi(G) > |E|$
- $\alpha_i \geq \alpha_{i-1}$, otherwise reduction rules increase Ψ

$$
\begin{aligned}
\alpha_3 \quad & +c_3(\alpha_3 - \alpha_2) + c_2\alpha_2 + c_2 \min\{\alpha_3 - \alpha_2, \alpha_3/3\} \\
\alpha_4 \quad & +c_4(\alpha_4 - \alpha_3) + c_3(\alpha_3 - \alpha_2) + c_2\alpha_2 \\
& +c_2 \min\{\alpha_4 - \alpha_3, \alpha_3 - \alpha_2, \alpha_3/3\} \\
\alpha_5 \quad & +c_5(\alpha_5 - \alpha_4) + c_4(\alpha_4 - \alpha_3) + c_3(\alpha_3 - \alpha_2) + c_2\alpha_2 \\
& +c_2 \min\{\alpha_4 - \alpha_3, \alpha_3 - \alpha_2, \alpha_3/3\}
\end{aligned}
$$

Computing α_i

We know (for $\alpha_2 = 0.5, \alpha_i = i/2$):

- Lower bound: 4
- Upper bound: 4.5 ($1.5 + c_3 \cdot (1.5 - \alpha_2) + c_2 \cdot (\alpha_2 + 0.5)$ is balanced)

$$4 < \alpha_4 + c_4(\alpha_4 - \alpha_3) + c_3(\alpha_3 - \alpha_2) + c_2\alpha_2 + c_2 \min\{\alpha_4 - \alpha_3, \alpha_3 - \alpha_2, \alpha_3/3\}$$

implies

- $\alpha_4 - \alpha_3 \geq 0.5$,
- $\alpha_3 - \alpha_2 \geq 0.5$,
- $\alpha_2 + \min\{\alpha_4 - \alpha_3, \alpha_3 - \alpha_2, \alpha_3/3\} \geq 0.5$

If we set $\alpha_5 = 5/2$, we obtain

$$5 \leq \alpha_5 + c_5(\alpha_5 - \alpha_4) + c_4(\alpha_4 - \alpha_3) + c_3(\alpha_3 - \alpha_2) + c_2\alpha_2 + c_2 \min\{\alpha_4 - \alpha_3, \alpha_3 - \alpha_2, \alpha_3/3\}$$

Larger degree also good enough.

Computing α_i

Remaining:

$$\begin{aligned}
\alpha_3 \quad & +c_3(\alpha_3-\alpha_2)+c_2\alpha_2+c_2\min\{\alpha_3-\alpha_2,\alpha_3/3\} \\
\alpha_4 \quad & +c_4(\alpha_4-\alpha_3)+c_3(\alpha_3-\alpha_2)+c_2\alpha_2 \\
& +c_2\min\{\alpha_4-\alpha_3,\alpha_3-\alpha_2,\alpha_3/3\}
\end{aligned}$$

α_4 only positive. Hence: $\alpha_4 = 2$.

Computing α_i

Remaining:

$$\begin{aligned}
\alpha_3 \quad & +c_3(\alpha_3-\alpha_2)+c_2\alpha_2+c_2\min\{\alpha_3-\alpha_2,\alpha_3/3\} \\
\alpha_4 \quad & +c_4(\alpha_4-\alpha_3)+c_3(\alpha_3-\alpha_2)+c_2\alpha_2 \\
& +c_2\min\{\alpha_4-\alpha_3,\alpha_3-\alpha_2,\alpha_3/3\}
\end{aligned}$$

α_4 only positive. Hence: $\alpha_4 = 2$.

This implies $\alpha_4 - \alpha_3 \geq \alpha_3/3$. In every equation:

$$c_3(\alpha_3-\alpha_2) \text{ and } c_2(\alpha_2+\min\{\alpha_3-\alpha_2,\alpha_3/3\})$$

α_2 only depends on α_3

$$\begin{aligned}
\alpha_3-\alpha_2 &= \alpha_2+\min\{\alpha_3-\alpha_2,\alpha_3/3\} \\
\alpha_3-\alpha_2 &= \alpha_2+\alpha_3/3 \\
\alpha_3 &= 3\alpha_2
\end{aligned}$$

$$
\begin{aligned}
\alpha_3 \quad & +c_3(\alpha_3-\alpha_2)+c_2\alpha_2+c_2\min\{\alpha_3-\alpha_2,\alpha_3/3\} \\
\alpha_4 \quad & +c_4(\alpha_4-\alpha_3)+c_3(\alpha_3-\alpha_2)+c_2\alpha_2 \\
& +c_2\min\{\alpha_4-\alpha_3,\alpha_3-\alpha_2,\alpha_3/3\}
\end{aligned}
$$

First equation:α_3 as large as possible
Only one reason to decrease α_3: $(2-\alpha_3)$

Balancing:

$$
\begin{aligned}
2+4(2-\alpha_3) &= \alpha_3+3(\alpha_3-\alpha_2) \\
2+4(2-3\alpha_2) &= 3\alpha_2+3(3\alpha_2-\alpha_2) \\
10-12\alpha_2 &= 3\alpha_2+6\alpha_2 \\
10 &= 21\alpha_2
\end{aligned}
$$

Computing α_i

$$\alpha_2 = 10/21, \alpha_3 = 30/21, \alpha_i = i/2 \text{ for } i \geq 5$$

$$\begin{aligned}
\alpha_3 \quad & +c_3(\alpha_3 - \alpha_2) + c_2\alpha_2 + c_2 \min\{\alpha_3 - \alpha_2, \alpha_3/3\} \\
\alpha_4 \quad & +c_4(\alpha_4 - \alpha_3) + c_3(\alpha_3 - \alpha_2) + c_2\alpha_2 \\
& +c_2 \min\{\alpha_4 - \alpha_3, \alpha_3 - \alpha_2, \alpha_3/3\}
\end{aligned}$$

Verification:

$$\begin{aligned}
30/21 \quad & +c_3(20/21) + c_2(20/21) \\
2 \quad & +c_4(12/21) + c_3(20/21) + c_2(20/21)
\end{aligned}$$

Theorem

MaxCut *can be solved in* $O^*(\tau(90/21, 90/21)^{|E|}) = O^*(1.16^{|E|})$

Compared to $\tau(3, 3)$ without measure!

Independent Set

Why could we compute the measure for MaxCut?

- Branching vectors symmetric
- Balanced single value

In general, this is not so easy.

Algorithm $\mathcal{IS}$

Apply folding **and** domination rules;
If $(\deg(v) \leq 1$ **for** some $v \in V)$ **return** $\alpha(G \setminus N[v]) + 1$;
if $(V = \emptyset)$ **return** 0;
Choose $u \in V$ of maximum degree;
return $\max(\mathcal{IS}(G \setminus \{u\}), \mathcal{IS}(G \setminus N[u]) + 1)$;

$$(4,6), \qquad (1,5), \qquad (1,6)$$

A Measure for Independent Set

Let $d_i := |\{v \in V \mid \deg(v) = i\}|$

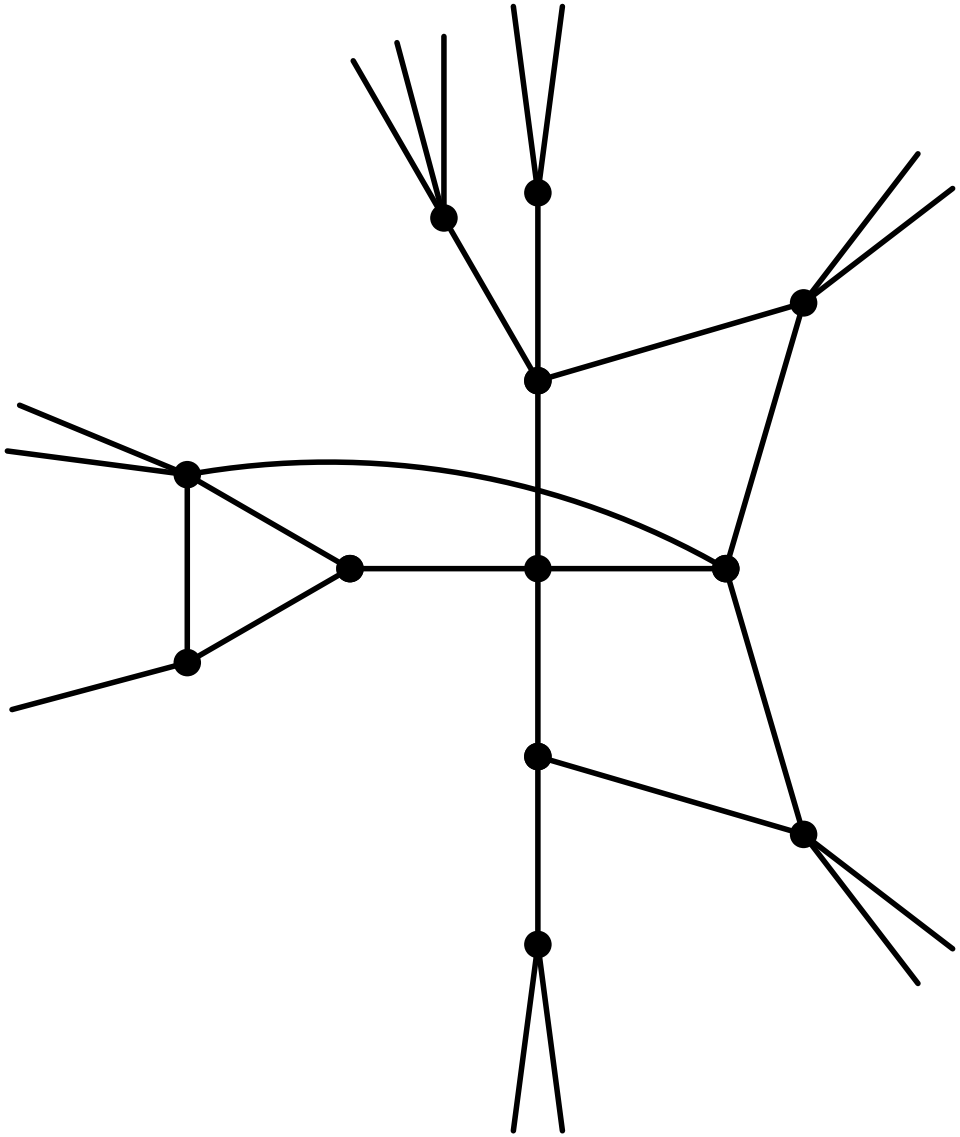
$$\Phi(G) = \sum_{i=1}^{\Delta(G)} d_i \alpha_i$$

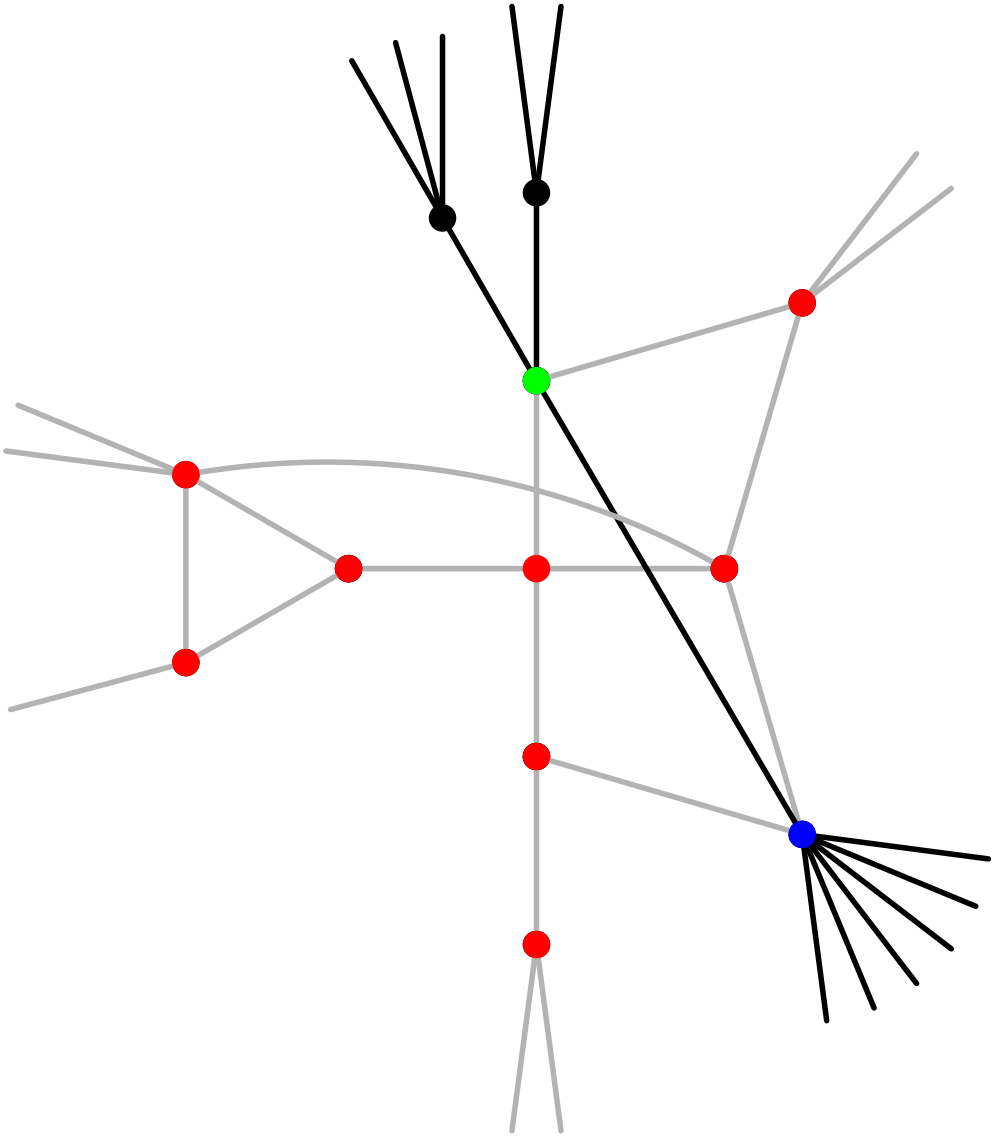
where $\alpha_i = 0$ for $i \leq 2$.

Restraints

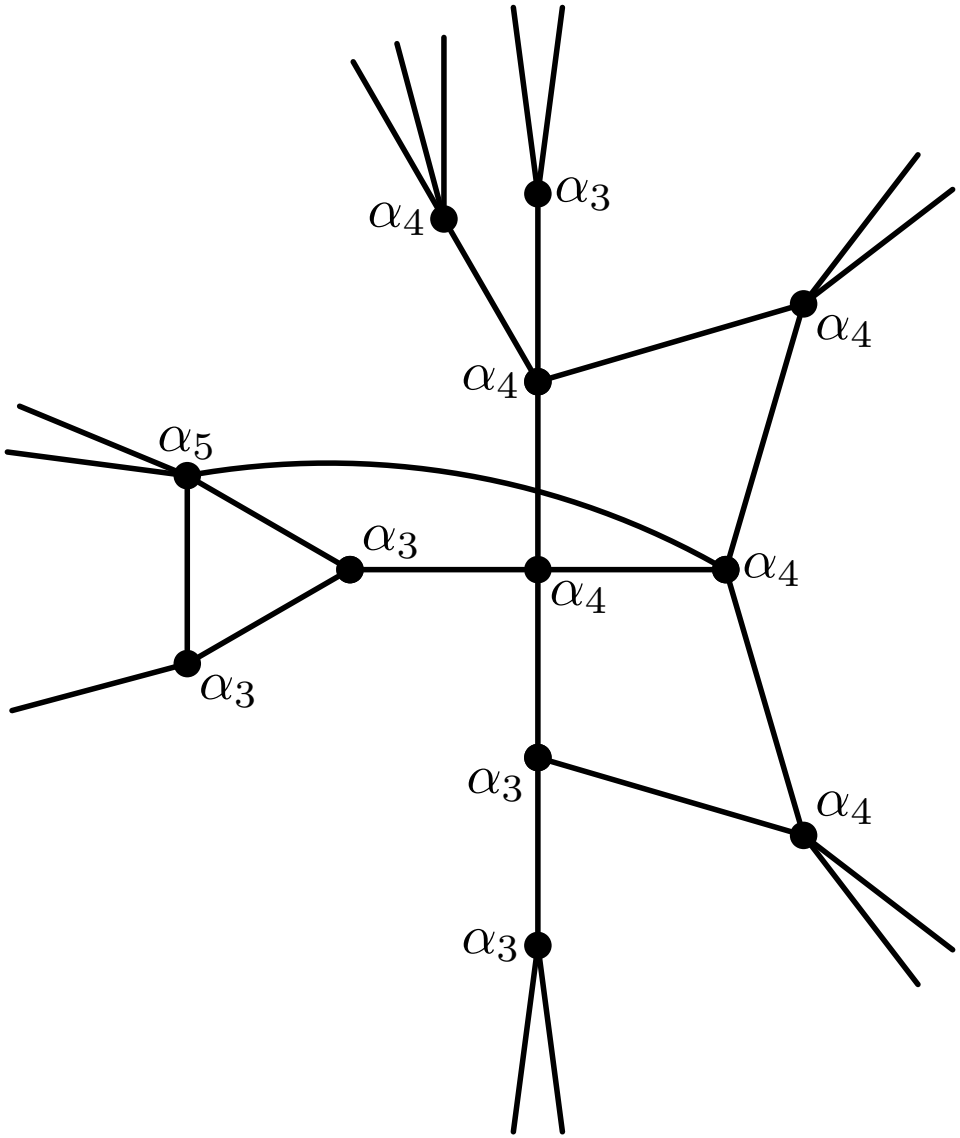
- $0 \leq \alpha_i \leq \alpha_{i+1} \leq 1$
- $\alpha_i + \alpha_j \geq \alpha_{i+j-2}$ (Folding!)

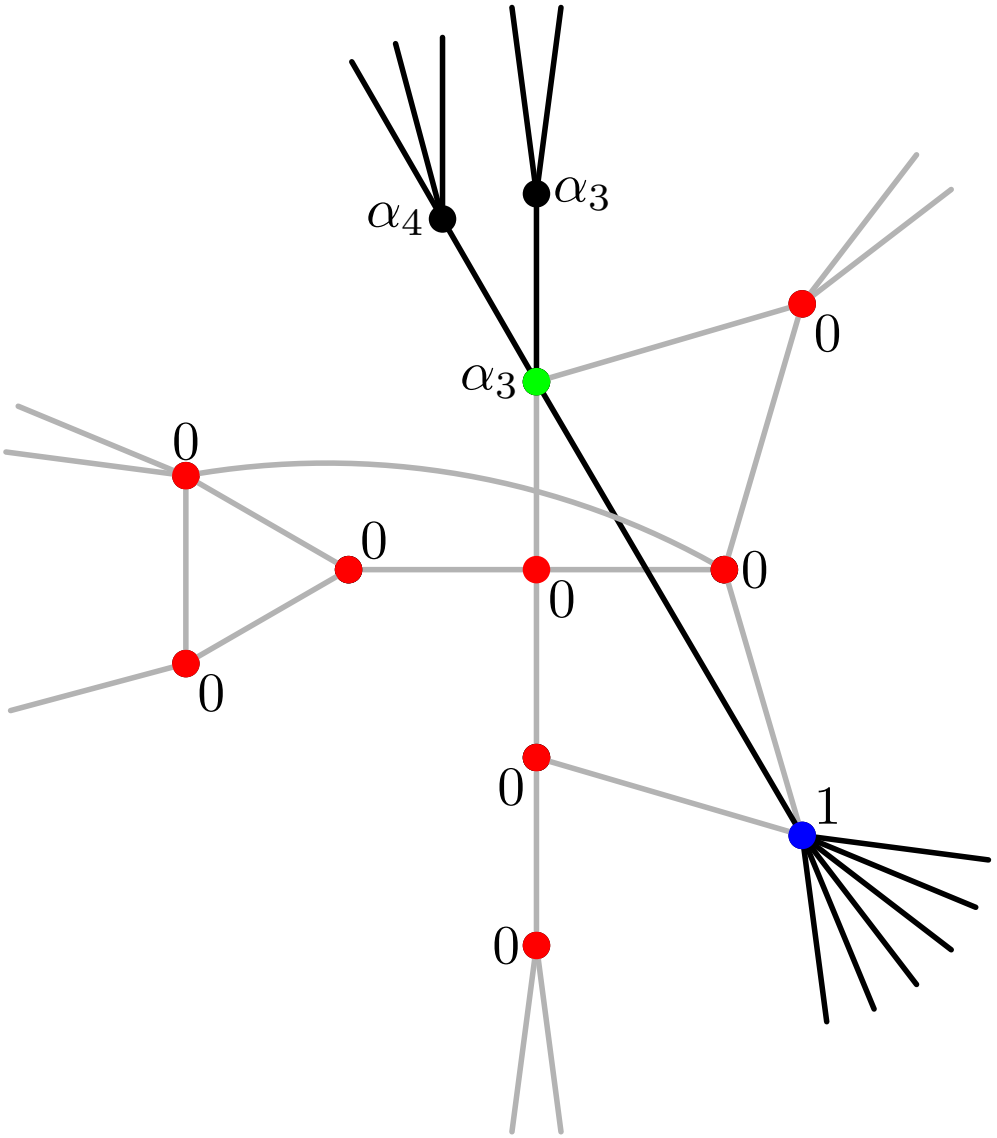
Example





Example


α_4
α_3
α_4
α_4
α_5
α_3
α_4
α_4
α_3
α_3
α_4
α_3


α_4
α_3
0
α_3
0
0
0
0
0
0
0
0
1
0

A Measure for Independent Set

Branching on u. Let $c_i := |\{v \in N(u) \mid \deg(v) = i\}|$. Then $\sum c_i = \deg(u)$.
First branch:

$$\alpha_{\deg(u)} + \sum_{i=3}^{\Delta(G)} c_i(\alpha_i - \alpha_{i-1})$$

Second branch:

$$\alpha_{\deg(u)} + \sum_{i=3}^{\Delta(G)} c_i\alpha_i + \deg(u)\min\{\alpha_3/3, \alpha_i - \alpha_{i-1} \mid 3 \leq i \leq \deg(u)\}$$

because of the domination reduction rule.

A Measure for Independent Set

If $\Delta(G) = 4$:

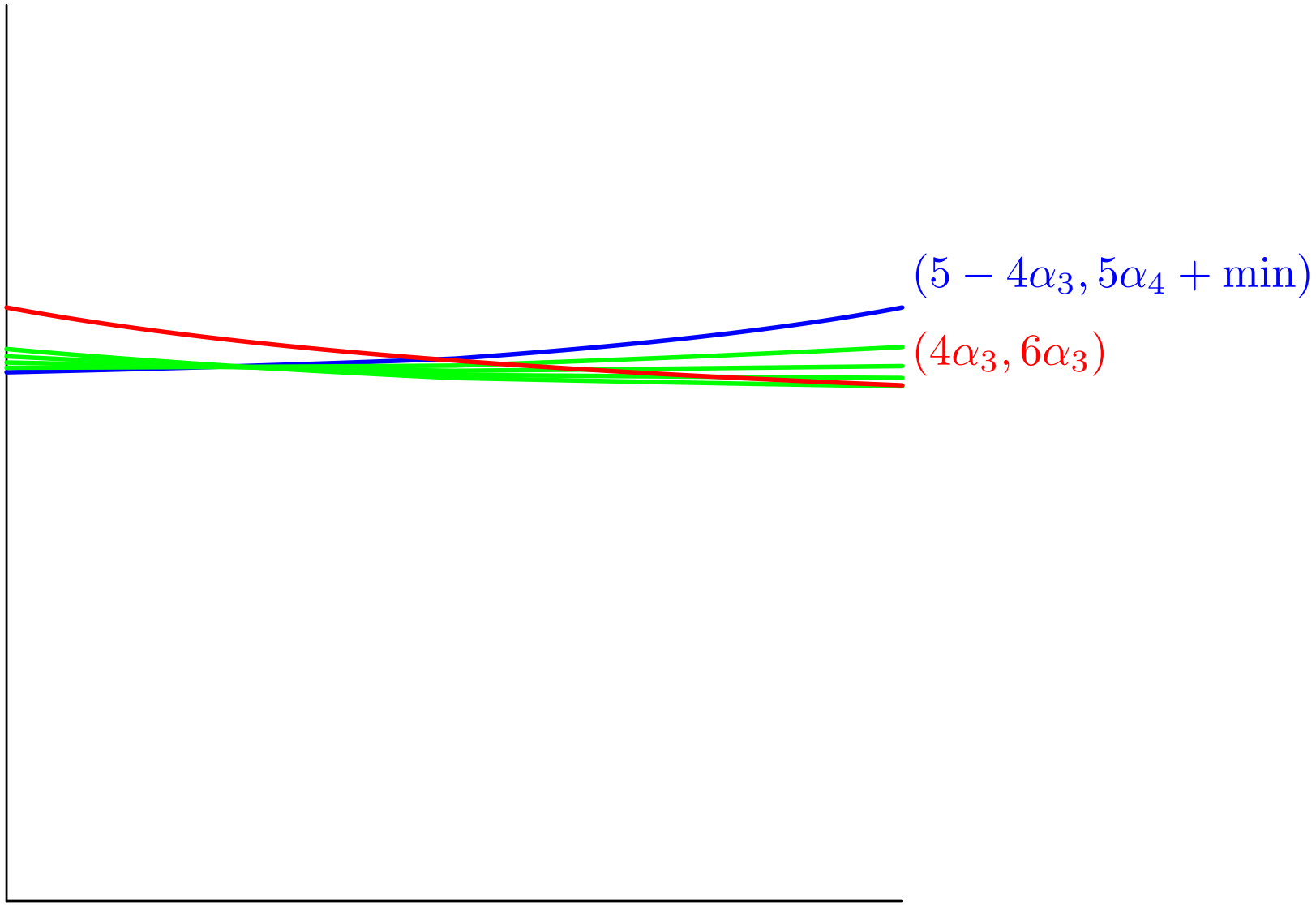
$$\begin{aligned}(\quad & \alpha_4 + c_4(\alpha_4 - \alpha_3) + c_3\alpha_3, \\ & \alpha_4 + c_4\alpha_4 + c_3\alpha_3 + 4\min\{\alpha_4 - \alpha_3, \alpha_3/3\})\end{aligned}$$

$$\begin{aligned}(5\alpha_4 - 4\alpha_3 \quad &, \quad 5\alpha_4 + 4\min\{\alpha_4 - \alpha_3, \alpha_3/3\}) \\ (4\alpha_4 - 2\alpha_3 \quad &, \quad 4\alpha_4 + 1\alpha_3 + 4\min\{\alpha_4 - \alpha_3, \alpha_3/3\}) \\ (3\alpha_4 \quad &, \quad 3\alpha_4 + 2\alpha_3 + 4\min\{\alpha_4 - \alpha_3, \alpha_3/3\}) \\ (2\alpha_4 + 2\alpha_3 \quad &, \quad 2\alpha_4 + 3\alpha_3 + 4\min\{\alpha_4 - \alpha_3, \alpha_3/3\}) \\ (1\alpha_4 + 4\alpha_3 \quad &, \quad 1\alpha_4 + 4\alpha_3 + 4\min\{\alpha_4 - \alpha_3, \alpha_3/3\})\end{aligned}$$

For 3-regular graphs, we know that $|N^2(u)| \geq 2$:

$$(\alpha_3 + 3(\alpha_3 - \alpha_2), \alpha_3 + 3\alpha_3 + 2\alpha_3) = (4\alpha_3, 6\alpha_3)$$

A Measure for Independent Set



Assuming $\alpha_i = 1$ for $i \geq 4$.

Optimum: $\alpha_3 = 0.74$, Bound for $\Delta(G) \leq 4$: 1.21

How to deal with higher degree nodes

Branching Vector:

$$\Big(\quad \alpha_{\deg(u)} + \sum_{i=3}^{\Delta(G)} c_i(\alpha_i - \alpha_{i-1}),$$

$$\alpha_{\deg(u)} + \sum_{i=3}^{\Delta(G)} c_i\alpha_i + \deg(u)\min\{\alpha_3/3, \alpha_i - \alpha_{i-1} \mid 3 \leq i \leq \deg(u)\}\Big)$$

Stop analysis as soon as desired bound is guaranteed.

Here:

$$\alpha_3 = 0.74,\ \alpha_4 = 1,\ \alpha_5 = 1,\ \ldots$$

How to deal with higher degree nodes

Branching Vector:

$$\Big(\quad \alpha_{\deg(u)} + \sum_{i=3}^{\Delta(G)} c_i(\alpha_i - \alpha_{i-1}),$$

$$\alpha_{\deg(u)} + \sum_{i=3}^{\Delta(G)} c_i\alpha_i + \deg(u)\min\{\alpha_3/3, \alpha_i - \alpha_{i-1} \mid 3 \leq i \leq \deg(u)\}\Big)$$

Stop analysis as soon as desired bound is guaranteed.

Here:

$$\alpha_3 = 0.74,\ \alpha_4 = 1,\ \alpha_5 = 1,\ \ldots$$

- Either nodes are completely removed (α_i) or
- One incident edge is removed ($\alpha_i - \alpha_{i-1}$)

Don't need to analyze degree 6: $\alpha_6 - \alpha_5 = \alpha_5 - \alpha_4 \neq \alpha_4 - \alpha_3$

Independent Set

$\Delta(G) = 5$:

- Worst Case: $\delta(G) = 5$
- Just compute all cases

Theorem

Independent Set *can be solved in time* $O^*(\tau(1,6)^{|V|})$.

Measure with $\alpha_4 < 1$?

$\Rightarrow$ Exercise!

(Just more cases and optimization in 2 dimensions)

Algorithm $\mathcal{SC}$

If $(\mathcal{U} = \emptyset)$ **return** 0;
Remove all $s \in \bigcup_{S \in \mathcal{S}} S \setminus \mathcal{U}$;
Apply reduction rules;
Choose a $S \in \mathcal{S}$ of maximum cardinality;
If $|S| = 2$ then solve instance in polynomial time;
else return $\min(\mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U}), \mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U} \setminus S) + 1)$;

Let the frequency of a $u \in \mathcal{U}$ be $f(u) := |\{S \in S \mid u \in S\}|$.
$n_i := |\{S \in \mathcal{S} \mid |S| = i\}|$, and $m_j = |\{u \in \mathcal{U} \mid f(u) = j\}|$

$$\Psi(\mathcal{S}) = \sum_{i \geq 1} \alpha_i n_i + \sum_{j \geq 1} \beta_i m_i \leq |\mathcal{S}| + |\mathcal{U}|$$

Useful:

- $\alpha_i \leq \alpha_{i+1}$, $\beta_j \leq \beta_{j+1}$
- $\alpha_1 = \beta_1 = 0$
- $\alpha_i = \beta_j = 1$, $i, j \geq 6$

Define:

$$\Delta(\alpha_i) = \alpha_i - \alpha_{i-1}$$

and

$$\Delta(\beta_i) = \beta_i - \beta_{i-1}$$

Assumption:

$$\Delta(\alpha_i) \geq \Delta(\alpha_{i-1})$$

$\mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U})$

Let $r_i = |\{u \in S \mid f(u) = i\}|$. We gain at least

$$\alpha_{|S|} + \sum_{i \geq 2} r_i \Delta(\beta_i) = \alpha_{|S|} + \sum_{i=2}^{6} r_i \Delta(\beta_i)$$

If $u \in S$ with $f(u) = 2$, we have to take the other set containing u

$\mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U})$

Let $r_i = |\{u \in S \mid f(u) = i\}|$. We gain at least

$$\alpha_{|S|} + \sum_{i \geq 2} r_i \Delta(\beta_i) = \alpha_{|S|} + \sum_{i=2}^{6} r_i \Delta(\beta_i)$$

If $u \in S$ with $f(u) = 2$, we have to take the other set containing u

$$k = \begin{cases} 0 & r_2 = 0 \\ \beta_2 + \alpha_2 & r_2 = 1 \end{cases}$$

$\mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U})$

Let $r_i = |\{u \in S \mid f(u) = i\}|$. We gain at least

$$\alpha_{|S|} + \sum_{i \geq 2} r_i \Delta(\beta_i) = \alpha_{|S|} + \sum_{i=2}^{6} r_i \Delta(\beta_i)$$

If $u \in S$ with $f(u) = 2$, we have to take the other set containing u

$$k' = \begin{cases} 0 & r_2 = 0 \\ \beta_2 + \alpha_2 & r_2 = 1 \\ \beta_2 + \min\{2\alpha_2, \alpha_3\} & r_2 = 2 \end{cases}$$

$\mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U})$

Let $r_i = |\{u \in S \mid f(u) = i\}|$. We gain at least

$$\alpha_{|S|} + \sum_{i \geq 2} r_i \Delta(\beta_i) = \alpha_{|S|} + \sum_{i=2}^{6} r_i \Delta(\beta_i)$$

If $u \in S$ with $f(u) = 2$, we have to take the other set containing u

$$k' = \begin{cases} 0 & r_2 = 0 \\ \beta_2 + \alpha_2 & r_2 = 1 \\ \beta_2 + \min\{2\alpha_2, \alpha_3\} & r_2 = 2 \\ \beta_2 + \min\{3\alpha_2, \alpha_2 + \alpha_3\} & r_2 = 3, |S| = 3 \\ \beta_2 + \min\{3\alpha_2, \alpha_2 + \alpha_3, \alpha_4\} & r_2 = 3, |S| \geq 4 \end{cases}$$

Algorithm $\mathcal{SC}$

$\mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U} \setminus S)$

$$\alpha_{|S|} + \sum_{i \geq 2} r_i \beta_i = \alpha_{|S|} + \sum_{i=2}^{6} r_i \beta_i + r_7$$

But the size of all sets S' sharing elements with S decreases

Remember: $\Delta(\alpha_{|S|}) \leq \Delta(\alpha_{|S'|})$

Algorithm $\mathcal{SC}$

$\mathcal{SC}(\mathcal{S} \setminus \{S\}, \mathcal{U} \setminus S)$

$$\alpha_{|S|} + \sum_{i \geq 2} r_i \beta_i = \alpha_{|S|} + \sum_{i=2}^{6} r_i \beta_i + r_7$$

But the size of all sets S' sharing elements with S decreases

Remember: $\Delta(\alpha_{|S|}) \leq \Delta(\alpha_{|S'|})$

$$l = \Delta(\alpha_{|S|}) \sum_{i \geq 2} (i-1) r_i$$

Runtime Bound

Overall:

$$(\alpha_{|S|} + \sum_{i=2}^{6} r_i \Delta(\beta_i) + k, \alpha_{|S|} + \sum_{i=2}^{6} r_i \beta_i + r_7 + l)$$

forall $|S|$, all possible $\sum r_i = |S|$, all possible k and all possible l.

Use computers to optimize values

i	α_i	β_i
2	0.377	0.399
3	0.754	0.767
4	0.909	0.929
5	0.976	0.985

Runtime Bound

Overall:

$$(\alpha_{|S|} + \sum_{i=2}^{6} r_i \Delta(\beta_i) + k, \alpha_{|S|} + \sum_{i=2}^{6} r_i \beta_i + r_7 + l)$$

forall $|S|$, all possible $\sum r_i = |S|$, all possible k and all possible l.

Use computers to optimize values

Theorem

Set Cover *can be solved in time* $O^*(1.24^{|\mathcal{U}+\mathcal{S}|})$.

Theorem

Dominating Set *can be solved in* $O^*(1.24^{2n}) = O^*(1.53^n)$

Overview

1. Introduction
2. Branching
3. Measure&Conquer
4. Dynamic Programming
5. Local Search
6. Split & List
7. Lower Bounds
8. Inclusion-exclusion
9. Subset Convolution

Overview

1. Dynamic Programming
2. Inclusion-exclusion
3. Subset Convolution

Dynamic Programming

Basic Idea

- Solve small subproblems; compose solutions to obtain final solution.
- Branching Algorithms: essentially top-down approach;
  Dynamic Programming: bottom-up approach.
- Used to construct polynomial-time and exponential-time algorithms.

Dynamic Programming

Basic Idea

- Solve small subproblems; compose solutions to obtain final solution.
- Branching Algorithms: essentially top-down approach;
  Dynamic Programming: bottom-up approach.
- Used to construct polynomial-time and exponential-time algorithms.

- Permutation Problems.
- Partition Problems.
- Covering Problems.

Permutation Problems

The Travelling Salesperson Problem (TSP)

- **Input:** A set of n cities $c_1, \ldots, c_n$ and the distance $d(c_i, c_j)$ between every pair of cities.
- **Goal:** Construct a tour that starts at city c_1, visits all the remaining cities and returns to c_1, and is of minimum total length.

- Classic NP-complete problem.
- Several versions have been studied since the 1960s.

Permutation Problems

The Travelling Salesperson Problem (TSP)

- **Input:** A set of n cities $c_1, \ldots, c_n$ and the distance $d(c_i, c_j)$ between every pair of cities.
- **Goal:** Construct a tour that starts at city c_1, visits all the remaining cities and returns to c_1, and is of minimum total length.

- Classic NP-complete problem.
- Several versions have been studied since the 1960s.

Naive Solution

Generate all permutations of $\{c_1, \ldots, c_n\}$ and choose a tour of minimum total length.

Permutation Problems

The Travelling Salesperson Problem (TSP)

- **Input:** A set of n cities $c_1, \ldots, c_n$ and the distance $d(c_i, c_j)$ between every pair of cities.
- **Goal:** Construct a tour that starts at city c_1, visits all the remaining cities and returns to c_1, and is of minimum total length.

- Classic NP-complete problem.
- Several versions have been studied since the 1960s.

Naive Solution

Generate all permutations of $\{c_1, \ldots, c_n\}$ and choose a tour of minimum total length.

Time Complexity $= \Omega(n!)$.

TSP by Dynamic Programming (Held and Karp 1962)

For $S \subseteq \{c_2, \ldots, c_n\}$ and $c_i \in S$, define

$$\text{OPT}[S, c_i] := \text{length of optimal tour that starts at } c_1,$$
$$\text{visits all cities of } S, \text{ in } \textit{some} \text{ order, and ends at } c_i.$$

TSP by Dynamic Programming (Held and Karp 1962)

For $S \subseteq \{c_2, \ldots, c_n\}$ and $c_i \in S$, define

$$\text{OPT}[S, c_i] := \text{length of optimal tour that starts at } c_1, \text{ visits all cities of } S, \text{ in } \textit{some} \text{ order, and ends at } c_i.$$

- If $S = \{c_i\}$,

$$\text{OPT}[S, c_i] = d(c_1, c_i).$$

TSP by Dynamic Programming (Held and Karp 1962)

For $S \subseteq \{c_2, \ldots, c_n\}$ and $c_i \in S$, define

$$\text{OPT}[S, c_i] := \text{length of optimal tour that starts at } c_1, \text{ visits all cities of } S, \text{ in } \textit{some} \text{ order, and ends at } c_i.$$

- If $S = \{c_i\}$,
$$\text{OPT}[S, c_i] = d(c_1, c_i).$$
- If $|S| \geq 2$,
$$\text{OPT}[S, c_i] = \min_{c_j \in S \setminus c_i} \{\text{OPT}[S \setminus c_i, c_j] + d(c_j, c_i)\}.$$

TSP by Dynamic Programming (Held and Karp 1962)

For $S \subseteq \{c_2, \ldots, c_n\}$ and $c_i \in S$, define

$$\text{OPT}[S, c_i] := \text{length of optimal tour that starts at } c_1, \text{ visits all cities of } S, \text{ in } \textit{some} \text{ order, and ends at } c_i.$$

- If $S = \{c_i\}$,
$$\text{OPT}[S, c_i] = d(c_1, c_i).$$
- If $|S| \geq 2$,
$$\text{OPT}[S, c_i] = \min_{c_j \in S \setminus c_i} \{\text{OPT}[S \setminus c_i, c_j] + d(c_j, c_i)\}.$$

Optimal Solution Size

$$\text{OPT} = \min_{c_2 \leq c_j \leq c_n} \{\text{OPT}[\{c_2, \ldots, c_n\}, c_j] + d(c_j, c_1)\}.$$

Proof of Correctness

Claim

$$\mathsf{OPT}[S, c_i] = \min_{c_j \in S \setminus c_i} \{\mathsf{OPT}[S \setminus c_i, c_j] + d(c_j, c_i)\}.$$

Proof of Correctness

Claim

$$\mathsf{OPT}[S, c_i] = \min_{c_j \in S \setminus c_i} \{\mathsf{OPT}[S \setminus c_i, c_j] + d(c_j, c_i)\}.$$

Proof

$$\mathsf{OPT}[S, c_i] \leq \min_{c_j \in S \setminus c_i} \{\mathsf{OPT}[S \setminus c_i, c_j] + d(c_j, c_i)\}.$$

Any path that starts at c_1, visits all vertices in $S \setminus c_i$, and finally ends at c_i must be at least as expensive as $\mathsf{OPT}[S, c_i]$.

Proof of Correctness

Claim

$$\mathsf{OPT}[S, c_i] = \min_{c_j \in S \setminus c_i} \{\mathsf{OPT}[S \setminus c_i, c_j] + d(c_j, c_i)\}.$$

Proof

$$\mathsf{OPT}[S, c_i] \geq \min_{c_j \in S \setminus c_i} \{\mathsf{OPT}[S \setminus c_i, c_j] + d(c_j, c_i)\}.$$

Consider an optimal path that starts at c_1, visits all vertices of S in some order, and terminates at c_i. If c_j is the second-last vertex on this path then $\mathsf{OPT}[S, c_i] = \mathsf{OPT}[S \setminus c_i, c_j] + d(c_j, c_i)$, and hence the inequality follows.

TSP by Dynamic Programming

Algorithm TSP

1. **for** $i = 2$ to n **do**
   $\mathrm{OPT}[\{c_i\}, c_i] := d(c_1, c_i)$;
2. **for** $k = 2$ to $n - 1$ **do**
2.1. **forall** $S \subseteq \{2, \ldots, n\}$ with $|S| = k$ **do**
2.2. $\mathrm{OPT}[S, c_i] := \min_{c_j \in S \setminus c_i} \{\mathrm{OPT}[S \setminus c_i, c_j] + d(c_j, c_i)\}$;
3. **return** $\min_i \{\mathrm{OPT}[S \setminus c_1, c_i] + d(c_i, c_1)\}$;

TSP by Dynamic Programming

Algorithm TSP

1. **for** $i = 2$ to n **do**
   $\text{OPT}[\{c_i\}, c_i] := d(c_1, c_i)$;
2. **for** $k = 2$ to $n - 1$ **do**
2.1. **forall** $S \subseteq \{2, \ldots, n\}$ with $|S| = k$ **do**
2.2. $\text{OPT}[S, c_i] := \min_{c_j \in S \setminus c_i} \{\text{OPT}[S \setminus c_i, c_j] + d(c_j, c_i)\}$;
3. **return** $\min_i \{\text{OPT}[S \setminus c_1, c_i] + d(c_i, c_1)\}$;

Time Complexity

1. Step 1: $O(n)$.

TSP by Dynamic Programming

Algorithm TSP

1. **for** $i = 2$ to n **do**
   $\text{OPT}[\{c_i\}, c_i] := d(c_1, c_i)$;
2. **for** $k = 2$ to $n - 1$ **do**
2.1. **forall** $S \subseteq \{2, \ldots, n\}$ with $|S| = k$ **do**
2.2. $\text{OPT}[S, c_i] := \min_{c_j \in S \setminus c_i} \{\text{OPT}[S \setminus c_i, c_j] + d(c_j, c_i)\}$;
3. **return** $\min_i \{\text{OPT}[S \setminus c_1, c_i] + d(c_i, c_1)\}$;

Time Complexity

1. Step 1: $O(n)$.
2. Step 2.2: Since $|S| = k$, time taken for computing $\text{OPT}[S, c_i]$ for all $c_i \in S$ is $O(k^2)$.

TSP by Dynamic Programming

Algorithm TSP

1. **for** $i = 2$ to n **do**
   $\text{OPT}[\{c_i\}, c_i] := d(c_1, c_i)$;
2. **for** $k = 2$ to $n - 1$ **do**
2.1. **forall** $S \subseteq \{2, \ldots, n\}$ with $|S| = k$ **do**
2.2. $\text{OPT}[S, c_i] := \min_{c_j \in S \setminus c_i} \{\text{OPT}[S \setminus c_i, c_j] + d(c_j, c_i)\}$;
3. **return** $\min_i \{\text{OPT}[S \setminus c_1, c_i] + d(c_i, c_1)\}$;

Time Complexity

1. Step 1: $O(n)$.
2. Step 2.2: Since $|S| = k$, time taken for computing $\text{OPT}[S, c_i]$ for all $c_i \in S$ is $O(k^2)$.
3. Step 2: $O(\sum_{k=2}^{n} \binom{n}{k} \cdot k^2) = O(2^n \cdot n^2)$.

TSP by Dynamic Programming

Algorithm TSP

1. **for** $i = 2$ to n **do**
   OPT$[\{c_i\}, c_i] := d(c_1, c_i)$;
2. **for** $k = 2$ to $n - 1$ **do**
2.1. **forall** $S \subseteq \{2, \ldots, n\}$ with $|S| = k$ **do**
2.2. OPT$[S, c_i] := \min_{c_j \in S \setminus c_i}\{\text{OPT}[S \setminus c_i, c_j] + d(c_j, c_i)\}$;
3. **return** $\min_i\{\text{OPT}[S \setminus c_1, c_i] + d(c_i, c_1)\}$;

Space Complexity

Exponential Space: $O(2^n)$.

Main drawback of DP algorithms.

TSP: Constructing an Optimal Solution

Optimal Solution Size

$$\mathsf{OPT} = \min_{c_2 \leq c_j \leq c_n} \{\mathsf{OPT}[\{c_2, \ldots, c_n\}, c_j] + d(c_j, c_1)\}.$$

TSP: Constructing an Optimal Solution

Optimal Solution Size

$$\text{OPT} = \min_{c_2 \leq c_j \leq c_n} \{\text{OPT}[\{c_2, \ldots, c_n\}, c_j] + d(c_j, c_1)\}.$$

1. Last city on an optimal tour: city c_j for which $\text{OPT}[\{c_2, \ldots, c_n\}, c_j] + d(c_j, c_1)$ is a minimum.
2. Second-last city on an optimal tour: city $c_{j'}$ for which $\text{OPT}[\{c_2, \ldots, c_n\} \setminus c_j, c'_j] + d(c'_j, c_j)$ is a minimum.
3. And so on ...

TSP: Constructing an Optimal Solution

Optimal Solution Size

$$\text{OPT} = \min_{c_2 \leq c_j \leq c_n} \{\text{OPT}[\{c_2, \ldots, c_n\}, c_j] + d(c_j, c_1)\}.$$

1. Last city on an optimal tour: city c_j for which $\text{OPT}[\{c_2, \ldots, c_n\}, c_j] + d(c_j, c_1)$ is a minimum.
2. Second-last city on an optimal tour: city $c_{j'}$ for which $\text{OPT}[\{c_2, \ldots, c_n\} \setminus c_j, c'_j] + d(c'_j, c_j)$ is a minimum.
3. And so on ...

A DP algorithm computing the optimal value can typically produce the optimal solution in the same time.

Some Remarks

1. The DP algorithm for TSP by Held and Karp was the *best* algorithm for general TSP for nearly fifty years!
2. A randomized algorithm for TSP with time complexity $O^*(1.657^n)$ was developed recently [Björklund, August 2010].

Permutation Problems

Job Scheduling

- **Input:** A set of n jobs $J_1, \ldots, J_n$ with execution times $\tau_1, \ldots, \tau_n$ and costs $c_1(t), \ldots, c_n(t)$.
- **Goal:** Construct a schedule of minimum cost.

Permutation Problems

Job Scheduling

- **Input:** A set of n jobs $J_1, \ldots, J_n$ with execution times $\tau_1, \ldots, \tau_n$ and costs $c_1(t), \ldots, c_n(t)$.
- **Goal:** Construct a schedule of minimum cost.

- Job J_i takes time τ_i to complete.
- $c_i(t) =$ the cost of completing job J_i at time t.
- Jobs are executed on a single machine without preemption.

Job Scheduling

- **Input:** A set of n jobs $J_1, \ldots, J_n$ with execution times $\tau_1, \ldots, \tau_n$ and costs $c_1(t), \ldots, c_n(t)$.
- **Goal:** Construct a schedule of minimum cost.

- If jobs are executed in the *schedule (order)* $\pi(1), \ldots, \pi(n)$, the **termination time** of $J_{\pi(i)}$ is

$$t_{\pi(i)} = \sum_{j=1}^{i} \tau_{\pi(j)}.$$

- **Total cost** associated with schedule π is

$$\sum_{i=1}^{n} c_{\pi(i)}(t_{\pi(i)}).$$

Job Scheduling: A DP Algorithm

Job Scheduling

- **Input:** A set of n jobs $J_1, \ldots, J_n$ with execution times $\tau_1, \ldots, \tau_n$ and costs $c_1(t), \ldots, c_n(t)$.
- **Goal:** Construct a schedule of minimum cost.

Job Scheduling: A DP Algorithm

Job Scheduling

- **Input:** A set of n jobs $J_1, \ldots, J_n$ with execution times $\tau_1, \ldots, \tau_n$ and costs $c_1(t), \ldots, c_n(t)$.
- **Goal:** Construct a schedule of minimum cost.

Naive Solution

Generate all permutations of $\{1, \ldots, n\}$ are find the schedule that minimizes the cost.

Job Scheduling: A DP Algorithm

Job Scheduling

- **Input:** A set of n jobs $J_1, \ldots, J_n$ with execution times $\tau_1, \ldots, \tau_n$ and costs $c_1(t), \ldots, c_n(t)$.
- **Goal:** Construct a schedule of minimum cost.

Naive Solution

Generate all permutations of $\{1, \ldots, n\}$ are find the schedule that minimizes the cost.

Time complexity $= \Omega(n!)$.

Job Scheduling: A DP Algorithm

Job Scheduling

- **Input:** A set of n jobs $J_1, \ldots, J_n$ with execution times $\tau_1, \ldots, \tau_n$ and costs $c_1(t), \ldots, c_n(t)$.
- **Goal:** Construct a schedule of minimum cost.

An Idea for a DP Algorithm

Find out the optimal solution to *small* subsets of jobs; *combine* solutions to small subsets to obtain solutions for larger subsets.

Job Scheduling: A DP Algorithm

Job Scheduling

- **Input:** A set of n jobs $J_1, \ldots, J_n$ with execution times $\tau_1, \ldots, \tau_n$ and costs $c_1(t), \ldots, c_n(t)$.
- **Goal:** Construct a schedule of minimum cost.

Job Scheduling: A DP Algorithm

Job Scheduling

- **Input:** A set of n jobs $J_1, \ldots, J_n$ with execution times $\tau_1, \ldots, \tau_n$ and costs $c_1(t), \ldots, c_n(t)$.
- **Goal:** Construct a schedule of minimum cost.

- For $S \subseteq \{J_1, \ldots, J_n\}$, define

$$t_S := \sum_{i \in S} \tau_i.$$

- OPT$[S]$:= minimum cost of executing all jobs in S, assuming that we start at time $t = 0$.

Job Scheduling: A DP Algorithm

Job Scheduling

- **Input:** A set of n jobs $J_1, \ldots, J_n$ with execution times $\tau_1, \ldots, \tau_n$ and costs $c_1(t), \ldots, c_n(t)$.
- **Goal:** Construct a schedule of minimum cost.

Job Scheduling: A DP Algorithm

Job Scheduling

- **Input:** A set of n jobs $J_1, \ldots, J_n$ with execution times $\tau_1, \ldots, \tau_n$ and costs $c_1(t), \ldots, c_n(t)$.
- **Goal:** Construct a schedule of minimum cost.

Constructing OPT$[S]$ bottom up

- If $S = \{J_i\}$ then $\mathsf{OPT}[S] = c_i(\tau_i)$.
- If $|S| > 1$,

$$\mathsf{OPT}[S] = \min\{\mathsf{OPT}[S \setminus J_i] + c_i(t_S) : J_i \in S\}.$$

Job Scheduling: A DP Algorithm

Job Scheduling

- **Input:** A set of n jobs $J_1, \ldots, J_n$ with execution times $\tau_1, \ldots, \tau_n$ and costs $c_1(t), \ldots, c_n(t)$.
- **Goal:** Construct a schedule of minimum cost.

Constructing OPT$[S]$ bottom up

- If $S = \{J_i\}$ then $\mathsf{OPT}[S] = c_i(\tau_i)$.
- If $|S| > 1$,

$$\mathsf{OPT}[S] = \min\{\mathsf{OPT}[S \setminus J_i] + c_i(t_S) : J_i \in S\}.$$

Time Complexity

Time $= O(n) + O(\sum_{j=2}^{n} \binom{n}{j} \cdot j) = O^*(2^n \cdot n)$.

Cutwidth

Given a graph $G = (V, E)$ and an ordering π of the vertex set V, the **cutwidth of** π is

$$\max_{u \in V} |\{x, y\} \in E \colon \pi(x) \leq \pi(u) < \pi(y)|.$$

The **cutwidth of** $u \in V(G)$ is $|\{x, y\} \colon \pi(x) \leq \pi(u) < \pi(y)|$. The **cutwidth of** G is the minimum cutwidth over all possible orderings.

Cutwidth

Given a graph $G = (V, E)$ and an ordering π of the vertex set V, the **cutwidth of** π is

$$\max_{u \in V} |\{x, y\} \in E \colon \pi(x) \leq \pi(u) < \pi(y)|.$$

The **cutwidth of** $u \in V(G)$ is $|\{x, y\} \colon \pi(x) \leq \pi(u) < \pi(y)|$. The **cutwidth of** G is the minimum cutwidth over all possible orderings.

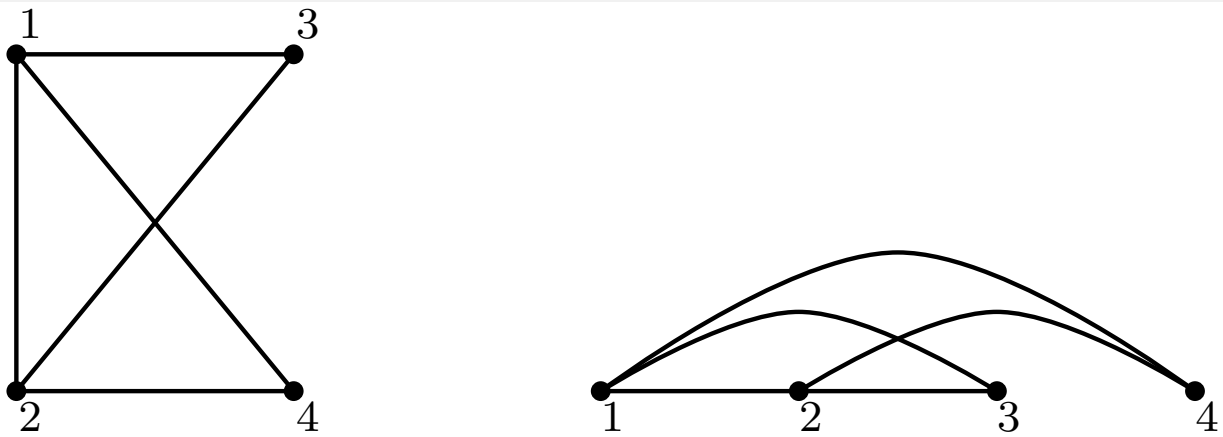


Figure: Cutwidth of ordering = 4.

DP Algorithm for Cutwidth: A First Approach

Let's follow the approach as before.

- Given a graph $G = (V, E)$, for $S \subseteq V(G)$ define

$$\mathsf{OPT}[S] := \mathsf{cutwidth}(G[S]).$$

- For $|S| = 1$, $\mathsf{OPT}[S] = 0$.
- For $|S| \geq 2$, guessing the position of the last few vertices in an optimal ordering for $G[S]$ is not sufficient.
  - Not clear how to compute $\mathsf{OPT}[S]$ from $\mathsf{OPT}[S \setminus c_j]$, where $c_j \in S$.

DP Algorithm for Cutwidth

Defining an Appropriate Subproblem

Given a graph $G = (V, E)$ and $S \subseteq V$,

- let $\Pi(S, \bar{S})$ be the set of all permutations of V that can be expressed as a permutation of S "followed by" a permutation of $\bar{S}$.
- For $S \subseteq V$, define OPT$[S]$ to be the maximum cutwidth of a vertex of S in an optimal permutation in $\Pi(S, \bar{S})$.
- Then OPT$[V]$ = cutwidth(G).

DP Algorithm for Cutwidth

Defining an Appropriate Subproblem

Given a graph $G = (V, E)$ and $S \subseteq V$,

- let $\Pi(S, \bar{S})$ be the set of all permutations of V that can be expressed as a permutation of S "followed by" a permutation of $\bar{S}$.
- For $S \subseteq V$, define OPT$[S]$ to be the maximum cutwidth of a vertex of S in an optimal permutation in $\Pi(S, \bar{S})$.
- Then OPT$[V]$ = cutwidth(G).

Computing OPT$[S]$

- For $S = \emptyset$, OPT$[S] = 0$.
- For $S = \{u\}$, OPT$[S] = d(u)$.
- For $|S| \geq 1$,

$$\text{OPT}[S] = \min_{v \in S} \max\{\text{OPT}[S \setminus v], |\text{edges}(S, \bar{S})|\}.$$

Proof of Correctness

Lemma

For $|S| \geq 1$,

$$\mathsf{OPT}[S] = \min_{v \in S} \max\{\mathsf{OPT}[S \setminus v], |\mathsf{edges}(S, \bar{S})|\}.$$

Proof

Exercise!

DP Algorithm for Cutwidth

Algorithm computing OPT$[S]$

1. OPT$[\emptyset] := 0$;
2. **for** $i = 1$ to n and **forall** $S \subseteq V$ with $|S| = i$ **do**
   1. OPT$[S] = \min_{v \in S} \max\{\text{OPT}[S \setminus v], |\text{edges}(S, \bar{S})|\}$;

Time Complexity

$$\text{Running time} = \sum_{i=1}^{n} \binom{n}{i} \cdot (i + n + m) = O(2^n \cdot (n + m)).$$

Optimal Linear Arrangement

Given a graph $G = (V, E)$ find a permutation $\pi \colon V \to \{1, \ldots, |V|\}$ of the vertices that minimizes the sum

$$\sum_{\{u,v\} \in E} |\pi(u) - \pi(v)|.$$

The quantity $|\pi(u) - \pi(v)|$ is the **stretch** of $\{u, v\}$. The optimal value is denoted **OLA**(G).

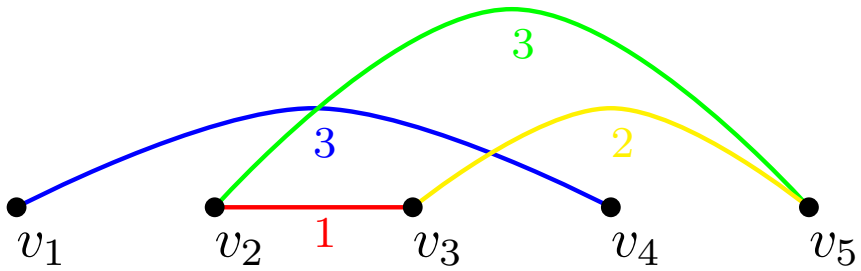


Figure: Linear Arrangement with cost 9.

Optimal Linear Arrangements and Cutwidth

Lemma

Given a graph $G = (V, E)$ *and a permutation* $\pi \colon V \to \{1, \ldots, |V|\}$ *of* V

$$\sum_{\{u,v\} \in E} |\pi(u) - \pi(v)| = \sum_{v \in V} |\{\{x, y\} \in E \colon \pi(x) \leq \pi(v) \leq \pi(y)\}|.$$

In words: *sum of edge stretches = sum of the vertex cutwidths*.

Optimal Linear Arrangements and Cutwidth

Lemma

Given a graph $G = (V, E)$ and a permutation $\pi\colon V \to \{1, \ldots, |V|\}$ of V

$$\sum_{\{u,v\}\in E} |\pi(u) - \pi(v)| = \sum_{v\in V} |\{\{x, y\} \in E\colon \pi(x) \leq \pi(v) \leq \pi(y)\}|.$$

In words: sum of edge stretches = sum of the vertex cutwidths.

Proof

If the edge $\{u, v\}$ has stretch $|\pi(u) - \pi(v)|$ in the ordering π, then this edge contributes a *one* to the cutwidth of $|\pi(u) - \pi(v)|$ vertices of V. Therefore the sum of the vertex cutwidths equals the sum of the edge stretches.

OLA: Alternative Formulation

Given a graph $G = (V, E)$, find a permutation $\pi \colon V \to \{1, \ldots, |V|\}$ of V that minimizes the sum of the vertex cutwidths

$$\min_{\pi} \sum_{v \in V} |\{\{x, y\} \in E \colon \pi(x) \leq \pi(v) \leq \pi(y)\}|.$$

DP Algorithm for OLA

Defining the Subproblem

Given a graph $G = (V, E)$ and $S \subseteq V$,

- let $\Pi(S, \bar{S})$ be the set of all permutations of V that can be expressed as a permutation of S "followed by" a permutation of $\bar{S}$.
- For $S \subseteq V$, define $\mathrm{OPT}[S]$ to be the sum of the cutwidths of the vertices in S in an optimal permutation in $\Pi(S, \bar{S})$.
- Then $\mathrm{OPT}[V] = \mathrm{OLA}(G)$.

DP Algorithm for OLA

Defining the Subproblem

Given a graph $G = (V, E)$ and $S \subseteq V$,

- let $\Pi(S, \bar{S})$ be the set of all permutations of V that can be expressed as a permutation of S "followed by" a permutation of $\bar{S}$.
- For $S \subseteq V$, define OPT$[S]$ to be the sum of the cutwidths of the vertices in S in an optimal permutation in $\Pi(S, \bar{S})$.
- Then OPT$[V]$ = OLA(G).

Computing OPT$[S]$

- For $S = \emptyset$, OPT$[S] = 0$.
- For $S = \{u\}$, OPT$[S] = d(u)$.
- For $|S| \geq 1$,

$$\text{OPT}[S] = \min_{v \in S}\{\text{OPT}[S \setminus v] + |\text{edges}(S, \bar{S})|\}.$$

DP Algorithm for OLA

Defining the Subproblem

Given a graph $G = (V, E)$ and $S \subseteq V$,

- let $\Pi(S, \bar{S})$ be the set of all permutations of V that can be expressed as a permutation of S "followed by" a permutation of $\bar{S}$.
- For $S \subseteq V$, define $\mathrm{OPT}[S]$ to be the sum of the cutwidths of the vertices in S in an optimal permutation in $\Pi(S, \bar{S})$.
- Then $\mathrm{OPT}[V] = \mathrm{OLA}(G)$.

Details: Exercise!

Directed Feedback Arc Set

- A directed graph $D = (V, A)$.
- Find a smallest arc set $F \subseteq A$ such that $D \setminus F$ has no directed cycles.

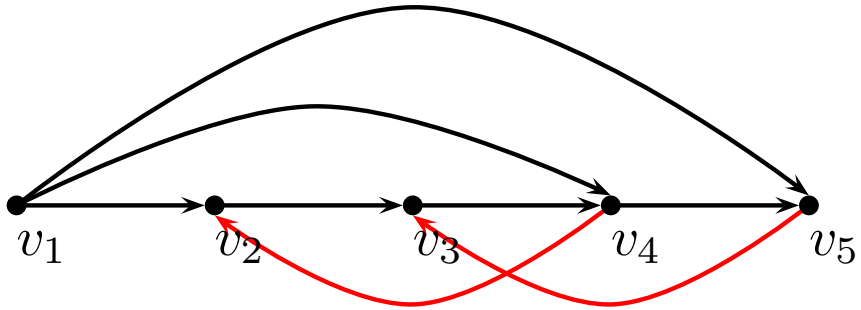


Figure: A digraph with a feedback arc set of size two.

Directed Feedback Arc Set

- A directed graph $D = (V, A)$.
- Find a smallest arc set $F \subseteq A$ such that $D \setminus F$ has no directed cycles.

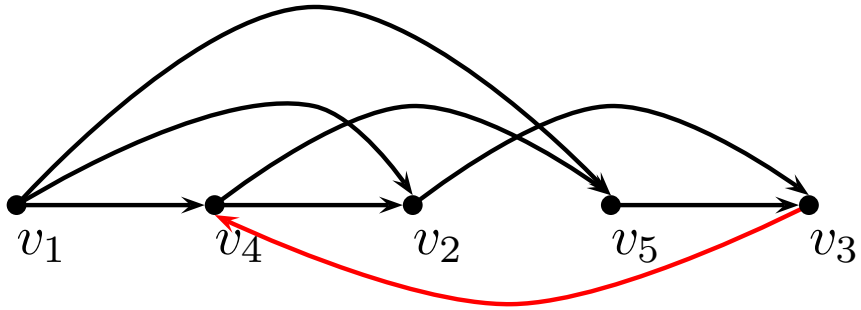


Figure: Same digraph with a feedback arc set of size one.

DFAS as a Permutation Problem

Definition (Topological Orderings)

A **topological ordering** of a directed graph is a permutation of the vertex set such that all arcs are directed from left to right in the ordering.

DFAS as a Permutation Problem

Definition (Topological Orderings)

A **topological ordering** of a directed graph is a permutation of the vertex set such that all arcs are directed from left to right in the ordering.

Lemma

A digraph has a topological ordering if and only if it is acyclic.

DFAS as a Permutation Problem

Definition (Topological Orderings)

A **topological ordering** of a directed graph is a permutation of the vertex set such that all arcs are directed from left to right in the ordering.

Lemma

A digraph has a topological ordering if and only if it is acyclic.

The DFAS Problem: Restatement

Find an ordering of the vertex set such that the number of arcs from right to left is minimized.

DFAS: A DP Algorithm

- Choosing all possible arc sets takes time $\Omega(2^m)$ and $m = O(n^2)$.
- Aim: obtain an algorithm with running time $O(2^n)$.

DFAS: A DP Algorithm

- Choosing all possible arc sets takes time $\Omega(2^m)$ and $m = O(n^2)$.
- Aim: obtain an algorithm with running time $O(2^n)$.

- Given a digraph $D = (V, A)$, for $S \subseteq V$ define

$$\mathsf{OPT}[S] := \text{size of the optimal solution for } D[S].$$

DFAS: A DP Algorithm

- Choosing all possible arc sets takes time $\Omega(2^m)$ and $m = O(n^2)$.
- Aim: obtain an algorithm with running time $O(2^n)$.

- Given a digraph $D = (V, A)$, for $S \subseteq V$ define

$$\mathsf{OPT}[S] := \text{size of the optimal solution for } D[S].$$

- For $|S| = 1$, $\mathsf{OPT}[S] = 0$.
- For $|S| \geq 2$,

$$\mathsf{OPT}[S] = \min_{u \in S} \{\mathsf{OPT}[S \setminus u] + d^+_{D[S]}(u)\}.$$

DFAS: A DP Algorithm

- Choosing all possible arc sets takes time $\Omega(2^m)$ and $m = O(n^2)$.
- Aim: obtain an algorithm with running time $O(2^n)$.

- Given a digraph $D = (V, A)$, for $S \subseteq V$ define

$$\mathsf{OPT}[S] := \text{size of the optimal solution for } D[S].$$

- For $|S| = 1$, $\mathsf{OPT}[S] = 0$.
- For $|S| \geq 2$,

$$\mathsf{OPT}[S] = \min_{u \in S} \{\mathsf{OPT}[S \setminus u] + d^+_{D[S]}(u)\}.$$

- Time need to compute $\mathsf{OPT}[S]$ for all $S \subseteq V$: $O(2^n mn)$.

Partition Problems: Computing Chromatic Number

Definition (k-Colorings)

A k-**coloring** of an undirected graph $G = (V, E)$ is an assignment of "colors" $\{1, \ldots, k\}$ to the vertices of G such that adjacent vertices get distinct colors.

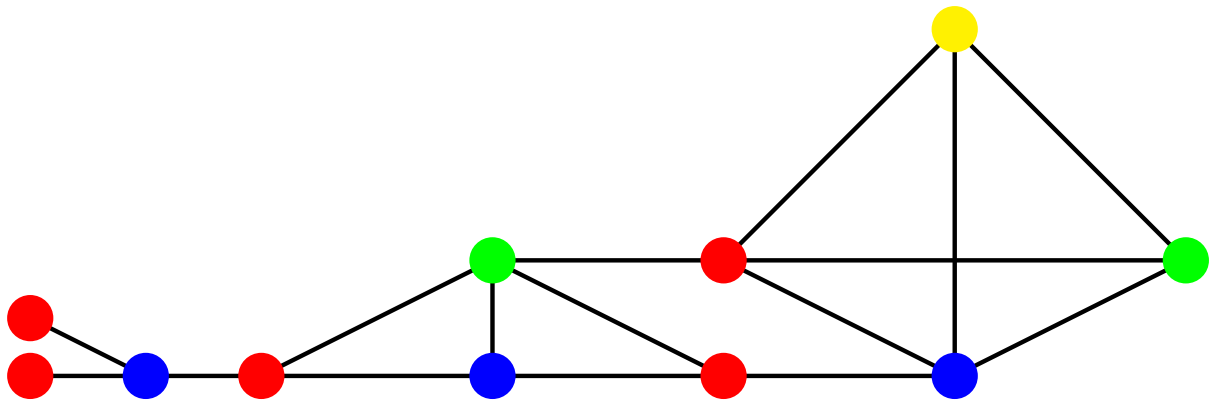


Figure: A 4-coloring of a graph.

Computing Chromatic Number

Definition (The Chromatic Number Problem)

Given an undirected graph $G = (V, E)$, find out the smallest k such that G admits a k-coloring. The smallest such integer is called the **chromatic number** of G and denoted $\chi(G)$.

Computing Chromatic Number

Definition (The Chromatic Number Problem)

Given an undirected graph $G = (V, E)$, find out the smallest k such that G admits a k-coloring. The smallest such integer is called the **chromatic number** of G and denoted $\chi(G)$.

Examples

- Trees: 2.
- Bipartite graphs: 2.
- Odd Cycles: 3.
- The complete graph K_n: n.
- Planar graphs: 4.

Computing Chromatic Number

Definition (The Chromatic Number Problem)

Given an undirected graph $G = (V, E)$, find out the smallest k such that G admits a k-coloring. The smallest such integer is called the **chromatic number** of G and denoted $\chi(G)$.

- NP-complete.
- Applications: Job scheduling.
- Computing the chromatic number by brute-force takes time $\Omega(n^n)$.

A DP Algorithm for Chromatic Number

- Given $G = (V, E)$, for $S \subseteq V$ define

$$\mathsf{OPT}[S] := \text{chromatic number of } G[S].$$

- $\mathsf{OPT}[\emptyset] = 0$.

A DP Algorithm for Chromatic Number

- Given $G = (V, E)$, for $S \subseteq V$ define

$$\mathsf{OPT}[S] := \text{chromatic number of } G[S].$$

- $\mathsf{OPT}[\emptyset] = 0$.
- For $|S| \geq 1$,

$$\mathsf{OPT}[S] = 1 + \min_{X} \{\mathsf{OPT}[S \setminus X] : X \text{ maximal independent set of } G[S]\}.$$

A DP Algorithm for Chromatic Number

Lemma

Let $G = (V, E)$ be a graph and let $S \subseteq V$. Then

$$\chi(G[S]) = 1 + \min_X \{\chi(G[S \setminus X]) \colon X \text{ maximal independent set of } G[S]\}.$$

A DP Algorithm for Chromatic Number

Lemma

Let $G = (V, E)$ be a graph and let $S \subseteq V$. Then

$$\chi(G[S]) = 1 + \min_X \{\chi(G[S \setminus X]) \colon X \text{ maximal independent set of } G[S]\}.$$

Proof

- If X is *any* maximal independent set of $G[S]$, then

$$\chi(G[S]) \leq 1 + \chi(G[S \setminus X]).$$

- Conversely if $\chi(G[S]) = i$, then there exists a partition of $V(G)$ into i independent sets $S_1, \ldots, S_i$. Hence

$$\chi(G[S]) \geq 1 + \chi(G[S \setminus S_1]) \geq 1 + \min_X \{1 + \chi(G[S \setminus X])\}.$$

A DP Algorithm for Chromatic Number

Lemma

Let $G = (V, E)$ be a graph and let $S \subseteq V$. Then

$$\chi(G[S]) = 1 + \min_X \{\chi(G[S \setminus X]) \colon X \text{ maximal independent set of } G[S]\}.$$

The exponential part of the running time taken to compute OPT$[S]$ for all $S \subseteq V$ is therefore

$$\sum_{i=1}^{n} \binom{n}{i} \cdot 2^i \leq 3^n.$$

A DP Algorithm for Chromatic Number

Lemma

Let $G = (V, E)$ be a graph and let $S \subseteq V$. Then

$$\chi(G[S]) = 1 + \min_X \{\chi(G[S \setminus X]) \colon X \text{ maximal independent set of } G[S]\}.$$

Lemma (Moon and Moser)

The number of maximal independent sets in an n-vertex graph is at most $3^{n/3}$ and these sets can be enumerated in time $O^*(3^{n/3})$.

The time taken to compute OPT$[S]$ is therefore

$$\sum_{i=1}^{n} \binom{n}{i} \cdot 3^{i/3} \leq (1 + 3^{1/3})^n < 2.4422^n.$$

Chromatic Number: Some Remarks

- The first DP algorithm for Chromatic Number with running time $O^*(2.422^n)$ was due to Lawler [1976].
- There were marginal improvements in developing faster algorithms for Chromatic Number.
- Breakthrough in 2006: Björklund, Husfeldt and Koivisto devised an algorithm using DP and inclusion-exclusion that runs in time $O^*(2^n)$.

We'll return to Chromatic Number later on!

Covering Problems

Set Cover

- **Input:** A finite nonempty universe $\mathcal{U} = \{1, \ldots, n\}$ and a family $\mathcal{F} = \{S_1, \ldots, S_m\}$ of subsets of $\mathcal{U}$.
- **Goal:** Find the minimum number of sets in $\mathcal{F}$ that cover $\mathcal{U}$.

- Assume that every element of $\mathcal{U}$ occurs in some set of $\mathcal{F}$.
- Checking all possible subsets of $\mathcal{F}$ takes time $O^*(2^m)$.
- Since $m = O(2^n)$, this is bad!
- Want an $O^*(2^n)$ algorithm!

A DP Algorithm for Set Cover

Given $(\mathcal{U} = \{1, \ldots, n\}, \mathcal{F} = \{S_1, \ldots, S_m\})$, for $X \subseteq \mathcal{U}$ define

$$\text{OPT}[X] := \text{optimum solution size of } (X, \mathcal{F}).$$

A DP Algorithm for Set Cover

Given $(\mathcal{U} = \{1, \ldots, n\}, \mathcal{F} = \{S_1, \ldots, S_m\})$, for $X \subseteq \mathcal{U}$ define

$$\text{OPT}[X] := \text{optimum solution size of } (X, \mathcal{F}).$$

- For $X = \emptyset$, $\text{OPT}[X] = 0$.

A DP Algorithm for Set Cover

Given $(\mathcal{U} = \{1, \ldots, n\}, \mathcal{F} = \{S_1, \ldots, S_m\})$, for $X \subseteq \mathcal{U}$ define

$$\text{OPT}[X] := \text{optimum solution size of } (X, \mathcal{F}).$$

- For $X = \emptyset$, $\text{OPT}[X] = 0$.
- For $|X| \geq 1$ and for some $i \in X$,

$$\text{OPT}[X] = \min_{i \in S; S \in \mathcal{F}} \{\text{OPT}[X \setminus S] + 1\}.$$

A DP Algorithm for Set Cover

Given $(\mathcal{U} = \{1, \ldots, n\}, \mathcal{F} = \{S_1, \ldots, S_m\})$, for $X \subseteq \mathcal{U}$ define

$$\text{OPT}[X] := \text{optimum solution size of } (X, \mathcal{F}).$$

- For $X = \emptyset$, $\text{OPT}[X] = 0$.
- For $|X| \geq 1$ and for some $i \in X$,

$$\text{OPT}[X] = \min_{i \in S; S \in \mathcal{F}} \{\text{OPT}[X \setminus S] + 1\}.$$

Computing $\text{OPT}[X]$ for all $X \subseteq \mathcal{U}$ takes time:

$$O^*(\sum_{j=1}^{n} \binom{n}{j} \cdot j \cdot m) = O^*(2^n \cdot nm).$$

Dominating Sets

Given a graph $G = (V, E)$, a vertex subset $X \subseteq V$ is a dominating set for G if every vertex in $V \setminus X$ has a neighbor in X.

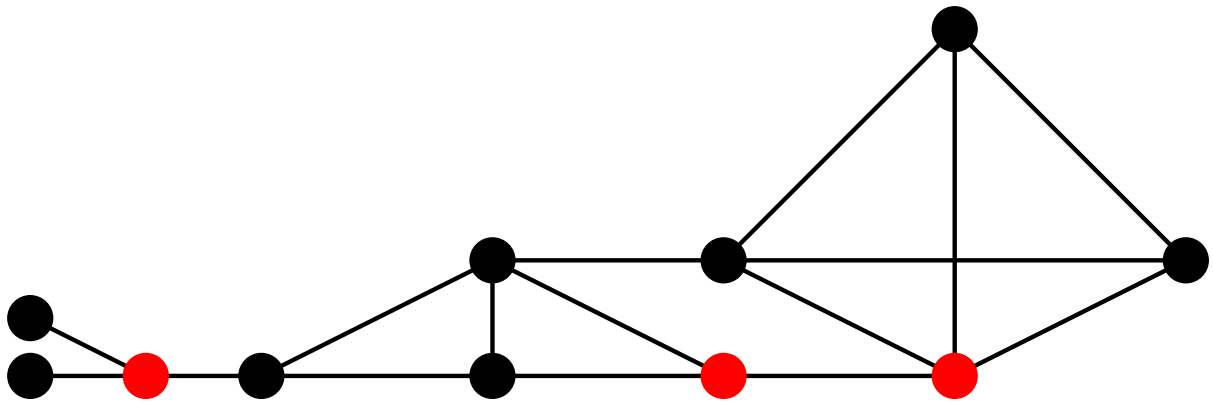


Figure: The red vertices form a dominating set.

Covering Problems

The Dominating Set Problem

Given a graph $G = (V, E)$ find a dominating set of G of smallest size.

- Can be cast as an instance $(\mathcal{U}, \mathcal{F})$ of Set Cover.
  - $\mathcal{U} = V(G)$;
  - $\mathcal{F} = \{N[v] : v \in V\}$.
  - The graph G has a k-dominating set iff $(\mathcal{U}, \mathcal{F})$ has a k-cover.
- Hence there's an $O^*(2^n)$ algorithm for Dominating Set. This is just as bad as the naive brute-force algorithm!

Dominating Set: A Better than $O^*(2^n)$ Algorithm

Basic Idea

- A *maximal independent set* in a graph is also a dominating set of the graph.
- If G is an n-vertex graph with a maximal independent set I, one can compute a minimum dominating set in time $O^*(2^{n-|I|})$.

Dominating Set: A Better than $O^*(2^n)$ Algorithm

Basic Idea

- A *maximal independent set* in a graph is also a dominating set of the graph.
- If G is an n-vertex graph with a maximal independent set I, one can compute a minimum dominating set in time $O^*(2^{n-|I|})$.

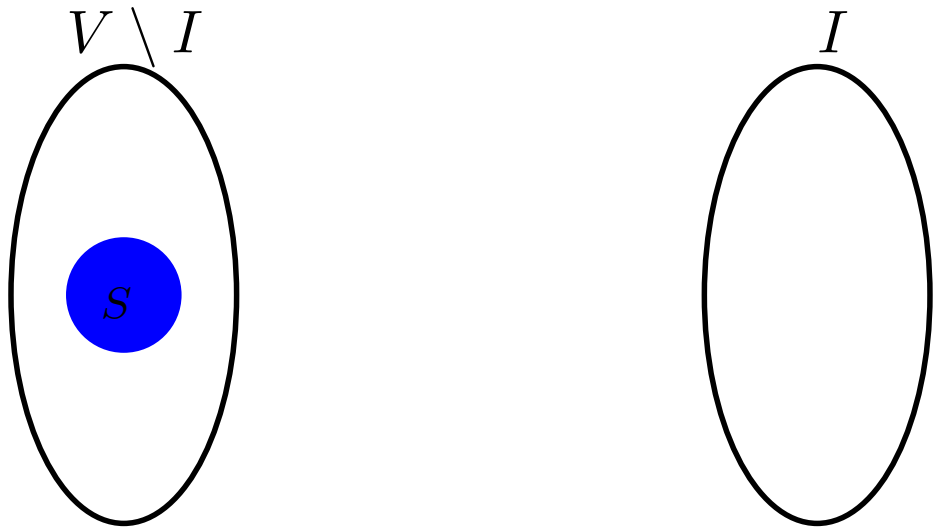
Algorithm Outline

1. Greedily compute a maximal independent set of size, say, α.
2. If α is "small", check for all subsets of size at most α.
3. Otherwise find a minimum dominating set in time $O^*(2^{n-\alpha})$.

Dominating Sets and Maximal Independent Sets

New Problem

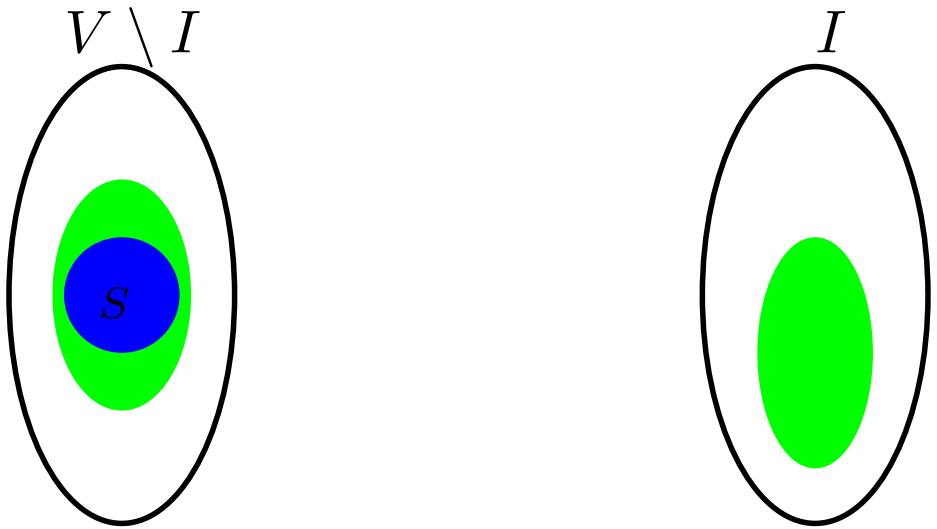
Given a graph $G = (V, E)$ and a maximal independent set I of G, and $S \subseteq V \setminus I$, find the smallest dominating set that intersects $V \setminus I$ in S.



Dominating Sets and Maximal Independent Sets

New Problem

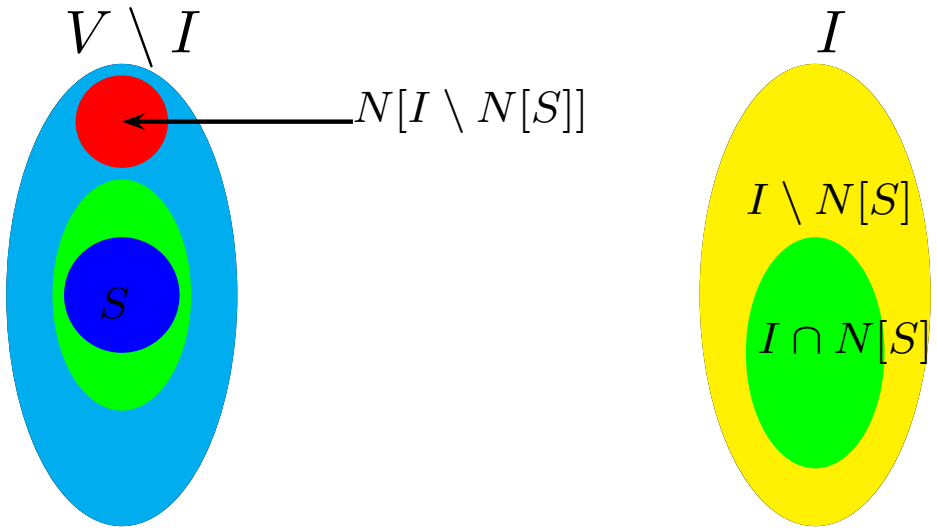
Given a graph $G = (V, E)$ and a maximal independent set I of G, and $S \subseteq V \setminus I$, find the smallest dominating set that intersects $V \setminus I$ in S.



Dominating Sets and Maximal Independent Sets

New Problem

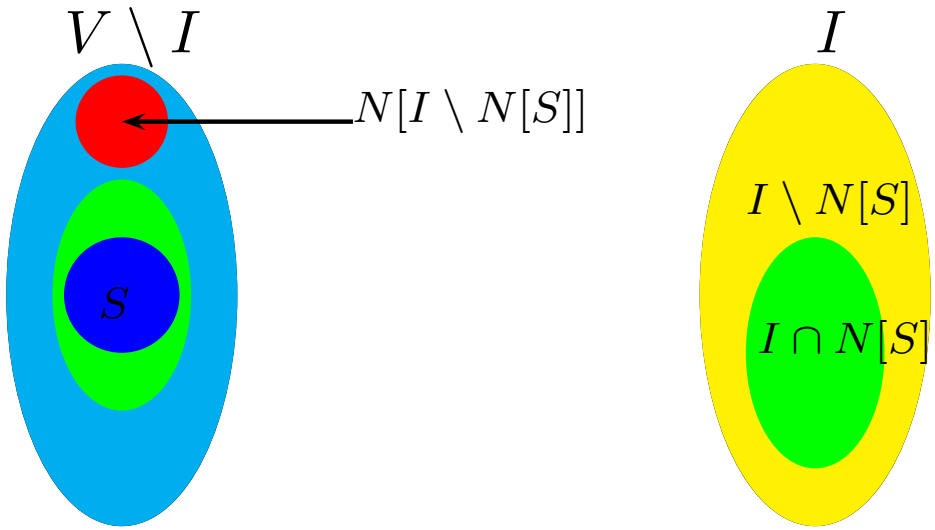
Given a graph $G = (V, E)$ and a maximal independent set I of G, and $S \subseteq V \setminus I$, find the smallest dominating set that intersects $V \setminus I$ in S.



Dominating Sets and Maximal Independent Sets

New Problem

Given a graph $G = (V, E)$ and a maximal independent set I of G, and $S \subseteq V \setminus I$, find the smallest dominating set that intersects $V \setminus I$ in S.



Equivalent Problem

Find the smallest set in $I \cap N[S]$ that dominates the cyan region.

Dominating Sets and Maximal Independent Sets

Another Problem

Given a graph $G = (V, E)$, a maximal independent set I of G and $X \subseteq V \setminus I$, find a minimum subset of I that dominates X.

If we solve this problem, then we have a solution to the previous problem.

Dominating Sets and Maximal Independent Sets

Another Problem

Given a graph $G = (V, E)$, a maximal independent set I of G and $X \subseteq V \setminus I$, find a minimum subset of I that dominates X.

If we solve this problem, then we have a solution to the previous problem.

Let $I = \{v_1, \ldots, v_k\}$ and for $X \subseteq V \setminus I$ define

$$D_{X,i} := \begin{cases} \text{a smallest subset of } \{v_1, \ldots, v_i\} \text{ that dominates } X; \\ I, \text{ if no such subset exists.} \end{cases}$$

- $D_{\emptyset,i} = \emptyset$.
- $$D_{X,1} = \begin{cases} \{v_1\}, & \text{if } X \subseteq N(v_1); \\ \{v_1, \ldots, v_k\}, & \text{otherwise.} \end{cases}$$

Computing $D_{X,i}$ for $i \geq 2$

Given $X \subseteq V \setminus I$ and $i \geq 2$,

1. X can be dominated by a subset of $\{v_1, \ldots, v_{i-1}\}$, or
2. X can be dominated by a subset of $\{v_1, \ldots, v_i\}$, or
3. X cannot be dominated by any subset of $\{v_1, \ldots, v_i\}$.

Computing $D_{X,i}$ for $i \geq 2$

Given $X \subseteq V \setminus I$ and $i \geq 2$,

1. X can be dominated by a subset of $\{v_1, \ldots, v_{i-1}\}$, or
2. X can be dominated by a subset of $\{v_1, \ldots, v_i\}$, or
3. X cannot be dominated by any subset of $\{v_1, \ldots, v_i\}$.

$$D_{X,i} = \begin{cases} D_{X,i-1}, & \text{if } |D_{X,i-1}| < |D_{X \setminus N(v_i),i-1}| + 1. \\ D_{X \setminus N(v_i),i-1} \cup \{v_i\}, & \text{otherwise.} \end{cases}$$

Note that this assignment holds good even in Case 3 above.

Time Complexity

- For fixed X, time taken to compute $D_{X,i}$ for all i is $O(n^{O(1)})$.
- To compute $D_{X,i}$ for all $X \subseteq V \setminus I$ and i takes time $O(2^{n-|I|} \cdot n^{O(1)})$.

An $O(2^{n-|I|} \cdot n^{O(1)})$ Algorithm for Dominating Set

Input: Graph $G = (V, E)$ and a maximal independent set $I = \{v_1, \ldots, v_k\}$ of G.

Algorithm

1. Compute $D_{X,k}$ for all $X \subseteq V \setminus I$.
2. For each $S \subseteq V \setminus I$, look up the smallest subset $T \subseteq I \setminus N[S]$ that dominates $(V \setminus I) \setminus (N[S] \cup N(I \setminus N[S]))$.
3. Output the smallest $S \cup T$ combination.

An $O(2^{n-|I|} \cdot n^{O(1)})$ Algorithm for Dominating Set

Input: Graph $G = (V, E)$ and a maximal independent set $I = \{v_1, \ldots, v_k\}$ of G.

Algorithm

1. Compute $D_{X,k}$ for all $X \subseteq V \setminus I$.
2. For each $S \subseteq V \setminus I$, look up the smallest subset $T \subseteq I \setminus N[S]$ that dominates $(V \setminus I) \setminus (N[S] \cup N(I \setminus N[S]))$.
3. Output the smallest $S \cup T$ combination.

Time Complexity

- Step 1: $O(2^{n-|I|} \cdot n^{O(1)})$.
- Step 2: $O(2^{n-|I|} \cdot n^{O(1)})$.
- Total time taken: $O(2^{n-|I|} \cdot n^{O(1)})$.

Computing Dominating Sets in less than $O^*(2^n)$ Time

Input: An n-vertex graph $G = (V, E)$.
Output: A minimum dominating set of G.

Algorithm

1. Greedily construct a maximal independent set I of G.
2. If $|I| \leq \alpha n$ with $0.2271 < \alpha < 0.22711$, try all possible subsets of size at most αn.
3. Else compute a minimum dominating set in time $O(2^{n-|I|} \cdot n^{O(1)})$.

Computing Dominating Sets in less than $O^*(2^n)$ Time

Input: An n-vertex graph $G = (V, E)$.
Output: A minimum dominating set of G.

Algorithm

1. Greedily construct a maximal independent set I of G.
2. If $|I| \leq \alpha n$ with $0.2271 < \alpha < 0.22711$, try all possible subsets of size at most αn.
3. Else compute a minimum dominating set in time $O(2^{n-|I|} \cdot n^{O(1)})$.

Time Complexity

For $\alpha \approx 0.22711$, $\binom{n}{\alpha n} = 2^{n-\alpha n} = 1.7088^n$.

$$\text{Run Time: } \binom{n}{\alpha n} \cdot n^{O(1)} + 2^{n-|I|} \cdot n^{O(1)} = 1.7088^n \cdot n^{O(1)}.$$

Permutation Problems (Again!)

The Travelling Salesperson Problem (TSP)

- **Input:** A set of n cities $c_1, \ldots, c_n$ and the distance $d(c_i, c_j)$ between every pair of cities.
- **Goal:** Construct a tour that starts at city c_1, visits all the remaining cities and returns to c_1, and is of minimum total length.

Formulation as a Graph-Theoretic Problem

Given a set of cities $C = \{c_1, \ldots, c_n\}$ and a distance function $d\colon C \times C \to \mathbb{N} \cup \{\infty\}$, we associate the graph $G = (V, E)$ where

- $V = \{c_1, \ldots, c_n\}$;
- $E = \{\{c_i, c_j\} \colon d(c_i, c_j) < \infty\}$.

TSP: Graphs of Bounded Degree

Claim

For graphs of bounded maximum degree, the Held-Karp DP algorithm can be modified to run in time $O^*(c^n)$ time, for some $c < 2$.

TSP: Graphs of Bounded Degree

Claim

For graphs of bounded maximum degree, the Held-Karp DP algorithm can be modified to run in time $O^*(c^n)$ time, for some $c < 2$.

- For $S \subseteq V$ and $c_i \in S$, define

$$\text{OPT}[S, c_i] := \text{length of optimal tour that starts at } c_1, \text{ visits all cities of } S, \text{ in } \textit{some} \text{ order, and ends at } c_i.$$

- **Observation:** $\text{OPT}[S, c_i] = \infty$ if $G[S]$ is *not* connected.
- Running time is dependent on the number of connected vertex sets, which in general graphs is $O(2^n)$; consider, for example, a clique: *every* vertex subset is connected.
- The number of connected subsets is significantly smaller in bounded degree graphs.

Connected Vertex Sets in Graphs of Bounded Degree

First we state a (a quite technical) lemma that we'll use as a blackbox.

Lemma (Shearer)

Let

- $\mathcal{U} = \{1, \ldots, n\}$;
- $\mathcal{F} = \{S_1, \ldots, S_r\}$ *be a family of subsets of* $\mathcal{U}$,

such that every element of $\mathcal{U}$ *is contained in at least* k *subsets of* $\mathcal{F}$.
Let $\mathcal{T}$ *be an arbitrary family of subsets and for* $1 \leq i \leq r$, *define*

$$\mathcal{T}_i = \{T \cap S_i : T \in \mathcal{T}\}.$$

Then

$$|\mathcal{T}|^k \leq \prod_{i=1}^{r} |\mathcal{T}_i|.$$

Using Shearer's Lemma

Given a graph $G = (V, E)$ with minimum degree δ, set $\mathcal{U} = V$, $\mathcal{F} = E$ and $\mathcal{T}$ to be the set of all subsets of V.

- If $e = \{u, v\} \in E$ then $\mathcal{T}_e = \{\emptyset, \{u\}, \{v\}, \{u, v\}\}$.
- By Shearer's Lemma, $|\mathcal{T}|^\delta \leq \prod_{e \in E} |\mathcal{T}_e|$, which yields

$$\begin{aligned} 2^{n\delta} &\leq 4^m, \\ n\delta &\leq 2m, \\ \delta &\leq 2m/n. \end{aligned}$$

The minimum degree is at most the average degree.

Using Shearer's Lemma

Independent Sets in Regular Graphs

Given a Δ-regular graph $G = (V, E)$, set $\mathcal{U} = V$, $\mathcal{F} = \{N[v] \colon v \in V\}$ and $\mathcal{T}$ to be the set of all independent sets of G of size at least two.

- Every vertex appears in exactly $\Delta + 1$ sets of $\mathcal{F}$.
- If $v \in V$ then $|\mathcal{T}_v| \leq 2^\Delta + 1$.
- By Shearer's Lemma, $|\mathcal{T}|^{\Delta+1} \leq \prod_{v \in V} |\mathcal{T}_v|$, which yields

$$|\mathcal{T}| \leq (2^\Delta + 1)^{n/(\Delta+1)}.$$

- Since there are n independent sets of size one, the total number of independent sets is at most $(2^\Delta + 1)^{n/(\Delta+1)} + n$.

Connected Vertex Sets in Graphs of Bounded Degree

Lemma

Let $G = (V, E)$ be an n-vertex graph with maximum degree Δ. The number of connected vertex subsets in G is at most

$$[(2^{\Delta+1} - 1)^{1/(\Delta+1)}]^n + n.$$

Proof

Using Shearer's Lemma and some facts on convex functions.

- Set $\mathcal{U} = V$ and $\mathcal{F} = \{S_v : v \in V\}$, where initially $S_v = N[v]$.
- If $v \in V$ is not of maximum degree, that is, $d(v) < \Delta$, add v to $\Delta - d(v)$ sets that do not contain v arbitrarily.
- Every vertex is now contained in exactly $\Delta + 1$ sets.
- Finally, let $\mathcal{T}$ be the set of all connected vertex sets of size at least two.

Connected Vertex Sets in Graphs of Bounded Degree

Lemma

Let $G = (V, E)$ be an n-vertex graph with maximum degree Δ. The number of connected vertex subsets in G is at most

$$[(2^{\Delta+1} - 1)^{1/(\Delta+1)}]^n + n.$$

Proof

- For $v \in V$, let $\mathcal{T}_v = \{T \cap S_v : T \in \mathcal{T}\}$. Then $|\mathcal{T}_v| \leq 2^{|S_v|} - 1$.
- By Shearer's Lemma, $|\mathcal{T}|^{\Delta+1} \leq \prod_{v \in V}(2^{|S_v|} - 1)$.

Connected Vertex Sets in Graphs of Bounded Degree

Claim

If $f(x) = 2^x - 1$ then $g(x) = -\log f(x)$ is convex on the interval $[1, \infty]$.

Claim

If $f(x) = 2^x - 1$ then $g(x) = -\log f(x)$ is convex on the interval $[1, \infty]$.

Proof

A differentiable function g is convex on an interval I if g' is increasing on I. Thus $g'' \geq 0$ if g' is differentiable.

- $g(x) = -\ln(2^x - 1)/\ln 2$ and $g'(x) = -2^x/(2^x - 1)$ which (strictly) increases from -2 at $x = 1$ to -1 at $x = \infty$.
- $g''(x) = \frac{2^x \ln 2}{2^x - 1}[\frac{2^x}{2^x - 1} - 1] > 0$ for $x \geq 1$.

Claim

If $f(x) = 2^x - 1$ then $g(x) = -\log f(x)$ is convex on the interval $[1, \infty]$.

Jensen's Inequality

If $f : I \to \mathbb{R}^1$ is convex on the interval I and if $p_1, \ldots, p_k \in [0, 1]$ are reals such that $\sum_{i=1}^{k} p_i = 1$ then for all $x_1, \ldots, x_k \in I$

$$f(\sum_{i=1}^{k} p_i x_i) \leq \sum_{i=1}^{p} p_i f(x_i).$$

Lemma

Let $G = (V, E)$ be an n-vertex graph with maximum degree Δ. The number of connected vertex subsets in G is at most

$$[(2^{\Delta+1} - 1)^{1/(\Delta+1)}]^n + n.$$

Proof (Continued)

$$\begin{aligned} -\log f(\frac{1}{n}\sum_{i=1}^{n} x_i) &\leq -\sum_{i=1}^{n} \frac{1}{n} \log f(x_i) \\ \frac{1}{n}\sum_{v\in V} \log f(|S_v|) &\leq \log f(\frac{1}{n}\sum_{v\in V} |S_v|) \leq \log f(\Delta + 1). \end{aligned}$$

Lemma

Let $G = (V, E)$ be an n-vertex graph with maximum degree Δ. The number of connected vertex subsets in G is at most

$$[(2^{\Delta+1} - 1)^{1/(\Delta+1)}]^n + n.$$

Proof (Continued)

$$\begin{aligned} \sum_{v \in V} \log f(|S_v|) &\leq \log[f(\Delta+1)]^n \\ \prod_{v \in V} f(|S_v|) &\leq f(\Delta+1)^n \\ |\mathcal{T}| &\leq (2^{\Delta+1} - 1)^{n/(\Delta+1)}. \end{aligned}$$

Connected Vertex Sets in Graphs of Bounded Degree

Lemma

Let $G = (V, E)$ be an n-vertex graph with maximum degree Δ. The number of connected vertex subsets in G is at most

$$[(2^{\Delta+1} - 1)^{1/(\Delta+1)}]^n + n.$$

Proof (Continued)

$$|\mathcal{T}| \leq (2^{\Delta+1} - 1)^{n/(\Delta+1)}.$$

Since the number of connected vertex sets of size one is n, the total number of connected vertex sets is $(2^{\Delta+1} - 1)^{n/(\Delta+1)} + n$.

DP Algorithm for Bounded Degree TSP

Algorithm TSP

Input: A set $\mathcal{C} = \{c_1, \ldots, c_n\}$ of cities and $d\colon \mathcal{C} \times \mathcal{C} \to \mathbb{N} \cup \{\infty\}$.
Output: The cost of an optimal tour that starts and ends at c_1.

1. **for** $i = 2$ to n **do**
   OPT$[\{c_i\}, c_i] := d(c_1, c_i)$;
2. **for** $k = 2$ to n **do**
2.1. **forall** connected $S \subseteq \{2, \ldots, n\}$ with $|S| = k - 1$ **do**
2.2. **forall** $c_i \in S$ and $c_j \in \mathcal{C} \setminus S$ such that $d(c_i, c_j) < \infty$ **do**
2.3 OPT$[S \cup c_j, c_j] = \min_{c_l \in S}\{\text{OPT}[S, c_l] + d(c_l, c_j)\}$;
3. **return** $\min_i\{\text{OPT}[S \setminus c_1, c_i] + d(c_i, c_1)\}$;

DP Algorithm for Bounded Degree TSP

Algorithm TSP

Input: A set $\mathcal{C} = \{c_1, \ldots, c_n\}$ of cities and $d\colon \mathcal{C} \times \mathcal{C} \to \mathbb{N} \cup \{\infty\}$.
Output: The cost of an optimal tour that starts and ends at c_1.

1. **for** $i = 2$ to n **do**
   OPT$[\{c_i\}, c_i] := d(c_1, c_i)$;
2. **for** $k = 2$ to n **do**
2.1. **forall** connected $S \subseteq \{2, \ldots, n\}$ with $|S| = k - 1$ **do**
2.2. **forall** $c_i \in S$ and $c_j \in \mathcal{C} \setminus S$ such that $d(c_i, c_j) < \infty$ **do**
2.3 OPT$[S \cup c_j, c_j] = \min_{c_l \in S}\{\text{OPT}[S, c_l] + d(c_l, c_j)\}$;
3. **return** $\min_i\{\text{OPT}[S \setminus c_1, c_i] + d(c_i, c_1)\}$;

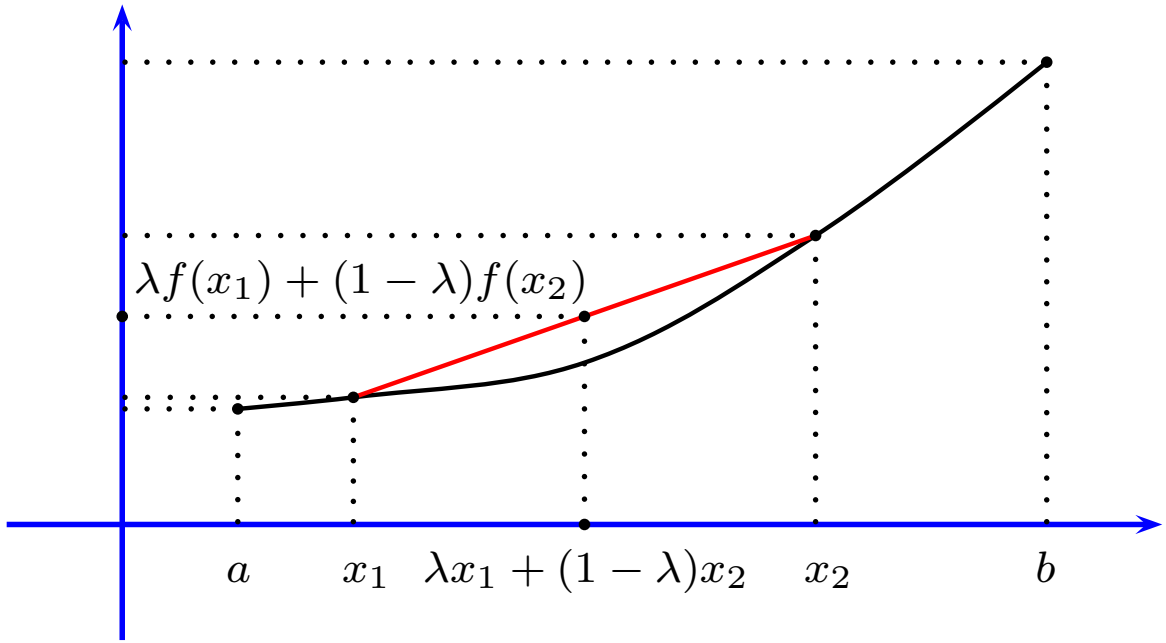
An appropriate data structure is needed such that in step 2.1 only *connected* sets of size $k - 1$ are considered.

Notes on Convex Functions

Definition

A function $f : I \to \mathbb{R}^1$ on an interval I is said to be on convex on I if for all $0 \leq \lambda \leq 1$ and $x_1, x_2 \in I$,

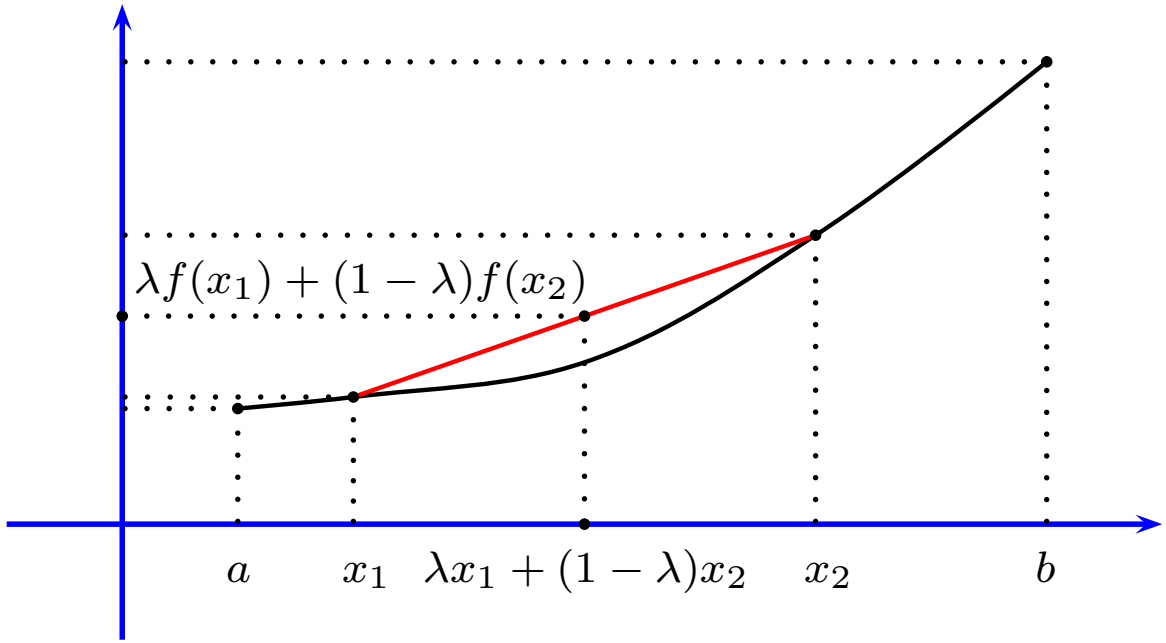
$$f(\lambda x_1 + (1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda) f(x_2).$$



Notes on Convex Functions

Definition

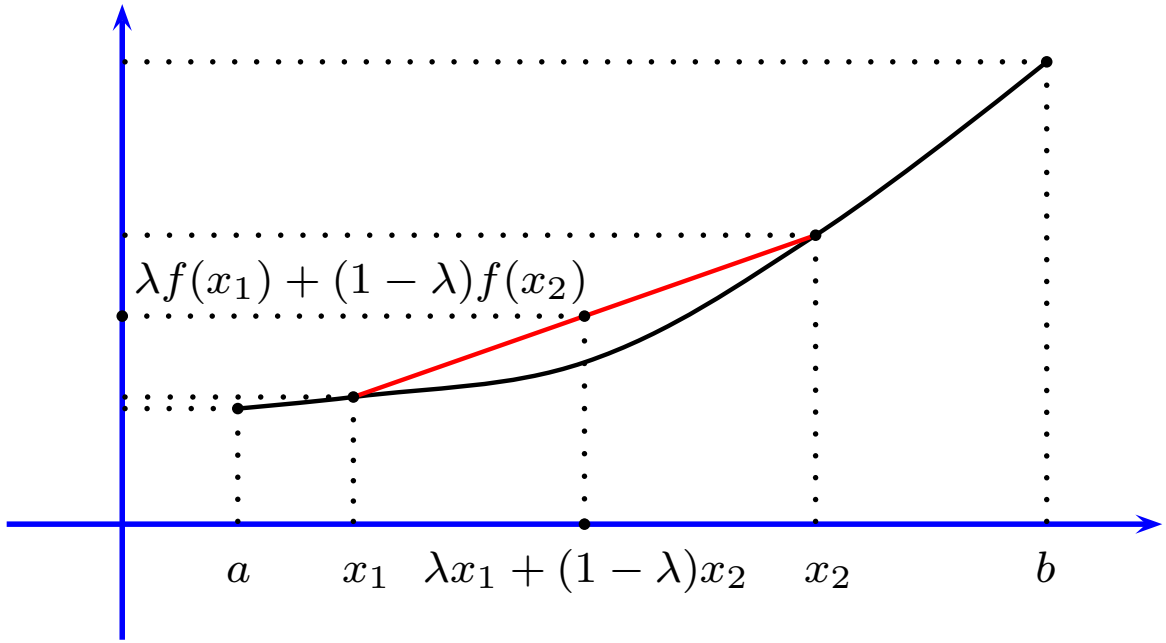
A function $f : I \to \mathbb{R}^1$ on an interval I is said to be on convex on I if for any two points $x_1, x_2 \in I$, the graph of the function "lies below" the chord through $(x_1, f(x_1))$ and $(x_2, f(x_2))$.



Differentiability and Convex Functions

Lemma

A function $f : I \to \mathbb{R}^1$ on an interval I is convex on I if the slope of the tangent to f does not decrease on I. That is, f' is non-decreasing on I and thus $f'' \geq 0$ (if f' is differentiable).



Examples of Convex Functions

- affine: $ax + b$ for all $a, b \in \mathbb{R}$ and $a \neq 0$.
- exponential: e^{ax} for any $a \in \mathbb{R}$.
- power: x^a for $a \geq 1$.

Jensen's Inequality

Jensen's Inequality

If $f : I \to \mathbb{R}^1$ is convex on the interval I and if $p_1, \ldots, p_k \in [0, 1]$ are reals such that $\sum_{i=1}^{k} p_i = 1$ then for all $x_1, \ldots, x_k \in I$

$$f(\sum_{i=1}^{k} p_i x_i) \leq \sum_{i=1}^{p} p_i f(x_i).$$

Jensen's Theorem: A Proof

Proof

By induction on k.

- For $k = 2$, this follows from the definition of convexity.
- Assume the theorem to hold for k values $p_1, \ldots, p_k$.

For $k+1$ values $p_1, \ldots, p_{k+1}$,

$$f(p_1 x_1 + \cdots + p_{k+1} x_{k+1}) \leq p' f(x) + p_{k+1} f(x_{k+1}),$$

where $x = (p_1 x_1 + \cdots + p_k x_k)/(p_1 + \cdots + p_k) = p'_1 x_1 + \cdots p'_k x_k$.

Since $\sum_{i=1}^{k} p'_i = 1$, we have by induction hypothesis,

$$f(p'_1 x_1 + \cdots + p'_k x_k) \leq p'_1 f(x_1) + \cdots + p'_k f(x_k).$$

This proves the theorem.

Partition Problems: Once More!

Partitioning into Sets

Given

- $\mathcal{U} = \{1, \ldots, n\}$;
- $\mathcal{F} = \{S_1, \ldots, S_r\}$ where $S_i \subseteq \mathcal{U}$.

A k-**partition of** $(\mathcal{U}, \mathcal{F})$ is a collection of k pairwise disjoint members $S_1, \ldots, S_k$ of $\mathcal{F}$ such that

$$\bigcup_{i=1}^{k} S_i = \mathcal{U}.$$

Question: How many *unordered* k-partitions of $(\mathcal{U}, \mathcal{F})$ are there?

An Example of Set Partitioning

Counting Perfect Matchings

Given an n-vertex graph $G = (V, E)$, set $\mathcal{U} = V$ and $\mathcal{F} = E$.

- an $n/2$-partition of $(\mathcal{U}, \mathcal{F})$ = a perfect matching in G.

Counting $n/2$-partitions is counting the number of perfect matchings.

- Finding a maximum matching in a graph is polynomial-time solvable.
- Counting the number of perfect matchings even in bipartite graphs is #P-complete.

Some Assumptions

- The family $\mathcal{F}$ is usually given *implicitly*: there is a polynomial (in $|\mathcal{U}|$) time algorithm that recognizes elements of $\mathcal{U}$.
- Each set of $\mathcal{F}$ has bounded cardinality.

A DP Solution to the Partitioning Problem

Input: A finite set system $(\mathcal{U}, \mathcal{F})$ and an integer k.
Goal: Count the number of k partitions of $(\mathcal{U}, \mathcal{F})$.

Subproblem Definition

For $X \subseteq \mathcal{U}$ and $1 \leq j \leq k$, define

$$\mathsf{OPT}[X, j] := \text{the number of } j\text{-partitions of } (X, \mathcal{F}).$$

Then

$$\mathsf{OPT}[X, 1] = \begin{cases} 1, & \text{if } X \in \mathcal{F}; \\ 0, & \text{otherwise.} \end{cases}$$

For $j \geq 2$,

$$\mathsf{OPT}[X, j] = \frac{1}{j} \sum_{S \in \mathcal{F}, S \subseteq X} \mathsf{OPT}[X \setminus S, j-1].$$

A DP Solution to the Partitioning Problem

Input: A finite set system $(\mathcal{U}, \mathcal{F})$ and an integer k.
Goal: Count the number of k partitions of $(\mathcal{U}, \mathcal{F})$.

Subproblem Definition

For $X \subseteq \mathcal{U}$ and $1 \leq j \leq k$, define

$$\text{OPT}[X, j] := \text{the number of } j\text{-partitions of } (X, \mathcal{F}).$$

Then

$$\text{OPT}[X, 1] = \begin{cases} 1, & \text{if } X \in \mathcal{F}; \\ 0, & \text{otherwise.} \end{cases}$$

For $j \geq 2$,

$$\text{OPT}[X, j] = \frac{1}{j} \sum_{S \in \mathcal{F}, S \subseteq X} \text{OPT}[X \setminus S, j-1].$$

Every partition $S_1, \ldots, S_j$ is counted exactly j times in the right-hand side; once in $\text{OPT}[X \setminus S_i]$ for each $1 \leq i \leq j$.

A DP Solution to the Partitioning Problem

Input: A finite set system $(\mathcal{U}, \mathcal{F})$ and an integer k.
Goal: Count the number of k partitions of $(\mathcal{U}, \mathcal{F})$.

Subproblem Definition

For $X \subseteq \mathcal{U}$ and $1 \leq j \leq k$, define

$$\mathsf{OPT}[X, j] := \text{the number of } j\text{-partitions of } (X, \mathcal{F}).$$

Then

$$\mathsf{OPT}[X, 1] = \begin{cases} 1, & \text{if } X \in \mathcal{F}; \\ 0, & \text{otherwise.} \end{cases}$$

For $j \geq 2$,

$$\mathsf{OPT}[X, j] = \frac{1}{j} \sum_{S \in \mathcal{F}, S \subseteq X} \mathsf{OPT}[X \setminus S, j-1].$$

Time Taken: $\sum_{i=1}^{n} \binom{n}{i} \cdot 2^i \cdot i \cdot k = O^*(3^n)$. There's a polynomial-time membership check for $\mathcal{F}$.

Counting Partitions When Set Sizes are Bounded

When the set sizes are "small" then the number of k-partitions can be computed in time $O^*((2-\epsilon)^n)$ for some $\epsilon > 0$.

Theorem

Let $(\mathcal{U}, \mathcal{F})$ be a finite set system such that each member of $\mathcal{F}$ has size at most $r \geq 1$. Then the number of k-partitions can be computed in time

$$O^*\left(\sum_{i=1}^{k}\sum_{j=i}^{ir}\binom{n-i}{j-i}\cdot|\mathcal{F}|\right).$$

Proof

Part 1: Lex Orderings

We may assume that the elements of $\mathcal{U}$ are linearly ordered

$$u_1 < u_2 < \cdots < u_n.$$

This total ordering allows any subset of $\mathcal{U}$ to be uniquely written as an increasing sequence of elements of $\mathcal{U}$.

Proof

Part 1: Lex Orderings

For $S_1, S_2 \in \mathcal{F}$ with $S_1 = a_1, \ldots, a_p$ and $S_2 = b_1, \ldots, b_q$, we say that S_1 is **lexicographically lesser than** S_2 and write $S_1 \prec S_2$ iff either

- $p \leq q$ and $a_i = b_i$ for $i = 1, \ldots, p$; or
- for some i, $a_i \neq b_i$ and for the smallest such i, we have $a_i < b_i$.

Example

- $\{u_1, u_7\} \prec \{u_1, u_7, u_8\}$;
- $\{u_1, u_7\} \prec \{u_2\}$;
- $\{u_3, u_4, u_8\} \prec \{u_3, u_4, u_{11}\}$.
- $\{u_1, u_2, u_4, u_5, u_6\} \prec \{u_1, u_2, u_5\}$.

Proof: Lex Orderings

Definition

- A k-partition $(S_1, \ldots, S_k)$ of $(\mathcal{U}, \mathcal{F})$ is **lexicographically ordered** if $S_1 \prec S_2 \prec \cdots \prec S_k$.
- Define $\mathcal{L}_k$ to be the set of lexicographically ordered k-partitions of $(\mathcal{U}, \mathcal{F})$.

Observations

- $|\mathcal{L}_k|$ equals the number of unordered k-partitions of $(\mathcal{U}, \mathcal{F})$.
- If $(S_1, \ldots, S_k) \in \mathcal{L}_k$ then
  - $u_1 \in S_1$;
  - for $2 \leq j \leq k$, S_j contains the smallest element not contained in $\bigcup_{i=1}^{j-1} S_i$.

Proof: Relevant Sets

- Lex ordering imposes certain constraints on the tuples $(S_1, \ldots, S_k) \in \mathcal{L}_k$.
- To compute $\mathcal{L}_k$, it suffices to do dynamic programming on certain relevant sets only.

Proof: Relevant Sets

- Lex ordering imposes certain constraints on the tuples $(S_1, \ldots, S_k) \in \mathcal{L}_k$.
- To compute $\mathcal{L}_k$, it suffices to do dynamic programming on certain relevant sets only.

Part 2: Relevant Sets

For $1 \leq i \leq k$, define the i**th relevant family** $\mathcal{R}_i$ as follows:

- $\mathcal{R}_1 := \{S \in \mathcal{F} : u_1 \in S\}$;
- $\mathcal{R}_i := \{T \cup S : T \in \mathcal{R}_{i-1} \wedge S \in \mathcal{F} \wedge T \cap S = \emptyset \wedge \min\{\mathcal{U} \setminus T\} \in S\}$.

The members of $\mathcal{R}_j$ are called **relevant sets**.

Observations

- Every set from $\mathcal{R}_j$ contains $u_1, \ldots, u_j$.
- If $(S_1, \ldots, S_k)$ is a lex-ordered partition, then $S_1 \cup \cdots \cup S_j \in \mathcal{R}_j$.

Proof: DP Algorithm

Part 3: Subproblem Definition

For every relevant set $X \subseteq \mathcal{U}$ and $1 \leq j \leq k$, define $\mathsf{OPT}[X; j]$ as

$$\mathsf{OPT}[X; j] := \text{the number of lex ordered } j\text{-partitions of } (X, \mathcal{F}).$$

Then $\mathsf{OPT}[\mathcal{U}; k] = |\mathcal{L}_k|$. For $j = 1$,

$$\mathsf{OPT}[X; j] = \begin{cases} 1, & \text{if } X \in \mathcal{R}_1; \\ 0, & \text{otherwise.} \end{cases}$$

For $j \geq 2$,

$$\mathsf{OPT}[X; j] = \sum_{S \subseteq X} \mathsf{OPT}[X \setminus S; j-1],$$

where the sum is taken over sets $S \subseteq X$ that satisfy

- $S \in \mathcal{F}$;
- $X \setminus S \in \mathcal{R}_{j-1}$;
- $\min\{\mathcal{U} \setminus (X \setminus S)\} \in S$.

Proof: Running Time of DP Algorithm

Part 4: Computing Relevant Sets

- Computing $\mathcal{R}_1$ takes time $O(|\mathcal{F}| \cdot r)$, where r is the upper bound on the size of a set in $\mathcal{F}$.
- For $2 \leq j \leq k$, computing $\mathcal{R}_j$ takes time $O(|\mathcal{F}| \cdot |\mathcal{R}_{j-1}| \cdot n)$.
- Total time taken: $O^*(|\mathcal{F}| \cdot \sum_{i=1}^{k-1} |\mathcal{R}_i|)$.

One can store each family $\mathcal{R}_i$ in a tree structure such that testing whether $X \in \mathcal{R}_i$ takes time $O(|\mathcal{U}|)$.

Part 5: Running Time of DP Algorithm

The time taken to compute OPT$[X; i]$ for all $X \in \mathcal{R}_j$ takes time $O(|\mathcal{R}_j| \cdot |\mathcal{F}| \cdot (r + n))$. Hence the total time taken is

$$O^* \left(|\mathcal{F}| \cdot \sum_{i=1}^{k} |\mathcal{R}_i| \right).$$

Proof: Running Time

Total Running Time

$$O^*\left(|\mathcal{F}| \cdot \sum_{i=1}^{k} |\mathcal{R}_i|\right).$$

- Every set in $\mathcal{R}_i$ contains the elements $u_1, \ldots, u_i$.
- Every set in $\mathcal{R}_i$ has size at most ir.
- Hence every set in $\mathcal{R}_i$ contains at most $i(r-1)$ elements from among $u_{i+1}, \ldots, u_n$.

$$\begin{aligned} |\mathcal{R}_i| &\leq \sum_{j=1}^{i(r-1)} \binom{n-i}{j} \leq \sum_{j=0}^{i(r-1)} \binom{n-i}{j} \\ &= \sum_{j=i}^{ir} \binom{n-i}{j-i}. \end{aligned}$$

Proof: Running Time

Total Running Time

The total running time is $O^*(|\mathcal{F}| \cdot \sum_{i=1}^{k} |\mathcal{R}_i|)$. Since

$$|\mathcal{R}_i| \leq \sum_{j=i}^{ir} \binom{n-i}{j-i},$$

this expression works out to

$$O^* \left(\sum_{i=1}^{k} \sum_{j=i}^{ir} \binom{n-i}{j-i} \cdot |\mathcal{F}| \right).$$

The quantity $\sum_{i=1}^{k} \sum_{j=i}^{ir} \binom{n-i}{j-i}$ can be shown to be $2^{(1-\epsilon)n}$ for some $\epsilon > 0$. For details see *Partitioning into Sets of Bounded Cardinality*, Mikko Koivisto, IWPEC 2009.

An Application: Counting Perfect Matchings

Given an n-vertex graph $G = (V, E)$, set

- $\mathcal{U} = V = \{v_1, \ldots, v_n\}$, $\mathcal{F} = E$;
- $r = 2$, $k = n/2$.

The running time of the previous algorithm is proportional to

$$O^* \left(|\mathcal{F}| \cdot \sum_{i=1}^{k} |\mathcal{R}_i| \right).$$

An Application: Counting Perfect Matchings

Given an n-vertex graph $G = (V, E)$, set

- $\mathcal{U} = V = \{v_1, \ldots, v_n\}$, $\mathcal{F} = E$;
- $r = 2$, $k = n/2$.

The running time of the previous algorithm is proportional to

$$O^* \left(|\mathcal{F}| \cdot \sum_{i=1}^{k} |\mathcal{R}_i| \right).$$

In this case,

- each set in $\mathcal{R}_j$ contains the elements $v_1, \ldots, v_j$ and has size exactly $2j$;
- thus each set in $\mathcal{R}_j$ has at most j elements from $\{v_{j+1}, \ldots, v_n\}$.

An Application: Counting Perfect Matchings

Given an n-vertex graph $G = (V, E)$, set

- $\mathcal{U} = V = \{v_1, \ldots, v_n\}$, $\mathcal{F} = E$;
- $r = 2$, $k = n/2$.

The running time of the previous algorithm is proportional to

$$O^*\left(|\mathcal{F}| \cdot \sum_{i=1}^{k} |\mathcal{R}_i|\right).$$

$|R_j| \leq \binom{n-j}{j}$ and since

$$\sum_{j=0}^{\lfloor n/2 \rfloor} \binom{n-j}{j} < \varphi^n,$$

where $\varphi = (\sqrt{5}+1)/2$, we have

Running Time

Time to compute all perfect matchings $= O^*(1.6181^n)$.

Notes on Binomial Sums

Lemma

$$\sum_{j=0}^{\lfloor n/2 \rfloor} \binom{n-j}{j} = F_{n+1} < \varphi^n,$$

where F_{n+1} is the $n+1$st Fibonacci number.

Proof

Let a_n denote the number of binary strings of length n with no consecutive zeroes. We'll count them in two ways.

- $a_1 = |\{1, 0\}| = 2$;
- $a_2 = |\{11, 10, 01\}| = 3$;
- $a_3 = |\{111, 110, 101, 011, 010\}| = 5$;
- $a_4 = |\{1111, 1110, 1101, 1011, 0111, 1010, 0101, 0110\}| = 8$;

Notes on Binomial Sums

Lemma

$$\sum_{j=0}^{\lfloor n/2 \rfloor} \binom{n-j}{j} = F_{n+1} < \varphi^n,$$

where F_{n+1} is the $n+1$st Fibonacci number.

Proof

Let a_n denote the number of binary strings of length n with no consecutive zeroes. We'll count them in two ways.

Claim. $a_n = a_{n-1} + a_{n-2}$.

- Strings of length n start with either 0 or 1.
- There are a_{n-1} strings that start with 1 and a_{n-2} strings that start with 0.

This proves the claim. Note that $a_n = F_{n+2}$.

Lemma

$$\sum_{j=0}^{\lfloor n/2 \rfloor} \binom{n-j}{j} = F_{n+1} < \varphi^n,$$

where F_{n+1} is the $n+1$st Fibonacci number.

Proof

The number of binary strings of length $k+r$ with r nonconsecutive zeroes is

$$\binom{k+1}{r}.$$

Thus the number of length n binary strings with r nonconsecutive zeroes is

$$\binom{n-r+1}{r}.$$

Lemma

$$\sum_{j=0}^{\lfloor n/2 \rfloor} \binom{n-j}{j} = F_{n+1} < \varphi^n,$$

where F_{n+1} is the $n+1$st Fibonacci number.

Proof

Hence $a_n = \sum_{r=0}^{\lfloor n/2 \rfloor} \binom{n-r+1}{r}$ and

$$a_{n-1} = F_{n+1} = \sum_{r=0}^{\lfloor n/2 \rfloor} \binom{n-r}{r} < \varphi^n.$$

Overview

1. Introduction
2. Branching
3. Measure&Conquer
4. Dynamic Programming
5. Local Search
6. Split & List
7. Lower Bounds
8. Inclusion-exclusion
9. Subset Convolution

Local Search

What is *local search*?

Given a (partial) solution S, modify S locally to obtain new (partial) solution.

local: only small changes, polynomial time

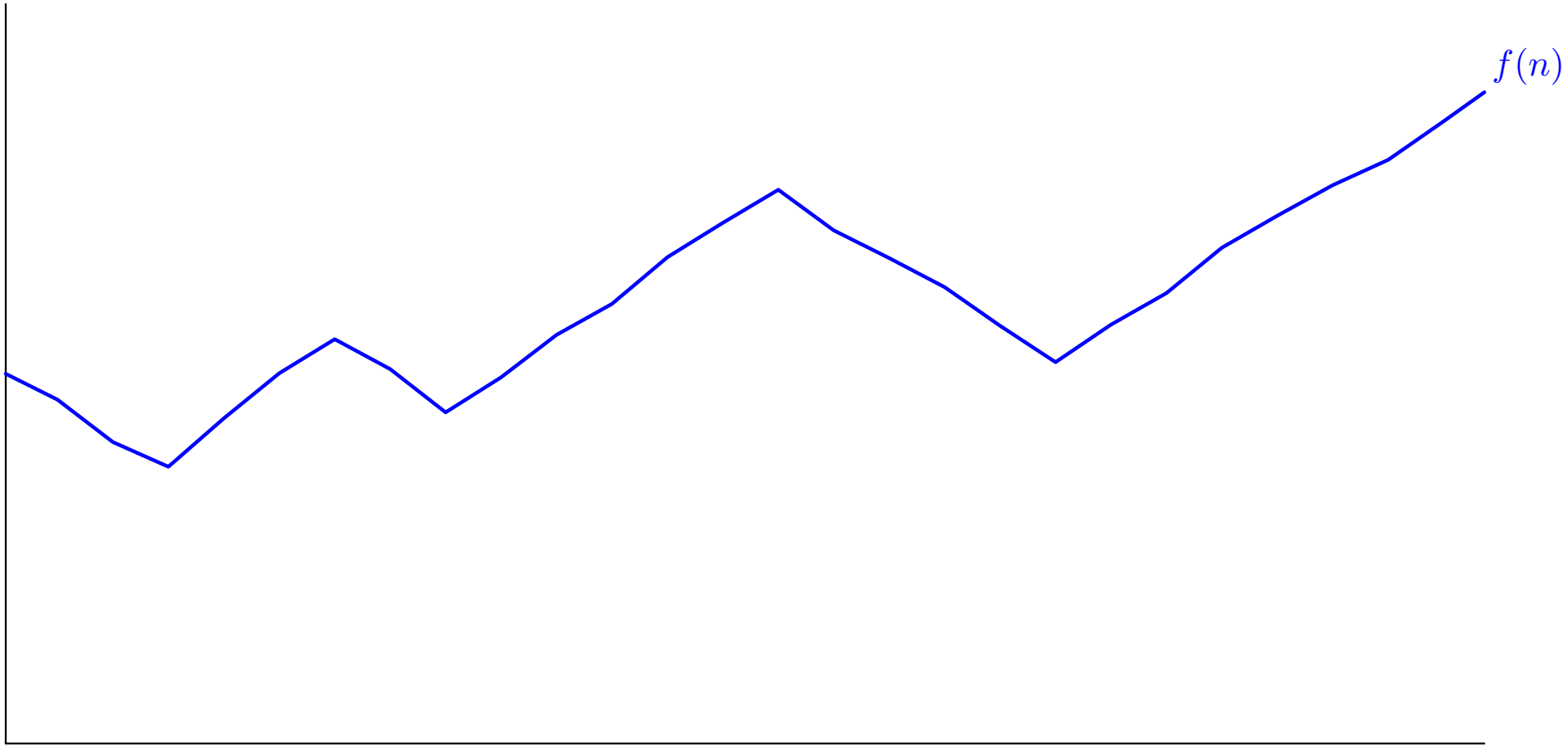
new solution: often better solution (optimization problems)

Repeat until no improvement possible

- Often only heuristic results, no guarantees on success
- Meta-Heuristic
- Very successful in practice

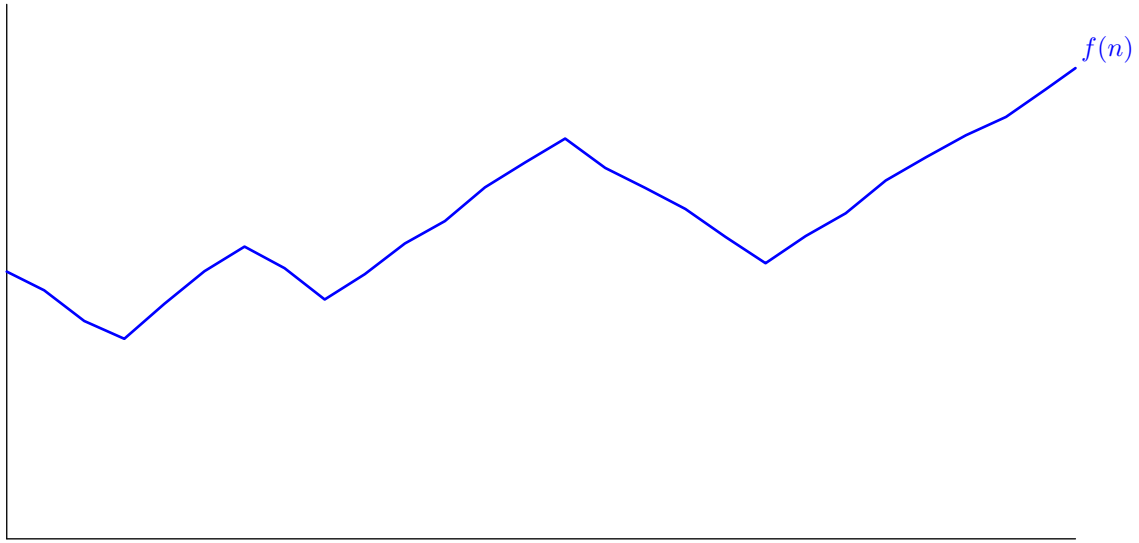
Local Search

If P$\neq$ NP, local search cannot find an optimal solution in any case.



Global optimum hard to find.

Local Search



In practice:

- Multiple starting points
- Randomized progress, accept worse solutions
- Simulated Annealing

Local neighborhood

Crucial:

Which solutions S' are in the neighborhood of a solution S?

- Add / remove parts of solution
- Exchange parts of solution

Local neighborhood

Crucial:

Which solutions S' are in the neighborhood of a solution S?

- Add / remove parts of solution
- Exchange parts of solution

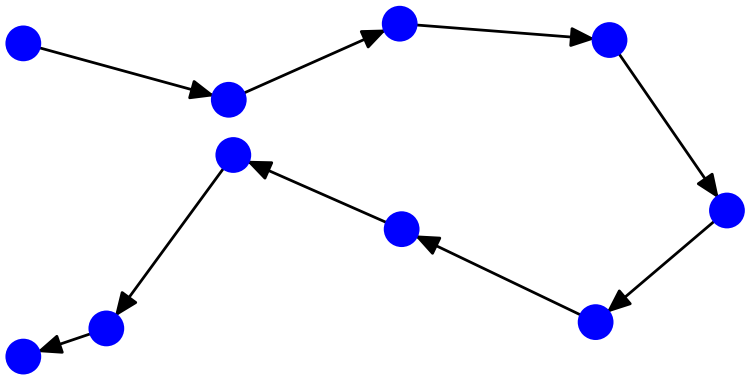
Example (Vertex Cover)

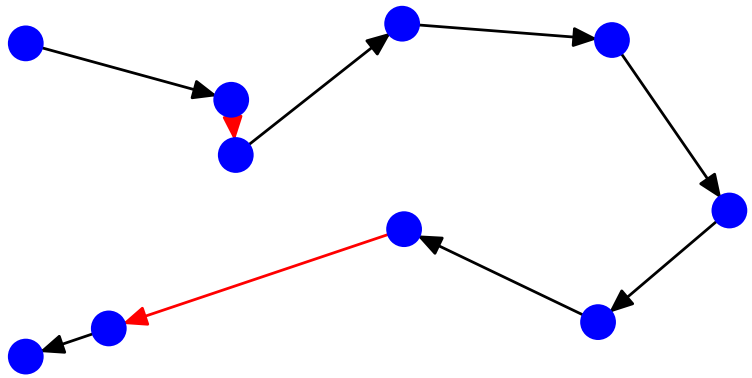
- Add nodes to cover
- remove nodes that are not needed
- Exchange node with all neighbors

Solution is always a vertex cover

If we do not obtain a valid solution in each step, how do we find a solution at all?

Traveling Salesman

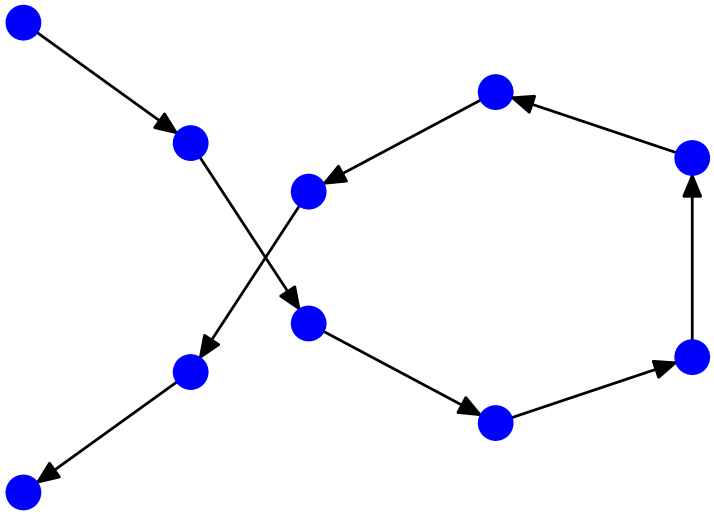


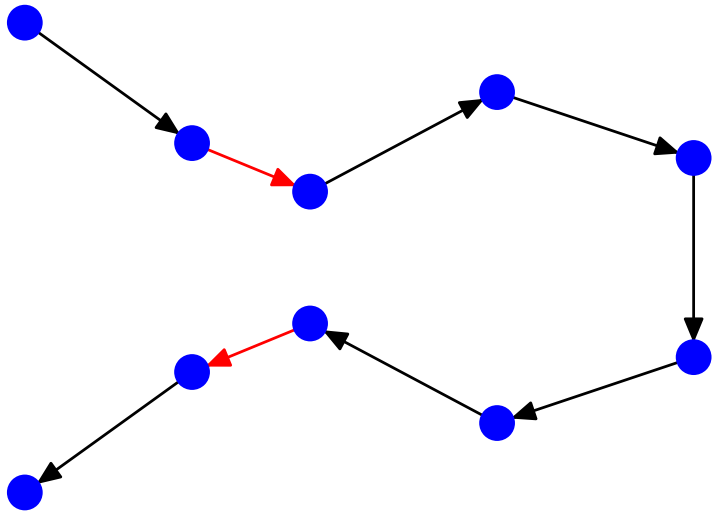


Move node to other position in cycle

- For each node v, try all new positions
- $O(n^2)$

Traveling Salesman





2-exchange

Exchange two edges

- For each pair of edges on the cycle one new possibility
- $O(n^2)$

k-exchange

- Exchange k edges
- $f(k)$ new solutions

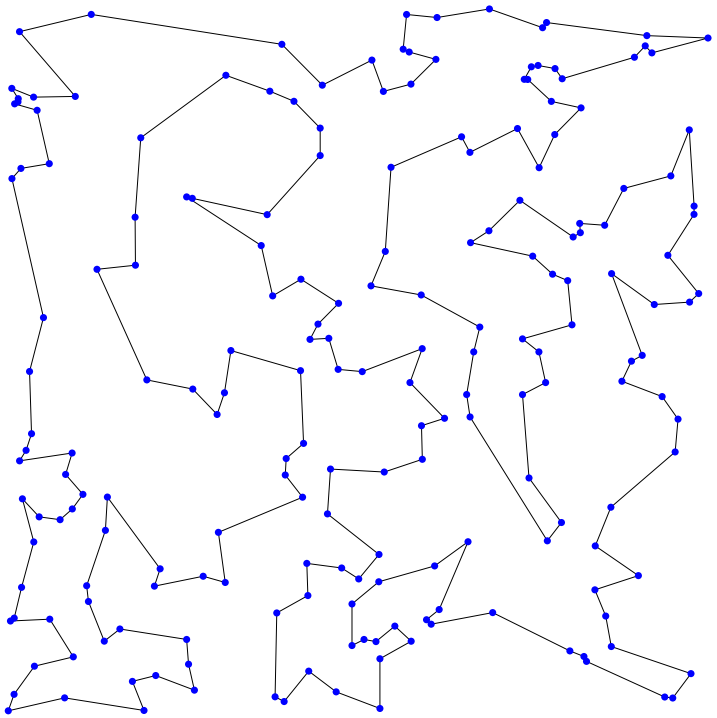
Local Search

Necessary:

Get from any solution S to S' in a series of local modifications

To find optimum:

- Accepting bad solutions in between
- How to choose modification?



Local Search and Exact Solutions

Theorem (Schöning)

MAX 3-SAT *can be solved in time* $O^*(1.334^n)$

Idea: local search

- Pick any assignment to the variables
- Pick any unsatisfied clause
- Flip variables

- At least on variable must be changed to satisfy clause
- Branching: 3^n? Something else?

Local Search and Exact Solutions

Theorem (Schöning)

MAX 3-SAT *can be solved in time* $O^*(1.334^n)$

Idea: local search

- Pick any assignment to the variables
- Pick any unsatisfied clause
- Flip variables

- At least on variable must be changed to satisfy clause
- Branching: 3^n? Something else?
- Randomized local search! (Random Walk)
- Possibility 1/3 for right direction

Random Walk for k-SAT

Random Walk(F)

```
Pick an assignment a;
count=0;
while (a is not satisfying && count < 3n)
   choose an unsatisfied clause c;
   choose on variable v ∈ c and flip v in a;
   count++;
return (a satisfies F);
```

Success Probability?

- Depends strongly on the initial a
- How to compute?

Analysis

Lemma

Algorithm Random Walk finds a satisfying assignment with probability of at least

$$\frac{c}{\sqrt{n}}\left(\frac{k}{2(k-1)}\right)^n$$

if such an assignment exists.

Idea:

Measure distance from optimal solution a^*

q_j : success probability if started on solution of distance j.

- distance decreases with probability at least $1/k$
- distance increases with probability at most $(k-1)/k$

Success Probability

Probability to find solution with l steps in wrong direction starting at distance j:

$$\binom{j+2l}{l}\left(\frac{k-1}{k}\right)^{l}\left(\frac{1}{k}\right)^{j+l}$$

Success Probability

Probability to find solution with l steps in wrong direction starting at distance j:

$$\binom{j+2l}{l}\left(\frac{k-1}{k}\right)^{l}\left(\frac{1}{k}\right)^{j+l}$$

$$\begin{aligned} q_j &\geq \max_{0\leq l\leq (3n-j)/2}\binom{j+2l}{l}\left(\frac{k-1}{k}\right)^{l}\left(\frac{1}{k}\right)^{j+l} \\ &\geq \max_{0\leq \alpha\leq 1}\binom{j+2\alpha j}{\alpha j}\left(\frac{k-1}{k}\right)^{\alpha j}\left(\frac{1}{k}\right)^{j+\alpha j} \\ &\geq \max_{0\leq \alpha\leq 1}\binom{(1+2\alpha)j}{\alpha j}\left(\left(\frac{k-1}{k}\right)^{\alpha}\left(\frac{1}{k}\right)^{1+\alpha}\right)^{j} \end{aligned}$$

Success Probability

$$
\begin{aligned}
q_j &\geq \max_{0 \leq l \leq (3n-j)/2} \binom{j+2l}{l} \left(\frac{k-1}{k}\right)^l \left(\frac{1}{k}\right)^{j+l} \\
&\geq \max_{0 \leq \alpha \leq 1} \binom{j+2\alpha j}{\alpha j} \left(\frac{k-1}{k}\right)^{\alpha j} \left(\frac{1}{k}\right)^{j+\alpha j} \\
&\geq \max_{0 \leq \alpha \leq 1} \binom{(1+2\alpha)j}{\alpha j} \left(\left(\frac{k-1}{k}\right)^{\alpha} \left(\frac{1}{k}\right)^{1+\alpha}\right)^j
\end{aligned}
$$

Lemma

$$
\binom{(1+2\alpha)j}{\alpha j} \geq \frac{1}{8\sqrt{j}} \left(\frac{1+2\alpha}{\alpha}\right)^{\alpha j} \left(\frac{1+2\alpha}{1+\alpha}\right)^{(1+\alpha)j}
$$

Proof of Lemma

$$
\begin{aligned}
\binom{(1+2\alpha)j}{\alpha j} &\geq \frac{((1+2\alpha)j)!}{(\alpha j)!((1+\alpha)j)!} \\
&\geq \frac{\sqrt{2\pi(1+2\alpha)j}}{2\sqrt{2\pi\alpha j}2\sqrt{2\pi(1+\alpha)j}}\left(\frac{(1+2\alpha)j}{e}\right)^{(1+2\alpha)j} \\
&\quad \left(\frac{e}{\alpha j}\right)^{\alpha j}\left(\frac{e}{(1+\alpha)j}\right)^{(1+\alpha)j} \\
&\geq \frac{\sqrt{1+2\alpha}}{8\sqrt{\alpha(1+\alpha)j}}\left(\frac{1+2\alpha}{1}\right)^{(1+2\alpha)j}\left(\frac{1}{\alpha}\right)^{\alpha j}\left(\frac{1}{1+\alpha}\right)^{(1+\alpha)j} \\
&\geq \frac{1}{8\sqrt{j}}\left(\frac{1+2\alpha}{\alpha}\right)^{\alpha j}\left(\frac{1+2\alpha}{1+\alpha}\right)^{(1+\alpha)j} \qquad \square
\end{aligned}
$$

$$
\begin{aligned}
q_j &\geq \max_{0\leq\alpha\leq 1} \frac{1}{8\sqrt{j}} \left(\left(\frac{1+2\alpha}{\alpha} \right)^{\alpha} \left(\frac{1+2\alpha}{1+\alpha} \right)^{(1+\alpha)} \left(\frac{k-1}{k} \right)^{\alpha} \left(\frac{1}{k} \right)^{1+\alpha} \right)^{j} \\
&\geq c \frac{1}{8\sqrt{j}} \left(\left(\frac{1+2\frac{1}{k-2}}{\frac{1}{k-2}} \right)^{(1/k-2)} \left(\frac{1+2\frac{1}{k-2}}{1+\frac{1}{k-2}} \right)^{(1+(1/k-2))} \right)^{j} \\
&\quad \left(\left(\frac{k-1}{k} \right)^{(1/k-2)} \left(\frac{1}{k} \right)^{1+(1/k-2)} \right)^{j} \\
&= \frac{1}{8\sqrt{j}} \left(\left(\frac{(k-2)+2}{1} \right)^{(1/k-2)} \left(\frac{(k-2)+2}{k-1} \right)^{(1+(1/k-2))} \right)^{j} \\
&\quad \left(\left(\frac{k-1}{k} \right)^{(1/k-2)} \left(\frac{1}{k} \right)^{1+(1/k-2)} \right)^{j}
\end{aligned}
$$

$$
\begin{aligned}
q_j &\geq \frac{1}{8\sqrt{j}} \left(\left(\frac{k}{1} \right)^{(1/k-2)} \left(\frac{k}{k-1} \right)^{(1+(1/k-2))} \right)^j \\
&\quad \left(\left(\frac{k-1}{k} \right)^{(1/k-2)} \left(\frac{1}{k} \right)^{1+(1/k-2)} \right)^j \\
&\geq \frac{1}{8\sqrt{j}} \left(\frac{k}{k-1} \frac{1}{k} \right)^j = \frac{1}{8\sqrt{j}} \left(\frac{1}{k-1} \right)^j
\end{aligned}
$$

The starting value

We know that $q_j \geq \frac{1}{8\sqrt{j}} \left(\frac{1}{k-1}\right)^j$.

But what is the probability p_j to start at distance j?

The starting value

We know that $q_j \geq \frac{1}{8\sqrt{j}} \left(\frac{1}{k-1}\right)^j$.

But what is the probability p_j to start at distance j?

$$p_j \geq \binom{n}{j} \left(\frac{1}{2}\right)^n$$

Random Walk

Overall success probability:

$$\sum_{j=0}^{n} p_j q_j$$

Overall Success Probability

$$
\begin{aligned}
\sum_{j=0}^{n} p_j q_j &\geq \sum_{j=0}^{n} \binom{n}{j} \left(\frac{1}{2}\right)^n \frac{1}{8\sqrt{j}} \left(\frac{1}{k-1}\right)^j \\
&= \frac{1}{8\sqrt{n}} \left(\frac{1}{2}\right)^n \sum_{j=0}^{n} \binom{n}{j} \left(\frac{1}{k-1}\right)^j \\
&= \frac{1}{8\sqrt{n}} \left(\frac{1}{2}\right)^n \sum_{j=0}^{n} \binom{n}{j} \left(\frac{1}{k-1}\right)^j 1^{n-j} \\
&= \frac{1}{8\sqrt{n}} \left(\frac{1}{2}\right)^n \left(1 + \frac{1}{k-1}\right)^n \\
&\geq \frac{1}{8\sqrt{n}} \left(\frac{k}{2(k-1)}\right)^n \quad \square
\end{aligned}
$$

Schönings Algorithm

Algorithm k-SAT(F)

$count = 0$;
while $(count < 8\sqrt{n}\left(\frac{2(k-1)}{k}\right)^n)$
  if (RandomWalk(F)) **return** ``**true**'';
  count++;
return ``no''

Random Walk(F)

Pick an assignment a;
$count = 0$;
while (a is **not** satisfying && $count < 3n$)
  choose an unsatisfied clause c;
  choose on variable $v \in c$ **and** flip v in a;
  $count$++;
return (a satisfies F);

Constant Success Probability

Theorem

Schönings Algorithm finds a satisfying assignment with probability at least $1/2$ *if such an assignment exists.*

$$\begin{aligned} & \left(1 - \frac{1}{8\sqrt{n}}\left(\frac{k}{2(k-1)}\right)^n\right)^{8\sqrt{n}\left(\frac{2(k-1)}{k}\right)^n} \\ = & \left(1 - \frac{1}{8\sqrt{n}\left(\frac{2(k-1)}{k}\right)^n}\right)^{8\sqrt{n}\left(\frac{2(k-1)}{k}\right)^n} \\ \leq & \; e^{-1} \leq 1/2 \end{aligned}$$

Exponentially small error : linear number of repetitions

Deterministic Algorithm

Definition

Let F be a formula in CNF with n variables $x_1, \ldots, x_n$. A (truth) assignment a of the variables $x_1, \ldots, x_n$ is $a \in \{0,1\}^n$. The

Definition

Let a, b be two truth assignments. The Hamming Distance $dist(a, b)$ is $|\{i \in \{1, \ldots, n\} \mid a_i \neq b_i\}|$.

Example

$$00011100111001$$
$$10010100101011$$

$$dist(00011100111001, 10010100101011) = 4$$

Deterministic Algorithm

Lemma

Let F be Formula in k-CNF and a a variable assignment for F. We can decide in $O^*(k^r)$ if there is an assignment a^* that satisfies F with $dist(a, a^*) \leq r$.

- Pick an unsatisfied clause C in F
- Recursively flip all k variables in C

Lemma

k-SAT *can be solved in $O^*(k^n)$.*

Deterministic Algorithm

Lemma

Let F be Formula in k-CNF and a a variable assignment for F. We can decide in $O^*(k^r)$ if there is an assignment a^* that satisfies F with $dist(a, a^*) \leq r$.

- Pick an unsatisfied clause C in F
- Recursively flip all k variables in C

Lemma

k-SAT *can be solved in $O^*(k^{n/2})$.*

Deterministic Algorithm

Lemma

Let F be Formula in k-CNF and a a variable assignment for F. We can decide in $O^*(k^r)$ if there is an assignment a^* that satisfies F with $dist(a, a^*) \leq r$.

- Pick an unsatisfied clause C in F
- Recursively flip all k variables in C

Lemma

k-SAT *can be solved in $O^*(k^{n/2})$.*

Start with 0^n and 1^n.

3-SAT can be solved in $O^*(1.733^n)$.

Smaller Neighborhoods

Definition

Let a be a truth assignment. $N(a, d) = \{a' \mid dist(a, a') \leq d\}$

Idea:

- Pick several assignments
- For each assignment a, only search $N(a, \alpha n)$

Lemma

Let F be a formula in k-CNF and $a*$ an assignment that satisfies F. The probability that $a* \in N(a, \alpha n)$ for some random a is at least

$$\frac{\sum_{i=0}^{\alpha n} \binom{n}{i}}{2^n}$$

- a differs in at most αn positions from a^*

A partly randomized algorithm

- Pick $\frac{2^n}{\sum_{i=0}^{\alpha n}\binom{n}{i}}$ assignments
- Check for each assignment a in $k^{\alpha n}$ whether there is satisfying assignment in $N(a, \alpha n)$

For $0 < \alpha \leq 1/2$:

$$\frac{1}{\sqrt{8n\alpha(1-\alpha)}}2^{h(\alpha)n} \leq \sum_{i=0}^{\alpha n}\binom{n}{i} \leq 2^{h(\alpha)n} = \left(\frac{1}{\alpha}\right)^{\alpha n}\left(\frac{1}{1-\alpha}\right)^{(1-\alpha)n}$$

where the binary entropy function h is defined as

$$h(\alpha) = -\alpha\log_2(\alpha) - (1-\alpha)\log_2(1-\alpha).$$

Runtime

$$O^*(2^{(1-h(\alpha))n}k^{\alpha n})$$

A partly randomized algorithm

Let $\alpha = 1/(k+1)$

$$
\begin{aligned}
2^{(1-h(\alpha))n} k^{\alpha n} &= \frac{2^n k^{\alpha n}}{\left(\frac{1}{\alpha}\right)^{\alpha n} \left(\frac{1}{1-\alpha}\right)^{(1-\alpha)n}} \\
&= \frac{2^n k^{1/(k+1)n}}{\left(\frac{1}{1/(k+1)}\right)^{1/(k+1)n} \left(\frac{1}{1-1/(k+1)}\right)^{(1-1/(k+1))n}} \\
&= \frac{2^n k^{1/(k+1)n}}{(k+1)^{1/(k+1)n} \left(\frac{k+1}{k}\right)^{(1-1/(k+1))n}} \\
&= \frac{2^n k^{1/(k+1)n} k^{(1-1/(k+1))n}}{(k+1)^{1/(k+1)n} (k+1)^{(1-1/(k+1))n}} \\
&= \left(\frac{2k}{k+1}\right)^n
\end{aligned}
$$

Why is this better?

New algorithm:

- Runtime $\left(\frac{2k}{k+1}\right)^n$ instead of $\left(\frac{2(k-1)}{k}\right)^n$
- Still randomized
- Only randomized starting positions
- No random walk anymore

Why is this better?

New algorithm:

- Runtime $\left(\frac{2k}{k+1}\right)^n$ instead of $\left(\frac{2(k-1)}{k}\right)^n$
- Still randomized
- Only randomized starting positions
- No random walk anymore

Need to find starting positions to cover solution space!

Definition

Let F be a formula in CNF with n variables. A set $C \subseteq \{0,1\}^n$ is called *covering code* of radius d if for each assignment $a*$ that satisfies F, there is a $c \in C$ with $dist(a^*, c) \leq d$.

Construction C

Overview:

- Show that a solution C_{opt} of size $O^*(2^{(1-h(\alpha))n})$ can be found $(0 < \alpha \leq 1/2)$.
- Construct solution C that is of size at most $n|C_{opt}|$.

Construction C

Overview:

- Show that a solution C_{opt} of size $O^*(2^{(1-h(\alpha))n})$ can be found $(0 < \alpha \leq 1/2)$.
- Construct solution C that is of size at most $n|C_{opt}|$.
- Unfortunately construction of C is too slow.
- Use C for n/l and combine several C.

Existence of a Small Covering Code

Lemma

Let $0 < \alpha < 1/2$. There exists a covering code C of radius αn with $|C| \leq n\sqrt{n\alpha(1-\alpha)}2^{(1-h(\alpha))n}$

Choose randomly

$$\left\lceil \frac{n2^n}{\sum_{i=0}^{\alpha n} \binom{n}{i}} \right\rceil$$

assignments. Probability for a single neighborhood to not include a^*:

$$1 - \frac{\sum_{i=0}^{\alpha n} \binom{n}{i}}{2^n}.$$

Overall:

$$\left(1 - \frac{\sum_{i=0}^{\alpha n} \binom{n}{i}}{2^n}\right)^{\left\lceil \frac{n2^n}{\sum_{i=0}^{\alpha n} \binom{n}{i}} \right\rceil} \leq e^{-n}.$$

Existence of a Small Covering Code

For all assignments:

$$e^{-n}2^n \leq \frac{2}{e} < 1$$

We find a covering code of size

$$\left\lceil \frac{n2^n}{\sum_{i=0}^{\alpha n} \binom{n}{i}} \right\rceil$$

with positive probability. Hence, such a code exists.
Moreover

$$\left\lceil \frac{n2^n}{\sum_{i=0}^{\alpha n} \binom{n}{i}} \right\rceil \leq n\sqrt{n\alpha(1-\alpha)}2^{(1-h(\alpha))n}.$$

Constructing C

Lemma

A covering code of radius αn and size $n^2\sqrt{n\alpha(1-\alpha)}2^{(1-h(\alpha))n}$ can be computed in time $O^*(8^n)$.

- A covering code is a special set cover with sets $N(a, d)$
- Greedy algorithm for Set Cover yields a $log(m)$-approximation, where m denotes the number of sets.

Greedy finds a solution of size $\log(2^n)|Opt| = n^2\sqrt{n\alpha(1-\alpha)}2^{(1-h(\alpha))n}$.
In each step:

- Go through all sets and find largest set S
- Go through all sets S' and remove S from S'

There are 2^n sets, $|S| \leq 2^n$, $|S'| \leq 2^n$, $\Rightarrow$ runtime bound $O^*(8^n)$.

Smaller Codes

Lemma

A covering code of radius αn and size $\left(n^2\sqrt{n\alpha(1-\alpha)}\right)^l 2^{(1-h(\alpha))n}$ can be computed in time $O^*(2^{3n/l} + 2^{(1-h(\alpha))n})$.

- Construct code C' for length n/l of radiues $\alpha n/l$.
  Size $|C'| \leq n^2\sqrt{n\alpha(1-\alpha)}2^{\frac{(1-h(\alpha))n}{l}}$ in time $O^*(2^{3n/l})$.
- Combine pieces of C' to obtain C: Runtime $|C'|^l$.

Overall runtime:

$$O^*\left(2^{3n/l} + \left(n^2\sqrt{n\alpha(1-\alpha)}\right)^l 2^{\frac{(1-h(\alpha))nl}{l}}\right)$$

Size:

$$\left(n^2\sqrt{n\alpha(1-\alpha)}\right)^l 2^{(1-h(\alpha))n}$$

Deterministic Algorithm

- Choose $l = 6$, $\alpha = 1/(k+1)$ (like the randomized version)
- Construct Code C
- For each $cin C$, search $N(c, \alpha n)$.

Runtime: Constructing the code:

$$O^*\left(2^{3n/l} + \left(n^2\sqrt{n\alpha(1-\alpha)}\right)^l 2^{\frac{(1-h(\alpha))nl}{l}}\right)$$

Search for solution:

$$\left(n^2\sqrt{n\alpha(1-\alpha)}\right)^l 2^{(1-h(\alpha))n} k^{\frac{n}{k+1}}$$

Overall (recall bound in randomized case):

$$O^*\left(\left(\frac{2k}{k+1}\right)^n\right)$$

Overview

1. Introduction
2. Branching
3. Measure&Conquer
4. Dynamic Programming
5. Local Search
6. Split & List
7. Lower Bounds
8. Inclusion-exclusion
9. Subset Convolution

Split & List

Example

Input: Lists $A, B \subseteq \mathbf{N}$, $s \in \mathbf{N}$
Question: Exists $a \in A, b \in B$ with $a + b = s$?

- Trivial: $O(|A||B|)$

Split & List

Example

Input: Lists $A, B \subseteq \mathbf{N}$, $s \in \mathbf{N}$
Question: Exists $a \in A, b \in B$ with $a + b = s$?

- Trivial: $O(|A||B|)$
- $O(|A| \log |A| + |B| \log |B|)$?

Algorithm Sum(A,B,s)

```
sort(A);
sort(B);
i=1, j=|B|;
while (i<=|A| && j >=1)
   if (A[i]+B[j]>s ) j--;
   else if (A[i]+B[j]<s ) i++;
   else return ``yes'';
return ``no''
```

Example

Example

$$
\begin{array}{cccc}
\Rightarrow & 2 & 2 & \\
 & 3 & 3 & \\
 & 4 & 5 & \\
 & 5 & 6 & \\
 & 8 & 9 & \\
 & 11 & 12 & \Leftarrow
\end{array}
$$

$$s = 10$$

Example

Example

$$
\begin{array}{cccc}
\Rightarrow & 2 & 2 & \\
 & 3 & 3 & \\
 & 4 & 5 & \\
 & 5 & 6 & \\
 & 8 & 9 & \Leftarrow \\
 & 11 & 12 &
\end{array}
$$

$$s = 10$$

Example

Example

$$
\begin{array}{cccc}
 & 2 & 2 & \\
\Rightarrow & 3 & 3 & \\
 & 4 & 5 & \\
 & 5 & 6 & \Leftarrow \\
 & 8 & 9 & \\
 & 11 & 12 &
\end{array}
$$

$$s = 10$$

Example

Example

$$
\begin{array}{cccc}
 & 2 & 2 & \\
\Rightarrow & 3 & 3 & \\
 & 4 & 5 & \\
 & 5 & 6 & \Leftarrow \\
 & 8 & 9 & \\
 & 11 & 12 &
\end{array}
$$

$$s = 10$$

Example

Example

$$
\begin{array}{cccc}
 & 2 & 2 & \\
 & 3 & 3 & \\
\Rightarrow & 4 & 5 & \\
 & 5 & 6 & \Leftarrow \\
 & 8 & 9 & \\
 & 11 & 12 &
\end{array}
$$

$$s = 10$$

Correctness

The algorithm never outputs a wrong solution.

Obviously, only correct solutions are found.

If a solution exists, the algorithm finds it.

Let $a' + b' = s$ be a solution. We show that a', b' is never disregarded.
By Induction: $a[i] \leq a'$ and $b[j] \geq b'$.

- $a[1] \leq a[i]$ and $b[n] \geq b[j]$ for all i, j
- If $a[i] + b[j] > s$, we obtain $a' + b[j] > s$.
  Hence $b' \leq b[j-1]$
- If $a[i] + b[j] < s$, we obtain $a[i] + b' < s$.
  Hence $a' \geq a[i+1]$

Subset Sum

Definition

The Subset Sum problem is defined as follows:

Input: $S \subset \mathbf{N}$, $s \in \mathbf{N}$

Question: Exists $S' \subseteq S$, $\sum i \in S' i = s$?

Subset Sum can be solved in time $O^*(2^{|S|})$.

Subset Sum

Definition

The Subset Sum problem is defined as follows:

Input: $S \subset \mathbf{N}$, $s \in \mathbf{N}$

Question: Exists $S' \subseteq S$, $\sum i \in S' i = s$?

Subset Sum can be solved in time $O^*(2^{|S|})$.

Lemma

Subset Sum *can be solved in time* $O^*(2^{|S|/2})$.

Subset Sum

Wlog assume $|S|$ is even.

Subset Sum

Partition S into S_1, S_2 with $|S_1| = |S_2|$
Let $L_i = \{\sum_{s \in S'} s \mid S' \subseteq S_i\}$
return Sum(L_1, L_2, s)

Wlog assume $|S|$ is even.

Subset Sum

Partition S into S_1, S_2 with $|S_1| = |S_2|$
Let $L_i = \{\sum_{s \in S'} s \mid S' \subseteq S_i\}$
return Sum(L_1, L_2, s)

Correctness

Let $S' = \{s_1, \ldots, s_k\}$ be a solution.

- If $\sum_{i=1}^{k} s_i = s$, then $\sum_{i=1}^{\lfloor k/2 \rfloor} s_i + \sum_{i=\lceil k/2 \rceil}^{k} s_i = s$
- Holds for any permutation of S'
- Algorithm *Sum* is correct

Wlog assume $|S|$ is even.

Subset Sum

Partition S into S_1, S_2 with $|S_1| = |S_2|$
Let $L_i = \{\sum_{s \in S'} s \mid S' \subseteq S_i\}$
return Sum(L_1, L_2, s)

Running Time

- Partitioning: $O(|S|)$
- Constructing L_i: $O^*(2^{|S|/2})$
- Computing sum: $O(\log(2^{|S|/2}) + 2^{|S|/2})$

Definition

The Problem EXACT SAT (XSAT) is defined as follows:

Input:	A formula F in CNF
Question:	Is there a truth assignment that satisfies exactly one literal in each clause of F

Example

$$F = \{x_1, \overline{x_2}, x_3\}, \{\overline{x_2}, \overline{x_4}\}, \{\overline{x_1}, x_3\}, \{\overline{x_2}, x_3, \overline{x_4}\}$$

- $x_1 = x_3 = 1$, $x_4 = 0$ satisfies F, but $\{x_1, \overline{x_2}, x_3\}$ is not exactly satisfied
- $x_1 = 0$, $x_2 = 0$, $x_3 = 0$, $x_4 = 1$ exactly satisfies each clause

XSAT

Lemma

XSAT *can be solved in time* $O^*(1.45^n)$ *where* n *denotes the number of variables.*

Proof: Branching, exercise!

Lemma

XSAT *can be solved in time* $O^*(2^{n/2})$ *with split&list.*

XSAT

Lemma

XSAT *can be solved in time* $O^*(1.45^n)$ *where* n *denotes the number of variables.*

Proof: Branching, exercise!

Lemma

XSAT *can be solved in time* $O^*(2^{n/2})$ *with split&list.*

- Subset Sum: sorted lists of integers

XSAT

Lemma

XSAT *can be solved in time* $O^*(1.45^n)$ *where* n *denotes the number of variables.*

Proof: Branching, exercise!

Lemma

XSAT *can be solved in time* $O^*(2^{n/2})$ *with split&list.*

- Subset Sum: sorted lists of integers
- Method relies only on order on solutions

XSAT

Definition

Let F be a formula in CNF and X' a subset of the variables. For an assignment a of X' let $f(a, X') \in \mathbb{N}^m$ be the vector such that the ith component $f(a, X')[i]$ counts how many literals in clause c_i are satisfied.

Example

$$F = \{x_1, \overline{x_2}, x_3\}, \{\overline{x_2}, \overline{x_4}\}, \{\overline{x_1}, x_3\}, \{\overline{x_2}, x_3, \overline{x_4}\}$$

Let $X = \{x_1, x_2, x_3\}$ and $a := x_1 = x_3 = 1,\ x_2 = 0$.

$$f(a, X) = (3, 1, 1, 2)$$

We can order these vectors lexicographically.

Comparing Vectors

Idea:

- Partition X into two sets X, Y
- Compute ordered list of all $f(a, X)$ and $f(a, Y)$
- Test whether $f(a', X) + f(b', X) = (1, 1, \ldots, 1)$

Comparing Vectors

Idea:

- Partition X into two sets X, Y
- Compute ordered list of all $f(a, X)$ and $f(a, Y)$
- Test whether $f(a', X) + f(b', X) = (1, 1, \ldots, 1)$

Why does it work?

$$(1, 3, 0, 2) + (5, 2, 1, 4) > (a, b, c, d)$$

then

$$(1, 3, 0, 2) + (5, 3, 1, 4) > (a, b, c, d)$$

In general:

$a, b, s, b' \in \mathbf{N}^m$.

$a + b > s$ implies $a + b' > s$ for all $b' > b$.

Example

Example

$$F = \{x_1, \overline{x_2}, x_3\}, \{\overline{x_2}, \overline{x_4}\}, \{\overline{x_1}, x_3\}, \{\overline{x_2}, x_3, \overline{x_4}\}$$

x_1, x_2	$f(a, \{x_1, x_2\})$	x_3, x_4	$f(a, \{x_3, x_4\})$
00	1111	00	0101
01	0010	01	0000
10	2101	10	1112
11	1000	11	1011

Example

Example

$$F = \{x_1, \overline{x_2}, x_3\}, \{\overline{x_2}, \overline{x_4}\}, \{\overline{x_1}, x_3\}, \{\overline{x_2}, x_3, \overline{x_4}\}$$

$f(a, \{x_1, x_2\})$	x_1, x_2	x_3, x_4	$f(a, \{x_3, x_4\})$
0010	01	01	0000
1000	11	00	0101
1111	00	11	1011
2101	10	10	1112

Example

Example

$$F = \{x_1, \overline{x_2}, x_3\}, \{\overline{x_2}, \overline{x_4}\}, \{\overline{x_1}, x_3\}, \{\overline{x_2}, x_3, \overline{x_4}\}$$

	$f(a, \{x_1, x_2\})$	x_1, x_2	x_3, x_4	$f(a, \{x_3, x_4\})$	
$\Rightarrow$	0010	01	01	0000	
	1000	11	00	0101	
	1111	00	11	1011	
	2101	10	10	1112	$\Leftarrow$

Example

Example

$$F = \{x_1, \overline{x_2}, x_3\}, \{\overline{x_2}, \overline{x_4}\}, \{\overline{x_1}, x_3\}, \{\overline{x_2}, x_3, \overline{x_4}\}$$

	$f(a, \{x_1, x_2\})$	x_1, x_2	x_3, x_4	$f(a, \{x_3, x_4\})$	
$\Rightarrow$	0010	01	01	0000	
	1000	11	00	0101	
	1111	00	11	1011	$\Leftarrow$
	2101	10	10	1112	

Example

Example

$$F = \{x_1, \overline{x_2}, x_3\}, \{\overline{x_2}, \overline{x_4}\}, \{\overline{x_1}, x_3\}, \{\overline{x_2}, x_3, \overline{x_4}\}$$

	$f(a, \{x_1, x_2\})$	x_1, x_2	x_3, x_4	$f(a, \{x_3, x_4\})$	
$\Rightarrow$	0010	01	01	0000	
	1000	11	00	0101	$\Leftarrow$
	1111	00	11	1011	
	2101	10	10	1112	

Example

Example

$$F = \{x_1, \overline{x_2}, x_3\}, \{\overline{x_2}, \overline{x_4}\}, \{\overline{x_1}, x_3\}, \{\overline{x_2}, x_3, \overline{x_4}\}$$

	$f(a, \{x_1, x_2\})$	x_1, x_2	x_3, x_4	$f(a, \{x_3, x_4\})$	
	0010	01	01	0000	
$\Rightarrow$	1000	11	00	0101	$\Leftarrow$
	1111	00	11	1011	
	2101	10	10	1112	

Example

Example

$$F = \{x_1, \overline{x_2}, x_3\}, \{\overline{x_2}, \overline{x_4}\}, \{\overline{x_1}, x_3\}, \{\overline{x_2}, x_3, \overline{x_4}\}$$

	$f(a, \{x_1, x_2\})$	x_1, x_2	x_3, x_4	$f(a, \{x_3, x_4\})$	
	0010	01	01	0000	
	1000	11	00	0101	$\Leftarrow$
$\Rightarrow$	1111	00	11	1011	
	2101	10	10	1112	

Example

Example

$$F = \{x_1, \overline{x_2}, x_3\}, \{\overline{x_2}, \overline{x_4}\}, \{\overline{x_1}, x_3\}, \{\overline{x_2}, x_3, \overline{x_4}\}$$

	$f(a, \{x_1, x_2\})$	x_1, x_2	x_3, x_4	$f(a, \{x_3, x_4\})$	
	0010	01	01	0000	$\Leftarrow$
	1000	11	00	0101	
$\Rightarrow$	1111	00	11	1011	
	2101	10	10	1112	

Algorithm Sum(A,B,s)

A,B: list of vectors.

```
sort(A);
sort(B);
i=1, j=|B|;
while (i<=|A| && j >=1)
   if (A[i]+B[j]>s ) j--;
   else if (A[i]+B[j]<s ) i++;
   else return ``yes'';
return ``no''
```

Lemma

XSAT *can be solved in time* $O^*(2^{n/2})$

Proof:

- Partition set of variables into X, Y
- F is satisfied if there exists a such that $f(a, X \cup Y) = (1, \ldots, 1)$
- $f(a, X \cup Y) = (1, \ldots, 1) \Leftrightarrow f(a, X) + f(a, Y) = (1, \ldots, 1)$
- Algorithm *Sum* finds solution for $f(a, X) + f(a, Y) = (1, \ldots, 1)$

SAT?

Why can't we solve SAT with this method?

SAT?

Why can't we solve SAT with this method?

- We need $f(a, X) + f(a, Y) \geq (1, \ldots, 1)$
- Consider $(1, 0, 0, 0) + (1, 1, 1, 0) = (2, 1, 1, 0) > (1, 1, 1, 1)$

Where is the difference between $a + b \geq s$ and $a + b = s$?

SAT?

Why can't we solve SAT with this method?

- We need $f(a, X) + f(a, Y) \geq (1, \ldots, 1)$
- Consider $(1, 0, 0, 0) + (1, 1, 1, 0) = (2, 1, 1, 0) > (1, 1, 1, 1)$

Where is the difference between $a + b \geq s$ and $a + b = s$?

- If we want $a + (1, 1, 1, 0) = (1, 1, 1, 1)$, we can ignore all $a \geq (1, 0, 0, 0)$
- But $(1, 0, 0, 1) \geq (1, 0, 0, 0)$ and $(1, 0, 0, 1) + (1, 1, 1, 0) \geq (1, 1, 1, 1)$

If $a + b = c$ and c is not a solution, there might exist $a' + b' = c' > c$ such that c' is a solution.

Knapsack

Definition

The problem Knapsack is defined as follows:

Input:	Pairs $(a_i, w_i) \in \mathbb{N}^2$ $1 \leq i \leq n$ and $A, W \in \mathbb{N}^2$
Question:	Exists $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} a_i \geq A$ and $\sum_{i \in S} w_i \leq W$

Idea: Pack items with gain at least A but weight at most W.

Example

$$(1, 3), (2, 5), (3, 2), (1, 1)$$

Goal: $(5, 7)$.

- We want to apply split&list
- $\sum_{i \in S} a_i \geq A$ and $\sum_{i \in S} w_i \leq W$?

Idea

- Compute all solutions for $S_1 \subseteq \{(a_i, w_i) \mid i \leq \lfloor n/2 \rfloor\}$ and $S_2 \subseteq \{(a_i, w_i) \mid i \geq \lceil n/2 \rceil\}$
- Apply split&list

Knapsack

- We want to apply split&list
- $\sum_{i \in S} a_i \geq A$ and $\sum_{i \in S} w_i \leq W$?

Idea

- Compute all solutions for $S_1 \subseteq \{(a_i, w_i) \mid i \leq \lfloor n/2 \rfloor\}$ and $S_2 \subseteq \{(a_i, w_i) \mid i \geq \lceil n/2 \rceil\}$
- Preprocessing: For each w_j only one solution
- Apply split&list

Preprocessing

Assume we gain $(3, 5)$ and $(4, 5)$ for different subsets. Can we ignore one of those?

Exchange argument:

Let $(A_i, W_i) + (B, W)$ be a solution. Then for each A_j, W_j with $A_j \geq A_i$ and $W_j \leq W_j$, $(A_j, W_j) + (B, W)$ is also a solution.

Preprocessing

Assume we gain $(3, 5)$ and $(4, 5)$ for different subsets. Can we ignore one of those?

Exchange argument:

Let $(A_i, W_i) + (B, W)$ be a solution. Then for each A_j, W_j with $A_j \geq A_i$ and $W_j \leq W_j$, $(A_j, W_j) + (B, W)$ is also a solution.

Why does it help?

- On each side only optimal value for each W_i
- Combine W_i with largest possible W_j

Preprocessing might expensive!

Let $(A_i, W_i) < (A_j, W_j)$ if $W_i < W_j$ or $W_i = W_j$ and $A_i > A_j$

Preprocessing 1 ($S = \{A_i, W_i)\}$

```
Sort S according to W
for i=2 to |S|
   if (W_{i-1} = W_i) remove (A_i, W_i)
```

Let $(A_i, W_i) < (A_j, W_j)$ if $W_i < W_j$ or $W_i = W_j$ and $A_i > A_j$

Preprocessing 1 ($S = \{A_i, W_i)\}$

```
Sort S according to W
for i=2 to |S|
   if (W_{i-1} = W_i) remove (A_i, W_i)
```

Lemma

After Preprocessing 1, S contains only one (A_i, W) for each W and A_i is maximum among those.

Proof:

- If there are several (A_i, W) in S, they occur in a row after sorting
- All but the first one are removed
- Since $(A_1, W) < (A_2, W)$ if $A_1 > A_2$, the largest A is the first one and hence not removed.

Preprocessing 2 ($S = \{A_i, W_i)\}$

```
Sort S according to W
max = A_1
for i = 2 to |S|
    if (A_i < max) remove (A_i, W_i)
    else max = A_i ;
```

Lemma

After Preprocessing 2, S contains no two pair such that $A_i > A_j$ and $W_i < W_j$.

Proof:

- S is sorted according to W
- *max* stores the maximal value achievable with weight at most W_i

Knapsack

Input: (A_i, W_i) for $1 \leq i \leq n$. A, W.

Compute $S_1 = \{(A_I, W_I) \mid I \subseteq \{1 \leq i \leq \lfloor n/2 \rfloor\}\}$;
Compute $S_2 = \{(A_I, W_I) \mid I \subseteq \{\lceil n/2 \rceil \leq i \leq n\}\}$;
sort(S_1); sort(S_2);
Apply preprocessing to S_1 **and** S_2;
Find **for** each $(A_I, W_I) \in S_1$
the largest compatible $(A_J, W_J) \in S_2$;
return optimum of these;

Knapsack

Input: (A_i, W_i) for $1 \leq i \leq n$. A, W.

Compute $S_1 = \{(A_I, W_I) \mid I \subseteq \{1 \leq i \leq \lfloor n/2 \rfloor\}\}$;
Compute $S_2 = \{(A_I, W_I) \mid I \subseteq \{\lceil n/2 \rceil \leq i \leq n\}\}$;
sort(S_1); sort(S_2);
Apply preprocessing to S_1 **and** S_2;
Find **for** each $(A_I, W_I) \in S_1$
the largest compatible $(A_J, W_J) \in S_2$;
return optimum of these;

Runtime:

- Sorting S_1, S_2: $O^*(2^{n/2})$
- Split&List

Split & List Revisited

Example

$$\begin{array}{lll} \Rightarrow & 2 & 2 \\ & 3 & 3 \\ & 4 & 5 \\ & 5 & 6 \\ & 8 & 9 \\ & 11 & 12 \quad \Leftarrow \end{array}$$

$$W = 10$$

Just compute the corresponding $A_I + A_J$ in each step!

Split & List Revisited

Example

$$
\begin{array}{lrll}
\Rightarrow & 2 & 2 & \\
 & 3 & 3 & \\
 & 4 & 5 & \\
 & 5 & 6 & \\
 & 8 & 9 & \Leftarrow \\
 & 11 & 12 &
\end{array}
$$

$$W = 10$$

Just compute the corresponding $A_I + A_J$ in each step!

Split & List Revisited

Example

$$
\begin{array}{lll}
\Rightarrow & 2 & 2 \\
 & 3 & 3 \\
 & 4 & 5 \\
 & 5 & 6 \quad \Leftarrow \\
 & 8 & 9 \\
 & 11 & 12
\end{array}
$$

$$W = 10$$

Solution for 2

Just compute the corresponding $A_I + A_J$ in each step!

Split & List Revisited

Example

$$\begin{array}{cccc} & 2 & 2 & \\ \Rightarrow & 3 & 3 & \\ & 4 & 5 & \\ & 5 & 6 & \Leftarrow \\ & 8 & 9 & \\ & 11 & 12 & \end{array}$$

$$W = 10$$

Solution for 3

Just compute the corresponding $A_I + A_J$ in each step!

Split & List Revisited

Example

$$\begin{array}{cccc} & 2 & 2 & \\ & 3 & 3 & \\ \Rightarrow & 4 & 5 & \\ & 5 & 6 & \Leftarrow \\ & 8 & 9 & \\ & 11 & 12 & \end{array}$$

$$W = 10$$

Solution for 4

Just compute the corresponding $A_I + A_J$ in each step!

Split & List Revisited

Example

$$\begin{array}{ccc} & 2 & 2 \\ & 3 & 3 \\ & 4 & 5 \\ \Rightarrow & 5 & 6 \\ & 8 & 9 \\ & 11 & 12 \end{array} \Leftarrow$$

$$W = 10$$

Just compute the corresponding $A_I + A_J$ in each step!

Split & List Revisited

Example

$$\begin{array}{cccc} & 2 & 2 & \\ & 3 & 3 & \\ & 4 & 5 & \Leftarrow \\ \Rightarrow & 5 & 6 & \\ & 8 & 9 & \\ & 11 & 12 & \end{array}$$

$$W = 10$$

Solution for 5

Just compute the corresponding $A_I + A_J$ in each step!

Split & List Revisited

Example

$$\begin{array}{cccc} & 2 & 2 & \\ & 3 & 3 & \\ & 4 & 5 & \Leftarrow \\ & 5 & 6 & \\ \Rightarrow & 8 & 9 & \\ & 11 & 12 & \end{array}$$

$$W = 10$$

Just compute the corresponding $A_I + A_J$ in each step!

Split & List Revisited

Example

$$
\begin{array}{cccc}
 & 2 & 2 & \\
 & 3 & 3 & \Leftarrow \\
 & 4 & 5 & \\
 & 5 & 6 & \\
\Rightarrow & 8 & 9 & \\
 & 11 & 12 &
\end{array}
$$

$$W = 10$$

Just compute the corresponding $A_I + A_J$ in each step!

Split & List Revisited

Example

$$\begin{array}{cll} & 2 & 2 & \Leftarrow \\ & 3 & 3 & \\ & 4 & 5 & \\ & 5 & 6 & \\ \Rightarrow & 8 & 9 & \\ & 11 & 12 & \end{array}$$

$$W = 10$$

Solution for 8

Just compute the corresponding $A_I + A_J$ in each step!

Split & List Revisited

Example

$$\begin{array}{cccc} & 2 & 2 & \Leftarrow \\ & 3 & 3 & \\ & 4 & 5 & \\ & 5 & 6 & \\ & 8 & 9 & \\ \Rightarrow & 11 & 12 & \end{array}$$

$$W = 10$$

Just compute the corresponding $A_I + A_J$ in each step!

Theorem

Knapsack *can be solved in time* $O^*(2^{n/2})$.

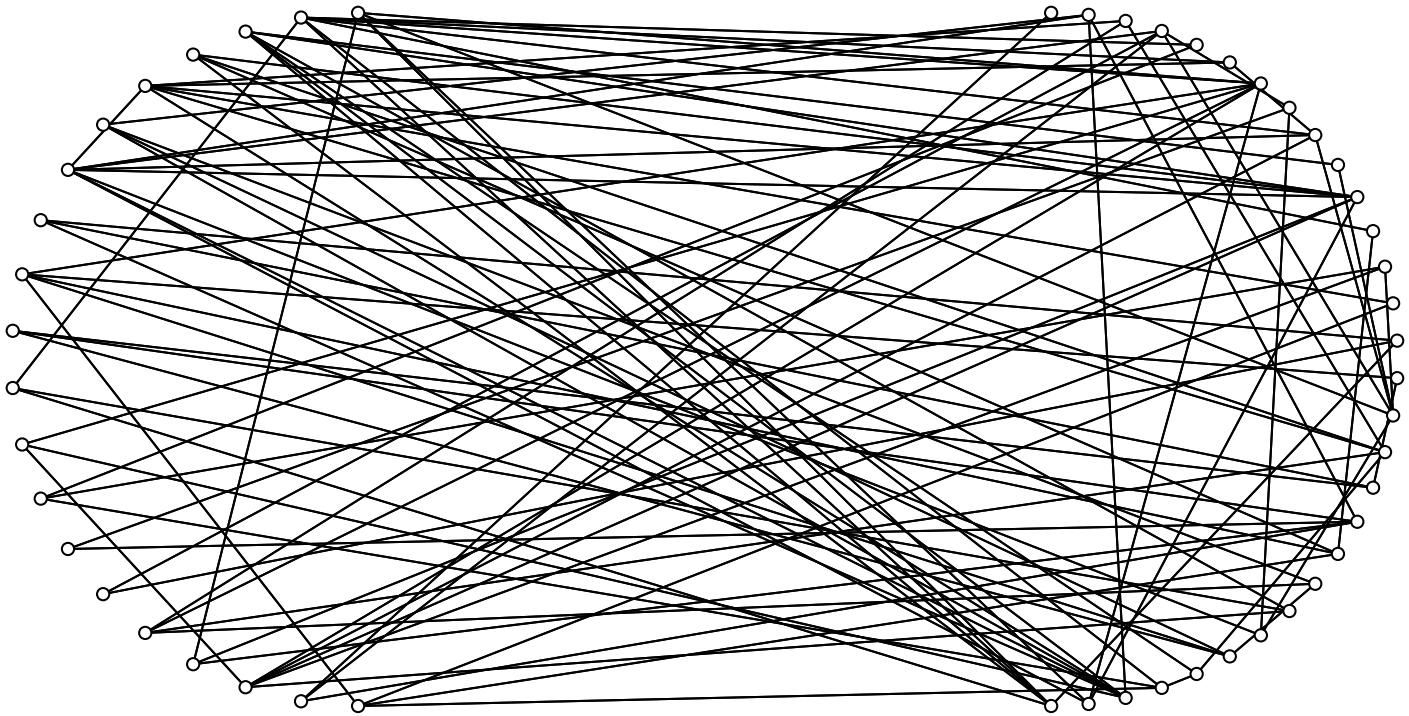
Proof:

- $2^{n/2}$ elements in S_1 and in S_2
- Sorting takes $O(|S_i|2^{|S_i|})$
- Preprocessing takes $O(2^{|S_i|})$
- Split& List takes $O(2^{|S_i|})$

Correctness by previous lemmas and correctness of split&list.

Definition (MaxCut)

Input:	A graph $G = (V, E)$.
Feasible solutions:	Any bipartition $V = V_1 \cup V_2$.
Goal:	Maximize the number of edges that are *cut* by the bipartition.



Split&List

The same idea?

- Partition V into V_1, V_2
- For each $X \subset V_1$ and each $Y \subset X_2$ do something

Does this work?

Split&List

The same idea?

- Partition V into V_1, V_2
- For each $X \subset V_1$ and each $Y \subset X_2$ do something

Does this work?

Problems:

- Costs of $X \subseteq V_1$ depend on $Y \subseteq V_2$
- Costs of $Y \subseteq V_2$ depend on $X \subseteq V_1$

Graph Representation

First Idea

- Bipartite Graph
- Node for each $X \subseteq V_1$, $Y \subseteq V_2$
- Edge of cost $cut(X \cup Y, (V_1 \setminus X) \cup (V_2 \setminus Y)$
- Solution = largest cost of an edge

Graph Representation

First Idea

- Bipartite Graph
- Node for each $X \subseteq V_1$, $Y \subseteq V_2$
- Edge of cost $cut(X \cup Y, (V_1 \setminus X) \cup (V_2 \setminus Y)$
- Solution = largest cost of an edge

Problem

- There are 2^n edges.
- We have to compute costs for all edges

We need fewer edges!

Trade Off

Runtime depends on size of intermediate problem and how fast we can solve it.

- Many edges
- Finding the optimum is easy

Second Approach:

- Graph structure captures less information
- Finding optimum is harder to compute

Trade Off

Runtime depends on size of intermediate problem and how fast we can solve it.

- Many edges
- Finding the optimum is easy

Second Approach:

- Graph structure captures less information
- Finding optimum is harder to compute

- Partition into $V_1, V_2, \ldots V_k$
- k-partite Graph with $k2^{n/k}$ nodes and $1/2k^2 2^{2n/k}$ edges
- Find most expensive clique in weighted k-partite graph

But how can we compute the most expensive clique?

Cycles instead of Cliques

- A 3-cycle is a 3-clique
- Triangles can be found easily (faster than n^3)

- Partition V into V_1, V_2, V_3
- For each $X_1 \subseteq V_1$,$X_2 \subseteq V_2$, $X_3 \subseteq V_3$ we obtain the costs :

$$cut(X_1 \cup X_2 \cup X_3, V \setminus (X_1 \cup X_2 \cup X_3))$$

Cycles instead of Cliques

- A 3-cycle is a 3-clique
- Triangles can be found easily (faster than n^3)

- Partition V into V_1, V_2, V_3
- For each $X_1 \subseteq V_1$,$X_2 \subseteq V_2$, $X_3 \subseteq V_3$ we obtain the costs :

$$cut(X_1 \cup X_2 \cup X_3, V \setminus (X_1 \cup X_2 \cup X_3))$$

$$\begin{aligned}
& cut(X_1 \cup X_2 \cup X_3, V \setminus (X_1 \cup X_2 \cup X_3)) \\
= \; & cut(X_1, V_1) + cut(X_2, V_2) + cut(X_3, V_3) \\
+ \; & cut(X_1, V_2) + cut(X_2, V_1) \\
+ \; & cut(X_2, V_3) + cut(X_3, V_2) \\
+ \; & cut(X_1, V_3) + cut(X_3, V_1)
\end{aligned}$$

3-partite Graph

$$
\begin{aligned}
& cut(X_1 \cup X_2 \cup X_3, V \setminus (X_1 \cup X_2 \cup X_3)) \\
= \quad & cut(X_1, V_1) + cut(X_1, V_2) + cut(X_2, V_1) \\
+ \quad & cut(X_2, V_2) + cut(X_2, V_3) + cut(X_3, V_2) \\
+ \quad & cut(X_3, V_3) + cut(X_3, V_1) + cut(X_1, V_3)
\end{aligned}
$$

3-partite Graph:

Nodes: $X_i \subseteq V_i$, edges: $\{\{X_i, X_j\} \mid i \neq j, X_i \subseteq V_i, V_j \subseteq V_j\}$

$$w(\{X_i, X_j\}) = cut(X_i, V_i) + cut(X_i, V_j) + cut(X_j, V_i)$$

Solving maxCut

Find a triangle of large cost

Much easier: Find a triangle.

Solving maxCut

Find a triangle of large cost

Much easier: Find a triangle.Let
$c(X_i, X_j) = cut(X_i, V_i) + cut(X_i, V_j) + cut(X_j, V_i)$

For $t_1 + t_2 + t_3 = t$:

- $$\{\{X_1, X_2\} \mid X_1 \subseteq V_1, V_2 \subseteq V_2, c(X_1, X_2) = t_1\}$$
- $$\{\{X_2, X_3\} \mid X_2 \subseteq V_2, X_3 \subseteq V_3, c(X_2, X_3) = t_2\}$$
- $$\{\{X_3, X_1\} \mid X_3 \subseteq V_3, X_1 \subseteq V_1, c(X_3, X_1) = t_3\}$$

Every triangle costs exactly t.

Finding a triangle

Trivial: n^3

Lemma

Deciding whether a graph $G = (V, E)$ has a triangle can be done in time $O(V^\omega)$ where ω is the square matrix multiplication exponent.

Let $A \in \mathbf{Q}^{n \times n}$. ω is the smallest value such that A^2 can be computed in time $O(n^\omega)$.

- Let A be the adjacency matrix of G
- $A^2[i, j] \neq 0$ if $A[i, t] \neq 0$ and $A[t, j] \neq 0$ for some t
- $A^2[i, j] \neq 0$ if there is a path of length 2 from i to j

Finding a triangle

- Let A be the adjacency matrix of G
- $A^3[i,j] \neq 0$ if $A^2[i,t] \neq 0$ and $A[t,j] \neq 0$ for some t
- $A^3[i,j] \neq 0$
  if there is a path of length 2 from i to t and an edge from t to j
- $A^3[i,i] \neq 0$
  if there is a path of length 2 from i to t and an edge from t to i

$A^3[i,i] \neq 0$ if there is a triangle containing i.

Lemma

MaxCut *can be solved in time* $O^*(2^{\omega n/3})$

Matrix Multiplication

$$C = A \cdot B,\ A, B \in \mathbb{N}^{n \times n}$$

trivial:

$O(n^3)$: $AB[i,j] = \sum_{s=1}^{n} A[i,s]B[s,j]$

Let

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}$$

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

$$f(n) = 8f(n/2)$$

Fewer Multiplications

$$
\begin{aligned}
M_1 &:= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\
M_2 &:= (A_{2,1} + A_{2,2})B_{1,1} \\
M_3 &:= A_{1,1}(B_{1,2} - B_{2,2}) \\
M_4 &:= A_{2,2}(B_{2,1} - B_{1,1}) \\
M_5 &:= (A_{1,1} + A_{1,2})B_{2,2} \\
M_6 &:= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\
M_7 &:= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})
\end{aligned}
$$

Fewer Multiplications

$$
\begin{aligned}
M_1 &:= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\
M_2 &:= (A_{2,1} + A_{2,2})B_{1,1} \\
M_3 &:= A_{1,1}(B_{1,2} - B_{2,2}) \\
M_4 &:= A_{2,2}(B_{2,1} - B_{1,1}) \\
M_5 &:= (A_{1,1} + A_{1,2})B_{2,2} \\
M_6 &:= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\
M_7 &:= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})
\end{aligned}
$$

$$
\begin{aligned}
M_1 + M_4 + M_7 - M_5 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) + A_{2,2}(B_{2,1} - B_{1,1}) \\
&\quad + (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) - (A_{1,1} + A_{1,2})B_{2,2}
\end{aligned}
$$

Fewer Multiplications

$$
\begin{aligned}
M_1 &:= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\
M_2 &:= (A_{2,1} + A_{2,2})B_{1,1} \\
M_3 &:= A_{1,1}(B_{1,2} - B_{2,2}) \\
M_4 &:= A_{2,2}(B_{2,1} - B_{1,1}) \\
M_5 &:= (A_{1,1} + A_{1,2})B_{2,2} \\
M_6 &:= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\
M_7 &:= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})
\end{aligned}
$$

$$
\begin{aligned}
M_1 + M_4 + M_7 - M_5 = {} & A_{1,1}B_{1,1} + A_{1,1}B_{2,2} + A_{2,2}B_{1,1} + A_{2,2}B_{2,2} \\
& + A_{2,2}B_{2,1} - A_{2,2}B_{1,1} + A_{1,2}B_{2,1} + A_{1,2}B_{2,2} \\
& - A_{2,2}B_{2,1} - A_{2,2}B_{2,2} - A_{1,1}B_{2,2} - A_{1,2}B_{2,2}
\end{aligned}
$$

Fewer Multiplications

$$
\begin{aligned}
C_{1,1} &= M_1 + M_4 + M_7 - M_5 \\
C_{1,2} &= M_3 + M_5 \\
C_{2,1} &= M_2 + M_4 \\
C_{2,2} &= M_1 + M_3 + M_6 - M_2
\end{aligned}
$$

Proof: Exercise!

Lemma (Strassen)

C can be computed in time $O^*(n^{2.808})$.

Number of multiplications: $f(n) = 7(n/2)$, $7 \leq 2^{2.808}$.

Lemma (Coppersmith–Winograd)

C can be computed in time $O^*(n^{2.376})$.

Theorem

MaxCut *can be solved in time* $O^*(2^{\frac{2.376n}{3}})$

Overview

1. Introduction
2. Branching
3. Measure&Conquer
4. Dynamic Programming
5. Local Search
6. Split & List
7. Lower Bounds
8. Inclusion-exclusion
9. Subset Convolution

What is possible?

- Algorithm A has a running time of at least $O^*(c^n)$
- Problem P cannot be solved in time $O^*(c^n)$ unless some conjecture fails

Simple Algorithms

Algorithm $\mathcal{IS}$

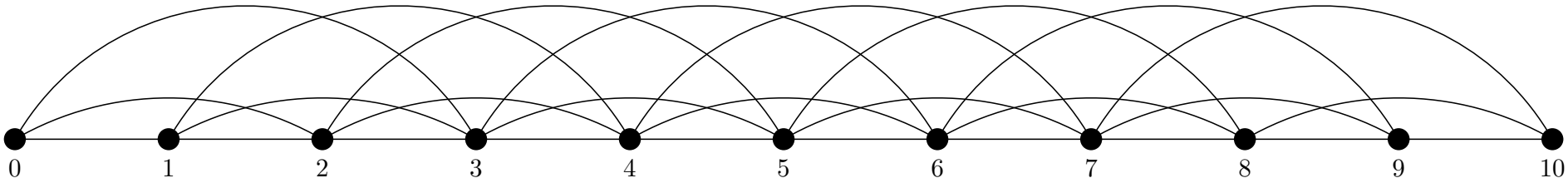
if $(\Delta(G) \leq 2)$ solve problem in polynomial time;
Choose $u \in V$ with $\deg(u) \geq 3$;
return $\max(\mathcal{IS}(G \setminus \{u\}), \mathcal{IS}(G \setminus N[u]) + 1)$;

- Known Bound: $O^*(\tau(1,4)^n)$
- More reduction rules: harder to find lower bounds
- More complex branching: harder to find lower bounds

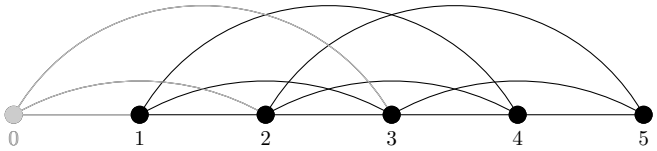
Idea

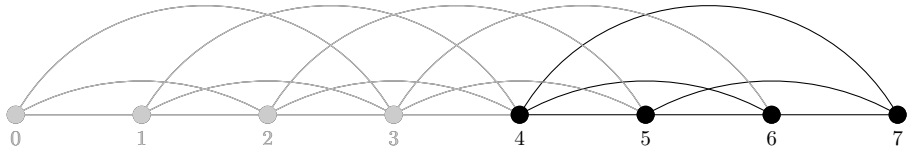
- Use underspecification to create worst case
- Which node is selected for branching?

Lower Bounds for Algorithms



Branch on node 0.





Lower Bound:

$$\Omega(\tau(1,4)^n)$$

Lower Bounds for Algorithms

In general

- Gap between upper and lower bound
- "Real" runtime ?

- Depends heavily on algorithm
- No toolbox

ETH

Exponential Time Hypothesis

3-SAT cannot be solved in time $O(2^{o(n)})$.

ETH

Exponential Time Hypothesis

3-SAT cannot be solved in time $O(2^{o(n)})$.

Strong Exponential Time Hypothesis

3-SAT cannot be solved faster than $O(2^n)$

ETH

Exponential Time Hypothesis

3-SAT cannot be solved in time $O(2^{o(n)})$.

Strong Exponential Time Hypothesis

3-SAT cannot be solved faster than $O(2^n)$

Use reductions to obtain bounds for other problems

- Size should not grow to much (linear)
- Time spent in reduction may be exponential

Sparsification

Lemma

3-SAT *can be solved in time* $O(2^{o(n)})$ *iff* 3-SAT *can be solved in time* $O(2^{o(m)})$.

- Uses Sparsification Lemma
- Not trivial

Sparsification

Lemma

3-SAT *can be solved in time* $O(2^{o(n)})$ *iff* 3-SAT *can be solved in time* $O(2^{o(m)})$.

- Uses Sparsification Lemma
- Not trivial

Lemma

Independent Set *cannot be solved in time* $O(2^{o(n)})$ *unless the ETH fails.*

- 3-SAT $\leq$ Independent Set
- New instance of size $3m + 2n$ nodes and $n + 6m$ edges.
- $O(2^{o(n)}) \leq O(2^{o(m)})$

Overview

1 Dynamic Programming

2 Inclusion-exclusion

3 Subset Convolution

Inclusion-Exclusion

Similar to dynamic programming

- algorithms go through all possible subsets;
- space required is usually polynomial.

The Inclusion-Exclusion Principle

Given

- a finite non-empty universe $\mathcal{U}$;
- sets $A_1, A_2 \subseteq \mathcal{U}$,

what is the cardinality of $A_1 \cup A_2$?

Inclusion-Exclusion

Similar to dynamic programming

- algorithms go through all possible subsets;
- space required is usually polynomial.

The Inclusion-Exclusion Principle

Given

- a finite non-empty universe $\mathcal{U}$;
- sets $A_1, A_2 \subseteq \mathcal{U}$,

what is the cardinality of $A_1 \cup A_2$?

$$|A_1 \cup A_2| = |A_1| + |A_2| - |A_1 \cap A_2|.$$

Inclusion-Exclusion

Similar to dynamic programming

- algorithms go through all possible subsets;
- space required is usually polynomial.

The Inclusion-Exclusion Principle

Given

- a finite non-empty universe $\mathcal{U}$;
- sets $A_1, A_2 \subseteq \mathcal{U}$,

what is the cardinality of $A_1 \cup A_2$?

$$|A_1 \cup A_2| = |A_1| + |A_2| - |A_1 \cap A_2|.$$

$$\begin{aligned} |\overline{A_1 \cup A_2}| &= |\mathcal{U}| - |A_1 \cup A_2| \\ &= |\mathcal{U}| - |A_1| - |A_2| + |A_1 \cup A_2|. \end{aligned}$$

The Inclusion-Exclusion Principle (Two Sets)

$$\begin{aligned} |\overline{A_1 \cup A_2}| &= |\mathcal{U}| - |A_1 \cup A_2| \\ &= |\mathcal{U}| - |A_1| - |A_2| + |A_1 \cup A_2|. \end{aligned}$$

Interpretation:

- n combinatorial objects and two properties A_1, A_2 such that each object has either property A_i or not;
- the above formula counts the number of objects that have none of these properties.

The Inclusion-Exclusion Principle

Given

- a set $\mathcal{U} = \{1, \ldots, n\}$ of n combinatorial objects;
- m properties $A_1, \ldots, A_m$ such that each object has either property A_i or not.

The number of objects that have none of these properties is given by

$$|\overline{A_1 \cup \cdots \cup A_m}| = |\mathcal{U}| + \sum_{\emptyset \neq X \subseteq [m]} (-1)^{|X|} \left| \bigcap_{i \in X} A_i \right|.$$

The Inclusion-Exclusion Principle

Given

- a set $\mathcal{U} = \{1, \ldots, n\}$ of n combinatorial objects;
- m properties $A_1, \ldots, A_m$ such that each object has either property A_i or not.

The number of objects that have none of these properties is given by

$$|\overline{A_1 \cup \cdots \cup A_m}| = |\mathcal{U}| + \sum_{\emptyset \neq X \subseteq [m]} (-1)^{|X|} \left| \bigcap_{i \in X} A_i \right|.$$

With the convention that $\bigcap_{i \in \emptyset} A_i = \mathcal{U}$, the above formula may be written as:

$$|\overline{A_1 \cup \cdots \cup A_m}| = \sum_{X \subseteq [m]} (-1)^{|X|} \left| \bigcap_{i \in X} A_i \right|.$$

The Inclusion-Exclusion Principle: A Proof

Theorem

Let $\mathcal{U}$ be a finite non-empty set and let $A_1, \ldots, A_m \subseteq \mathcal{U}$. Then

$$|\overline{A_1 \cup \cdots \cup A_m}| = \sum_{X \subseteq [m]} (-1)^{|X|} \left| \bigcap_{i \in X} A_i \right|.$$

Proof

A proof by induction on m exists but is tedious. An alternative proof is by determining how often each object is counted on either side of the equation.

- If an object has none of the properties $A_1, \ldots, A_m$, then it is counted once on the left hand side and once on the right hand side.

The Inclusion-Exclusion Principle: A Proof

Proof (continued)

Consider an object that has precisely the properties $A_{i_1}, \ldots, A_{i_r}$.

- Such an object is counted zero times on the left hand side.
- For each term $|A_{j_1} \cap \cdots \cap A_{j_s}|$, where $\{j_1, \ldots, j_s\} \subseteq \{i_1, \ldots, i_r\}$, such an object is counted once on the right hand side.
- Taking into account the sign alternation, the contribution of such an object on the right hand side is

$$\sum_{s=0}^{r} \binom{r}{s} (-1)^s = (1-1)^r = 0.$$

- Since both sides count each object an identical number of times, the equation is valid.

An Alternative Form of the Inclusion-Exclusion Formula

$$|\overline{A_1 \cup \cdots \cup A_m}| = \sum_{X \subseteq [m]} (-1)^{|X|} \left| \bigcap_{i \in X} A_i \right|.$$

This may be written as

$$|\bar{A}_1 \cap \cdots \cap \bar{A}_m| = \sum_{X \subseteq [m]} (-1)^{|X|} \left| \bigcap_{i \in X} A_i \right|.$$

An Alternative Form of the Inclusion-Exclusion Formula

$$|\overline{A_1 \cup \cdots \cup A_m}| = \sum_{X \subseteq [m]} (-1)^{|X|} \left| \bigcap_{i \in X} A_i \right|.$$

This may be written as

$$|\bar{A}_1 \cap \cdots \cap \bar{A}_m| = \sum_{X \subseteq [m]} (-1)^{|X|} \left| \bigcap_{i \in X} A_i \right|.$$

Setting $\bar{A}_i = A_i$, we obtain

An Alternative Form

$$|A_1 \cap \cdots \cap A_m| = \sum_{X \subseteq [m]} (-1)^{|X|} \left| \bigcap_{i \in X} \bar{A}_i \right|.$$

This latter formula counts the number of objects with all the properties.

An Alternative Form of the Inclusion-Exclusion Formula

An Alternative Form

Given a set of n combinatorial objects and m properties $A_1, \ldots, A_m$ such that each object has either property A_i or not. The number of objects with all these properties is given by

$$|A_1 \cap \cdots \cap A_m| = \sum_{X \subseteq [m]} (-1)^{|X|} \left| \bigcap_{i \in X} \bar{A}_i \right|.$$

Note

Inclusion-Exclusion is usually used for problems where it is difficult to directly determine the number of objects that satisfy all properties but it is easier to determine the terms

$$|\bar{A}_{i_1} \cap \bar{A}_{i_2} \cap \cdots \cap \bar{A}_{i_r}|.$$

Counting Derangements

Definition

A derangement of the set $\{1, \ldots, n\}$ is a permutation $\pi\colon [n] \to [n]$ such that $\pi(i) \neq i$ for all $1 \leq i \leq n$.

- **Objects:** permutations of $[n]$.
- **Properties:** an object π satisfies property A_i for $1 \leq i \leq n$ if $\pi(i) = i$.
- An object that does not satisfy any property A_i is a derangement.
- **Note:** $\left|\bigcap_{i \in X} A_i\right| = (n - |X|)!$.

By Inclusion-Exclusion,

$$\begin{aligned} |\bar{A}_1 \cap \cdots \cap \bar{A}_n| &= \sum_{X \subseteq [n]} (-1)^{|X|} \left|\bigcap_{i \in X} A_i\right| \\ &= \sum_{i=0}^{n} \binom{n}{i} (-1)^i (n-i)! = \sum_{i=0}^{n} (-1)^i \frac{n!}{i!}. \end{aligned}$$

Computing the Permanent

Definition

Given a matrix $A = [a_{ij}]_{n \times n}$, the **permanent** of A is defined as

$$\text{perm}(A) = \sum_{\pi \in S_n} \prod_{i=1}^{n} a_{i,\pi(i)},$$

where S_n is the set of all permutations of $\{1, \ldots, n\}$

Example

$$\begin{aligned} A &= \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}. \\ \text{perm}(A) &= a_{11}a_{22} + a_{21}a_{12}. \end{aligned}$$

Computing the Permanent

Definition

Given a matrix $A = [a_{ij}]_{n \times n}$, the **permanent** of A is defined as

$$\text{perm}(A) = \sum_{\pi \in S_n} \prod_{i=1}^{n} a_{i,\pi(i)},$$

where S_n is the set of all permutations of $\{1, \ldots, n\}$

Example

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}.$$

$$\begin{aligned} \text{perm}(A) &= a_{11}a_{22}a_{33} + a_{11}a_{32}a_{23} + a_{21}a_{12}a_{33} \\ & \quad a_{21}a_{32}a_{13} + a_{31}a_{12}a_{23} + a_{31}a_{22}a_{13}. \end{aligned}$$

The Permanent and the Determinant

Recall that the determinant of an $n \times n$ matrix is defined as

$$\det(A) = \sum_{\pi} (-1)^{\pi} \prod_{i=1}^{n} a_{i,\pi(i)},$$

where $(-1)^{\pi}$ is $+1$ for even permutations and -1 for odd permutations.

- The determinant can be computed in $O(n^3)$ time by Gaussian elimination.
- There is no polynomial time algorithm known for the permanent; the naive algorithm takes time $O(n!)$.
- Computing the permanent is #P-complete [Valiant, 1979] – computing it is as hard as computing the number of solutions to NP-complete problems such as SAT.

Computing the Permanent of Binary Matrices . . .

. . . is equivalent to computing the number of perfect matchings in a bipartite graph.

- Given a bipartite graph $G = (V_1 \cup V_2, E)$, let A_G be its adjacency matrix.
  - $A_G[i, j] = 1$ if and only if $(i, j) \in V_1 \times V_2$ is an edge.
- For a permutation π, the edges $(1, \pi(1)), \ldots, (n, \pi(n))$ form a perfect matching iff $\prod_{i=1}^{n} A_G[i, \pi(i)] = 1$.
- Hence

$$\mathsf{perm}(A_G) = \sum_{\pi} \prod_{i=1}^{n} A_G[i, \pi(i)]$$

  counts the number of perfect matchings in G.

An Alternative Formula for the Permanent

Lemma

Let $G = (V_1 \cup V_2, E)$ be a bipartite graph and let A be its adjacency matrix. Then

$$\mathsf{perm}(A) = \sum_{W \subseteq [n]} (-1)^{|W|} \prod_{i=1}^{n} \sum_{j \notin W} A[i,j].$$

Proof

We will use the following version of the Inclusion-Exclusion Theorem:

$$|P_1 \cap \cdots \cap P_m| = \sum_{X \subseteq [m]} (-1)^{|X|} \left| \bigcap_{i \in X} \bar{P}_i \right|.$$

An Alternative Formula for the Permanent

Proof (continued)

- **Objects:** a set of n edges such that each vertex in V_1 is adjacent to an edge.
- **Properties:** an object satisfies property P_j, for $1 \leq j \leq n$, if $j \in V_2$ to adjacent to an edge of the object.
- An object that satisfies all properties $P_1, \ldots, P_n$ is a perfect matching.
- For $W \subseteq \{1, \ldots, n\}$,

$$\left| \bigcap_{k \in W} \bar{P}_k \right| = \prod_{i=1}^{n} \sum_{j \notin W} A[i, j].$$

- Therefore the number of perfect matchings $(\text{perm}(A))$ equals

$$\sum_{W \subseteq [n]} (-1)^{|W|} \prod_{i=1}^{n} \sum_{j \notin W} A[i, j].$$

The Permanent of Binary Matrices

Theorem

The permanent of an $n \times n$ binary matrix can be computed in time $O^*(2^n)$ and polynomial space.

Proof

Given an $n \times n$ binary matrix A, we know

$$\text{perm}(A) = \sum_{W \subseteq [n]} (-1)^{|W|} \prod_{i=1}^{n} \sum_{j \notin W} A[i,j].$$

- For each $W \subseteq [n]$, one can compute the term $\prod_{i=1}^{n} \sum_{j \notin W} A[i,j]$ in polynomial (in n) time.
- Hence computing $\text{perm}(A)$ takes time $O^*(2^n)$.

Directed Hamiltonian s, t-Path

Definition

A **Hamiltonian path** in an n-vertex (directed) graph is sequence $v_1, \ldots, v_n$ of distinct vertices such that for all $1 \leq i \leq n-1$, there is an edge from v_i to v_{i+1}.

Problem Definition

Input: A directed graph $D = (V, A)$, where $V = \{s, t, v_1, \ldots, v_n\}$.
Question: Is there a (directed) Hamiltonian path from s to t?

- NP-complete.
- Naive algorithm takes time $\Omega(n!)$.
- The solution using Inclusion-Exclusion actually counts the number of Hamiltonian paths in time $O^*(2^n)$.

Counting the Number of Hamiltonian Paths

Theorem

The number of directed s,t-paths can be computed in time $O^*(2^n)$ and polynomial space.

Proof

The Inclusion-Exclusion scenario:

- n objects;
- m properties $A_1, \ldots, A_m$.

The number of objects that satisfy all properties:

$$|A_1 \cap \cdots \cap A_m| = \sum_{X \subseteq [m]} (-1)^{|X|} \left| \bigcap_{i \in X} \bar{A}_i \right|.$$

Counting the Number of Hamiltonian Paths

Theorem

The number of directed s,t-paths can be computed in time $O^*(2^n)$ and polynomial space.

Proof (continued)

- **Objects:** Directed walks of length $n+1$ from s to t.
- **Properties:** An object has property A_i for $1 \leq i \leq n$ if it contains vertex v_i.
- For $X \subseteq [n]$, the term $\left|\bigcap_{i \in X} \bar{A}_i\right|$ is the number of objects that do not contain vertices v_i with $i \in X$.
- Let $A[X]$ be the adjacency matrix of $D \setminus X$. Then $\left|\bigcap_{i \in X} \bar{A}_i\right|$ is the (s,t)th entry in $A[X]^{n+1}$, which is computable in polynomial time.
- One can compute the Inclusion-Exclusion formula in time $O(2^n)$ time and polynomial space.

Counting the Number of Hamiltonian Paths

Computing $\left|\bigcap_{i \in X} \bar{A}_i\right|$ by dynamic programming

- For $X \subseteq [n]$ and $u \in V \setminus X$, define $P_X(u, k) =$ number of directed walks of length k from s to u that avoid the vertices in X.
- $\left|\bigcap_{i \in X} \bar{A}_i\right| = P_X(t, n+1)$.
- $P_X(u, 0) = 1$ if $u = s$ and 0 otherwise.
- $P_X(u, k) = \sum_{v \in N^-(u) \setminus X} P_X(v, k-1)$.

Time and Space Requirements

- for fixed k, time taken to compute $P_X(u, k)$ for all u: $O\left(\sum_u (d(u) + 1)\right) = O(m)$.
- there are n values of k and total time taken is $O(mn)$.
- Space: $O(n)$.

Computing an s, t-Hamiltonian Path

The Inclusion-Exclusion algorithm counts the number of s, t-Hamiltonian paths. Can we use it to obtain one such path?

Computing an s, t-Hamiltonian Path

The Inclusion-Exclusion algorithm counts the number of s, t-Hamiltonian paths. Can we use it to obtain one such path?

Yes! By a method called self-reduction.

- If the count is positive, we know there exists at least one Hamiltonian path.
- For an edge e, delete e from the graph and count the number of Hamiltonian paths; if this is positive, there is such a path without e (non-essential edge); else e (essential edge) is in every Hamiltonian path of the graph.
- If an edge is essential, label it so that it is not deleted any more. If an edge is non-essential, delete it permanently and repeat this procedure for the next unlabeled edge.

Bin Packing

Definition

Input: Positive integers B and k denoting the bin capacity and the number of available bins; and n items of (positive integer) sizes $s_1, \ldots, s_n$

Question: Is there a partition of the n items into k sets such that each set can be fitted into a bin?

- NP-complete.
- Naive algorithm takes time $O(n^n)$.
- With Inclusion-Exclusion, we can do this in time $O^*(2^n B)$ and space $O^*(\max_i s_i)$.
- Note that the sizes s_i and the bin capacity B may be exponential in n and so the algorithm may require exponential space.

Bin Packing: Feasible Solutions

Feasible Solutions: A partition $U_1, \ldots, U_k$ of the items such that the sum of sizes in each U_i is at most B.
Assumption: Items may appear more than once either in the same bin or in different bins.
Relaxed Feasible Solution: A k-tuple $(U_1, \ldots, U_k)$ of sets such that each item occurs in at least one set and the sum of the sizes in each U_i is at most B.

Lemma

An instance of Bin Packing has a feasible solution if and only if it has a relaxed feasible solution.

Note that we impose an ordering (on the bins) as we will count the number of relaxed feasible solutions.

Counting the Number of Relaxed Feasible Solutions

Cast as an Inclusion-Exclusion problem:

Objects: A k-tuple $(U_1, \ldots, U_k)$ such that each U_i is a sequence from $1, \ldots, n$ such that $\sum_{j \in U_i} s_j \leq B$.
Properties: An object has property A_i if element i appears in at least one of the sequences.

An object that satisfies properties $A_1, \ldots, A_n$ is a relaxed feasible solution. By IE,

$$|A_1 \cap \cdots \cap A_n| = \sum_{X \subseteq [n]} (-1)^{|X|} \left| \bigcap_{i \in X} \bar{A}_i \right|.$$

For $X \subseteq [n]$, define

$$N(X) := \text{no. of sequences } U \text{ from } [n] \setminus X \text{ such that } \textstyle\sum_{i \in U} s_i \leq B.$$

Then $\left|\bigcap_{i \in X} \bar{A}_i\right| = N(X)^k$. Suffices to compute $N(X)$.

Computing $N(X)$

For $0 \leq j \leq B$, define

$$P_X(j) := \text{no. of sequences } U \text{ from } [n] \setminus X \text{ such that } \sum_{i \in U} s_i = j.$$

Also define $P_X(j) = 0$ for all $j < 0$.

Note: $N(X) = \sum_{i=0}^{B} P_X(i)$.

Computing $P_X(j)$

- $P_X(0) = 1$ (the empty sequence avoids all of X and has sum zero);
- For $j \geq 1$, the values of $P_X(j)$ can be computed by the recurrence:

$$P_X(j) = \sum_{i \notin X} P_X(j - s_i).$$

Time and Space Requirements

Computing $P_X(i)$ for $1 \leq i \leq B$

- **Time:** $O(Bn)$.
- **Space:** $O(\max_i s_i)$ (to compute $P_X(j)$, only the last $\max_i s_i$ values need be stored.)

Therefore

- **Total Time Required:** $O^*(2^n Bn)$.
- **Total Space Required:** $O(\max_i s_i)$.

Branching Walks . . .

. . . are a generalization of walks.

Just as Hamiltonian Path can be solved by counting the number of walks, many problems can be formulated in terms of counting branching walks [Nederlof 2009]:

- Steiner Tree;
- Degree Constrained Spanning Tree;
- Max Internal Spanning Tree;
- and many others . . .

A branching walk is defined via . . .

Homomorphisms

Definition (Homomorphisms)

Given graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, a **homomorphism** from G_1 to G_2 is a function $\varphi\colon V_1 \to V_2$ such that $(u, v) \in E_1$ implies that $(\varphi(u), \varphi(v)) \in E_2$.

- Homomorphisms generalize colorings: a homomorphism from G to the complete graph K_r is exactly an r-coloring of G.
- If $\varphi\colon G \to H$ is a homomorphism and G is connected then so is $\varphi(G)$.

Branching Walks

Definition

A **branching walk** B in a graph G is a pair (T, φ) where T is a rooted ordered tree and $\varphi\colon V(T) \to V(G)$ is a homomorphism from T to G. For $s \in V(G)$, the branching walk B **is from** s if $\varphi(\text{root}(T)) = s$. The **length** of B is $|E(T)|$.

- If T is a path then a branching walk (T, φ) is a walk.
- Two branching walks (T_1, φ_1) and (T_2, φ_2) are distinct if there is no isomorphism $\psi\colon T_1 \to T_2$ such that $\varphi_1(v) = \varphi_2(\psi(v))$ for all $v \in V(T_1)$.

The Steiner Tree Problem

Definition

Input: A connected graph $G = (V, E)$, a set $K \subseteq V$ of terminals and an integer $c \in \mathbf{N}$.

Question: Does G have a tree (as subgraph) with at most c edges that contains all terminals?

Lemma

Let s be an arbitrary but fixed terminal. There exists a tree $S = (V', E')$ in G with $K \subseteq V'$ and $|E'| \leq c$ iff there exists a branching walk $B = (T, \varphi)$ from s such that $K \subseteq \varphi(V(T))$ and $|\varphi(E(T))| \leq c$.

Lemma

Let s be an arbitrary but fixed terminal. There exists a tree $S = (V', E')$ in G with $K \subseteq V'$ and $|E'| \leq c$ iff there exists a branching walk $B = (T, \varphi)$ from s such that $K \subseteq \varphi(V(T))$ and $|\varphi(E(T)| \leq c$.

Proof

Forward direction:

- Suppose there exists a tree $S = (V', E')$ in G with $K \subseteq V'$ and $|E'| \leq c$.
- Root S at the terminal s and order the nodes of S in some arbitrary order.
- Define $B = (S, \varphi)$ where $\varphi \colon V' \to V'$ is the identity function.
- Clearly $K \subseteq \varphi(V(T))$ and $|\varphi(E(T))| \leq c$.

The Steiner Tree Problem and Branching Walks

Lemma

Let s be an arbitrary but fixed terminal. There exists a tree $S = (V', E')$ in G with $K \subseteq V'$ and $|E'| \leq c$ iff there exists a branching walk $B = (T, \varphi)$ from s such that $K \subseteq \varphi(V(T))$ and $|\varphi(E(T)| \leq c$.

Proof

Reverse direction:

- Suppose there exists a branching walk $B = (T, \varphi)$ from s such that $K \subseteq \varphi(V(T))$ and $|\varphi(E(T))| \leq c$.
- Then $(\varphi(V(T)), \varphi(E(T)))$ is connected.
- A spanning tree of $(\varphi(V(T)), \varphi(E(T)))$ is a tree in G with the required properties.

Counting the number of Steiner trees of with at most c edges is equivalent to counting the number of branching walks from some fixed terminal s.

IE-Formulation for Counting Branching Walks

Let s be an arbitrary but fixed terminal vertex.

- **Objects:** branching walks (T, φ) from s with $|\varphi(E(T))| \leq c$.
- **Properties:** an object (T, φ) satisfies property A_v for $v \in K$ if $v \in \varphi(V(T))$.
- An object that satisfies properties A_v for all $v \in K$ represents a Steiner tree with at most c edges.
- That is, $\left|\bigcap_{v \in K} A_v\right|$ represents the number of Steiner trees with at most c edges.

By Inclusion-Exclusion

$$\left|\bigcap_{v \in K} A_v\right| = \sum_{X \subseteq K} (-1)^{|X|} \left|\bigcap_{v \in X} \bar{A}_v\right|.$$

Now $\left|\bigcap_{v \in X} \bar{A}_v\right|$ represents the number of branching walks from s with at most c edges avoiding all the vertices in X.

Computing the Terms of the IE-Formula

For $X \subseteq K$, $u \in V \setminus X$ and $0 \leq w \leq c$, define

$B_X(u, w) =$ *set of all branching walks* (T, φ) *s.t.* $|\varphi(E(T))| \leq w$ *from* u *in* $G[V \setminus X]$.

Then

- The set of all objects $= B_\emptyset(s, c)$.
- $\left|\bigcap_{v \in X} \bar{A}_v\right| = |B_X(s, c)| := b_X(s, c)$.

Computing $b_X(s, c)$ by Dynamic Programming

Lemma

Let $X \subseteq K$, $u \in V \setminus X$ *and* $0 \leq w \leq c$,

$$b_X(u, w) = \begin{cases} 1 & \text{if } w = 0; \\ \displaystyle\sum_{v \in N(u) \setminus X} \sum_{w_1 + w_2 = w-1} b_X(u, w_1) \cdot b_X(v, w_2) & \text{otherwise.} \end{cases}$$

Computing the Terms of the IE-Formula ...

Proof

Case 1. There is one branching walk (T, φ) with $|\varphi(E(T))| = 0$, from u with T being a single vertex and φ mapping this vertex to u.

Computing the Terms of the IE-Formula ...

Proof

Case 2. Define a map γ

$$\gamma\colon \bigcup_{v\in N(u)\setminus X} \bigcup_{w_1,w_2} B_X(u,w_1)\times B_X(v,w_1) \to B_X(u,w_1+w_2+1),$$

as follows:

$$\gamma((T_1,\varphi_1),(T_2,\varphi_2)) \mapsto (T_1 \bullet T_2, \varphi_1 \cup \varphi_2),$$

where

- $(T_1 \bullet T_2)$ is the rooted ordered tree obtained by adding an edge from $\mathrm{root}(T_1)$ to $\mathrm{root}(T_2)$ and setting $\mathrm{root}(T_1 \bullet T_2) = \mathrm{root}(T_1)$.
- $\varphi(u) = \varphi_i(u)$ for $u \in V(T_i)$.

Note: The map γ is a bijection and this proves Case 2.

Time and Space Requirements for Computing $b_X(u, w)$

Time Complexity

- Time taken to compute $b_X(u, w)$ is $O(w|N(u)|)$.
- Time taken to compute $b_X(u, w)$ for all u is $O(w|E(G)|)$.
- Time required to compute $b_X(s, c)$ is $O(c^2|E(G)|)$.

Space Complexity

- Space Required is $O(c|V(G)|)$.

Theorem

The Steiner Tree problem can be solved in time $O^*(2^k c^2)$ and space $O^*(c)$.

The Degree Constrained Spanning Tree Problem

Definition

Input: A connected graph $G = (V, E)$ on n vertices and $1 \leq c \leq n$.
Question: Does G have a spanning tree with maximum degree at most c?

- Generalization of Hamiltonian Path (where $c = 2$) and hence NP-complete.

Connection with Branching Walks

Lemma

Let s be an arbitrarily chosen vertex. There exists a spanning tree of G with maximum degree at most c iff there exists a branching walk $B = (T, \varphi)$ from s of length $n - 1$ such that $\varphi(V(T)) = V(G)$ and the maximum degree of T is at most c.

Connection with Branching Walks

Lemma

Let s be an arbitrarily chosen vertex. There exists a spanning tree of G with maximum degree at most c iff there exists a branching walk $B = (T, \varphi)$ from s of length $n - 1$ such that $\varphi(V(T)) = V(G)$ and the maximum degree of T is at most c.

Proof

Forward direction: Let S be a spanning tree of G with maximum degree at most c.

- Root S at the vertex s and fix an arbitrary ordering of S.
- Define $B = (S, \varphi)$, where $\varphi \colon V(S) \to V(S)$ is the identity mapping.

Then $B = (S, \varphi)$ satisfies the required properties.

Connection with Branching Walks

Lemma

Let s be an arbitrarily chosen vertex. There exists a spanning tree of G with maximum degree at most c iff there exists a branching walk $B = (T, \varphi)$ from s of length $n - 1$ such that $\varphi(V(T)) = V(G)$ and the maximum degree of T is at most c.

Proof

Reverse direction: Let $B = (T, \varphi)$ be a branching walk from s with the stated properties.

- Then $(\varphi(V(T)), \varphi(E(T)))$ must be a spanning tree of G of degree at most c.

This proves the lemma.

Number of spanning trees with degree at most c = number of branching walks from s with the stated properties.

Counting Branching Walks Using IE

Given a graph $G = (V, E)$ with n vertices, fix $s \in V(G)$:

- **Objects:** branching walks $B = (T, \varphi)$ from s of length $n - 1$ such that maximum degree of $T \leq c$.
- **Properties:** an object $B = (T, \varphi)$ has property A_v for $v \in V(G)$ if $v \in \varphi(V(T))$.
- Interested in finding the number of objects that satisfy all properties.

$$|A_1 \cap \cdots \cap A_n| = \sum_{X \subseteq V(G)} (-1)^{|X|} \left| \bigcap_{i \in X} \bar{A}_i \right|.$$

Note that $\left|\bigcap_{i \in X} \bar{A}_i\right|$ = no. of branching walks from s of length $n - 1$ that avoid X.

Computing the Terms of the IE-Formula

For $X \subseteq V$, $u \in V \setminus X$ and $1 \leq d \leq c$, define $D_X(u, j, d) =$ set of branching walks (T, φ) from u of length j s.t.

1. maximum degree of T is at most c;
2. $\deg(\text{root}(T)) \leq d$;
3. $X \cap \varphi(V(T)) = \emptyset$.

Computing the Terms of the IE-Formula

For $X \subseteq V$, $u \in V \setminus X$ and $1 \leq d \leq c$, define $D_X(u, j, d) =$ set of branching walks (T, φ) from u of length j s.t.

1. maximum degree of T is at most c;
2. $\deg(\text{root}(T)) \leq d$;
3. $X \cap \varphi(V(T)) = \emptyset$.

Note: Set of all objects $= D_\emptyset(s, n-1, c)$ and

$$\left| \bigcap_{i \in X} \bar{A}_i \right| = |D_X(s, n-1, c)| := d_X(s, n-1, c).$$

Now $d_X(u, 0, d) = 1$ as there is only one branching walk from u of length zero and this contains just a single vertex.

Computing the Terms of the IE-Formula ...

Lemma

For $X \subseteq V$, $u \in V \setminus X$, $1 \leq d \leq c$ and $j \geq 1$

$$d_X(u, j, d) = \sum_{v \in N(u) \setminus X} \sum_{j_1 + j_2 = j-1} d_X(u, j_1, d-1) \cdot d_X(v, j_2, c-1).$$

Proof

Define a map γ

$$\gamma\colon \bigcup_{v \in N(u) \setminus X} \bigcup_{j_1, j_2} D_X(u, j_1, d-1) \times D_X(v, j_2, c-1) \to D_X(u, j_1 + j_2 + 1, d)$$

as follows:

$$\gamma((T_1, \varphi_1), (T_2, \varphi_2)) \mapsto (T = T_1 \bullet T_2, \varphi = \varphi_1 \cup \varphi_2),$$

where $T_1 \bullet T_2$ is as defined previously. Sufficient to show that γ is a bijection.

Computing the Terms of the IE-Formula ...

Lemma

For $X \subseteq V$, $u \in V \setminus X$, $1 \leq d \leq c$ and $j \geq 1$

$$d_X(u, j, d) = \sum_{v \in N(u) \setminus X} \sum_{j_1 + j_2 = j-1} d_X(u, j_1, d-1) \cdot d_X(v, j_2, c-1).$$

Proof (continued)

Define a (c, d)-tree as a tree of maximum degree c but whose root has degree at most d.

- $T_1 \bullet T_2$ is a (c, d)-tree iff T_1 is a $(c, d-1)$-tree and T_2 is a $(c, c-1)$-tree.
- $|E(T_1 \bullet T_2)| = j$ iff $|E(T_1)| = j_1$ and $|E(T_2)| = j_2$, where $j = j_1 + j_2 + 1$.
- Since $(u, v) \in E(G)$, we have $\varphi(\text{root}(T_2)) \in N(\varphi(\text{root}(T_1)))$.
- Hence γ is well-defined. Since $T_1 \bullet T_2$ uniquely determines T_1 and T_2 and vice versa, γ is a bijection and the formula holds.

Time and Space Requirements

Time Complexity

- For fixed u, j, d, time taken to compute $d_X(u, j, d)$ is $O(n \cdot |N(u)|)$.
- For fixed j, d, time taken to compute $d_X(u, j, d)$ for all u is $O(nm)$.
- Time taken to compute $d_X(s, n-1, c)$ is $O(n^2mc)$.

Space Complexity

- Space required $= O(n^2c)$.

Theorem

The Degree Constrained Spanning Tree problem can be decided in time $O^*(2^n)$ and space polynomial in n.

Covering and Partition Problems

Bin Packing can be viewed as an instance of a covering problem:

- cover the set $\{1, \ldots, n\}$ with at most k sequences such that the sum of the elements in every sequence is at most B.

For Bin Packing, the run-time of the IE-algorithm is the same as that of the DP-algorithm.

However for a number of covering and partition problems, the run-time of the IE-algorithm is significantly better than the DP-algorithm.

- Chromatic Number: best DP-algorithm: $O^*(2.40^n)$, IE-algorithm: $O^*(2^n)$.

The Basic Covering Problem: Set Cover

Definition

Input: A universe $\mathcal{U}$ with n elements and a family $\mathcal{F}$ of subsets of $\mathcal{U}$.
Goal: Find out a smallest subset of $\mathcal{F}$ that covers $\mathcal{U}$.

Assumptions

1. The sets of $\mathcal{F}$ can be enumerated in time $O^*(2^n)$.
2. In a typical example, the family $\mathcal{F}$ is not specified explicitly but rather *implicitly* by a polynomial-time computable predicate.

Terminology

- k-cover: a collection $S_1, \ldots S_k$ of k sets such that $\bigcup_i S_i = \mathcal{U}$.
- k-partition: a collection of k sets $S_1, \ldots, S_k \in \mathcal{F}$ such that $S_i \cap S_j = \emptyset$ for all $i \neq j$ and $\bigcup_i S_i = \mathcal{U}$.

Graph Coloring as a Covering Problem

Recall

- A **feasible coloring** of a graph G is an assignment of colors to the vertices of the graph such that adjacent vertices are assigned different colors.
- The minimum number of colors required in a feasible coloring is the **chromatic number** of the graph.

Lemma

Given a graph $G = (V, E)$, let $\mathcal{I}$ be the set of independent sets of G. Then G has a k-coloring iff the set system $(V, \mathcal{I})$ has a k-covering (k-partition).

Using Inclusion-Exclusion it is possible to count the number of k-partitions (k-covers).

Counting the ordered k-Covers of a Set System

An **ordered** k**-cover** of a set system $(\mathcal{U}, \mathcal{F})$ is a k-tuple $(S_1, \ldots, S_k) \in \mathcal{F}^k$ such that $\bigcup_i S_i = \mathcal{U}$.

For $X \subseteq \mathcal{U}$, define

$$\mathcal{F}[X] = \{S \in \mathcal{F} \colon S \cap X = \emptyset\}.$$

That is, $\mathcal{F}[X]$ is the subfamily of $\mathcal{F}$ that avoids X.
Also define $s[X] = |\mathcal{F}[X]|$.

Lemma

The number of ordered k-covers of a set system $(\mathcal{U}, \mathcal{F})$ is given by

$$c_k = \sum_{X \subseteq \mathcal{U}} (-1)^{|X|} s[X]^k.$$

Counting the Ordered k-Covers of a Set System ...

Proof

We will use the (now familiar) setting of Inclusion-Exclusion.

- **Objects:** elements of $\mathcal{F} \times \cdots \times \mathcal{F}$ (k-times).
- **Properties:** an object $(S_1, \ldots, S_k)$ satisfies property A_i for $i \in \mathcal{U}$ if $i \in \bigcup_j S_j$.
- An object that satisfies properties A_i for all $i \in \mathcal{U}$ is an ordered k-cover of $(\mathcal{U}, \mathcal{F})$.

By IE,

$$c_k = |A_1 \cap \cdots \cap A_n| = \sum_{X \subseteq \mathcal{U}} (-1)^{|X|} \left| \bigcap_{i \in X} \bar{A}_i \right|.$$

Now $\left|\bigcap_{i \in X} \bar{A}_i\right| = s[X]^k$ and hence the lemma follows.

Therefore all we have to do is estimate $s[X]$ for all $X \subseteq \mathcal{U}$.

Estimating $s[X]$

The quantity $s[X]$ may be written as

$$s[X] = \sum_{S \subseteq \mathcal{U} \setminus X} f(S),$$

where $f(S) = 1$ if $S \in \mathcal{F}$ and 0 otherwise.

For $i \in \{0, 1, \ldots, n\}$ and $X \subseteq \mathcal{U}$, define $g_i(X)$

$$g_i(X) = \left|\{S \in \mathcal{F}[X] \colon \{i+1, \ldots, n\} \setminus X \subseteq S\}\right|.$$

In other "words"

$$g_i(X) = \sum_{\{i+1,\ldots,n\} \setminus X \subseteq S \subseteq \mathcal{U} \setminus X} f(S).$$

Estimating $s[X]$ ...

Note that

$$\begin{aligned} g_0(X) &= f(\mathcal{U} \setminus X). \\ g_n(X) &= \sum_{S \subseteq \mathcal{U} \setminus X} f(S) = s[X]. \end{aligned}$$

For $i \geq 1$,

$$g_i(X) = \begin{cases} g_{i-1}(X), & \text{if } i \in X; \\ g_{i-1}(X \cup \{i\}) + g_{i-1}(X), & \text{otherwise.} \end{cases}$$

Explanation

- If $i \in X$ then clearly $g_i(X) = g_{i-1}(X)$.
- Otherwise $g_{i-1}(X)$ counts all sets in $\mathcal{F} \setminus X$ that contains i and $g_{i-1}(X \cup \{i\})$ counts all such sets not containing i.

Putting it all Together

To compute $g_i(X)$ for all $0 \leq i \leq n$ and all $X \subseteq \mathcal{U}$,

- create a table with 2^n entries, with one entry for each $X \subseteq \mathcal{U}$;
- for all $X \subseteq \mathcal{U}$ compute $g_0(X)$ (poly-time computable for each X);
- for each $i \geq 1$ and each $X \subseteq \mathcal{U}$ compute $g_i(X)$ using the previous recurrence.

Time and Space Required

- Time $= O^*(2^n)$.
- Space $= O^*(2^n)$.

Theorem

The ordered k-covers of a set system $(\mathcal{U}, \mathcal{F})$ can be counted in time $O^*(2^n)$ and space $O^*(2^n)$. Thus the Chromatic Number problem is solvable with the same time and space complexity.

Additional Remarks

- The previous algorithm takes exponential space as this is the assumed space complexity for enumerating the elements of $\mathcal{F}$.
- If the elements of $\mathcal{F}$ can be enumerated in polynomial space with a polynomial delay, then this algorithm can be made to run in polynomial space and time $O^*(2^n|\mathcal{F}|)$.

The Set Partitioning Problem

Definition

Input: A universe $\mathcal{U}$ with n elements and a family $\mathcal{F}$ of subsets of $\mathcal{U}$; an integer k.

Goal: Find out the number of k-partitions of $(\mathcal{U}, \mathcal{F})$.

Assumptions

1. The sets of $\mathcal{F}$ can be enumerated in time $O^*(2^n)$.
2. In a typical example, the family $\mathcal{F}$ is not specified explicitly but rather *implicitly* by a polynomial-time computable predicate.

Definitions

- Ordered k-Partitions: a k-tuple $(S_1, \ldots, S_k) \in \mathcal{F}^k$ s.t. $\bigcup_i S_i = \mathcal{U}$ and $S_i \cap S_j = \emptyset$ for all $i \neq j$.
- p_k: the number of ordered k-partitions of $(\mathcal{U}, \mathcal{F})$.

Computing p_k by Inclusion-Exclusion

For $X \subseteq [n]$,

- $\mathcal{F}[X] = \{S \in \mathcal{F} \colon S \cap X = \emptyset\}$.
- $a_k[X] =$ no. of k-tuples $(S_1, \ldots, S_k) \in \mathcal{F}[X]^k$ s.t. $\sum_i^k |S_i| = n$.

Lemma

The number of ordered k-partitions of a set system $(\mathcal{U}, \mathcal{F})$ is

$$p_k = \sum_{X \subseteq \mathcal{U}} (-1)^{|X|} a_k[X].$$

Computing p_k by Inclusion-Exclusion

Lemma

The number of ordered k-partitions of a set system $(\mathcal{U}, \mathcal{F})$ is

$$p_k = \sum_{X \subseteq \mathcal{U}} (-1)^{|X|} a_k[X].$$

Proof

Consider the following IE-formulation of the problem:

- **Objects:** k-tuples $(S_1, \ldots, S_k) \in \mathcal{F}^k$ s.t. $\sum_{i=1}^{k} |S_i| = n$.
- **Properties:** an object $(S_1, \ldots, S_k)$ satisfies property A_u for $u \in \mathcal{U}$ if $u \in \bigcup_{i=1}^{k} S_i$.
- An object that satisfies properties A_v for all $v \in \mathcal{U}$ is an ordered k-partition of $(\mathcal{U}, \mathcal{F})$.

Computing p_k by Inclusion-Exclusion

Lemma

The number of ordered k-partitions of a set system $(\mathcal{U}, \mathcal{F})$ is

$$p_k = \sum_{X \subseteq \mathcal{U}} (-1)^{|X|} a_k[X].$$

Proof

By IE,

$$|A_1 \cap \cdots \cap A_n| = \sum_{X \subseteq \mathcal{U}} (-1)^{|X|} \left| \bigcap_{i \in X} \bar{A}_i \right|.$$

Clearly $\left|\bigcap_{i \in X} \bar{A}_i\right| = a_k[X]$ and hence the lemma follows.

Therefore it is sufficient to estimate $a_k[X]$ for all $X \subseteq \mathcal{U}$.

Estimating $a_k[X]$

Computing $a_k[X]$ is a two-stage process:

For $0 \leq i \leq n$

- $\mathcal{F}^{(i)} = \{S \in \mathcal{F} \colon |S| = i\}$.
- $\mathcal{F}^{(i)}[X] = \{S \in \mathcal{F} \colon S \cap X = \emptyset, |S| = i\}$.
- $s^{(i)}[X] = |\mathcal{F}^{(i)}[X]|$.

Compute $\mathcal{F}^{(i)}[X]$ for all $X \subseteq \mathcal{U}$ by using the algorithm discussed in the k-cover problem to the set system $(\mathcal{U}, \mathcal{F}^{(i)})$.

- By assumption, for every i one can enumerate $\mathcal{F}^{(i)}$ in time $O^*(2^n)$.
- Hence for every fixed i, one can compute $\mathcal{F}^{(i)}[X]$ for all $X \subseteq \mathcal{U}$ in time $O^*(2^n)$ and space $O^*(2^n)$.

Estimating $a_k[X]$ ...

The second step uses the entries $s^{(i)}[X]$ to compute $a_k[X]$.

- Let $A(l, m, X)$ be the number of l-tuples $(S_1, \ldots, S_l)$ such that $S_i \cap X = \emptyset$ and $\sum_{i=1}^{l} |S_i| = m$.
- Then $a_k[X] = A(k, n, X)$.

Now

$$\begin{aligned} A(1, m, X) &= s^{(m)}[X]. \\ A(l, m, X) &= \sum_{i=1}^{m-1} A(l-1, i, X) \cdot s^{(m-i)}[X]. \end{aligned}$$

For a fixed $X \subseteq \mathcal{U}$, time taken to compute $A(l, m, X)$ is $O(kn)$. Since all entries $s^{(m)}[X]$ are required in the computation of $a_k[X]$ for all X, the space required in exponential.

Summing it up

Theorem

The ordered k-partitions of an n-element set system $(\mathcal{U}, \mathcal{F})$ can be enumerated in time $O^*(2^n)$ and space $O^*(2^n)$.

Poly Space Algorithms for k-Covers/Partitions

Under certain circumstances, the previous algorithms for counting k-covers and k-partitions of a set system $(\mathcal{U}, \mathcal{F})$ can be modified to run using polynomial space.

We look at two such situations:

1. The family $\mathcal{F}$ can be enumerated using polynomial space and with a polynomial delay.
2. Membership in $\mathcal{F}$ can be tested in polynomial time. That is, there is a polynomial time algorithm that given $S \subseteq \mathcal{U}$ decides whether $S \in \mathcal{F}$ in time poly$(|\mathcal{U}|)$.

Poly Space Algorithms for k-Covers

The exponential space algorithm for counting ordered k-covers is based on the IE-formula

$$c_k = \sum_{X \subseteq \mathcal{U}} (-1)^{|X|} s[X]^k,$$

where $s[X]$ is the number of sets $S \in \mathcal{F}$ not containing any element of X.

Case 1. The family $\mathcal{F}$ can be enumerated in polynomial space and with polynomial delay.

Then for any $X \subseteq \mathcal{U}$, the term $s[X]$ can be evaluated in time $O^*(|\mathcal{F}|)$ and polynomial space. Hence the above formula can be evaluated in time $O^*(2^n|\mathcal{F}|)$ and polynomial space.

Poly Space Algorithms for k-Covers

The exponential space algorithm for counting ordered k-covers is based on the IE-formula

$$c_k = \sum_{X \subseteq \mathcal{U}} (-1)^{|X|} s[X]^k,$$

where $s[X]$ is the number of sets $S \in \mathcal{F}$ not containing any element of X.

Case 2. Membership in $\mathcal{F}$ can be decided in polynomial time.

For any $X \subseteq \mathcal{U}$, run through all subsets of $S \subseteq \mathcal{U} \setminus X$ and check whether $S \in \mathcal{F}$. Then the term $s[X]$ can be evaluated in polynomial space and $O^*(2^{n-|X|})$ time. The total time taken to evaluate the IE-formula is

$$\sum_{i=0}^{n} \binom{n}{n-i} 2^{n-i} n^{O(1)} = O^*(3^n).$$

The space required in polynomial in n.

Poly Space Algorithm for Chromatic Number

The chromatic number of a graph $G = (V, E)$ can be computed by estimating the number of k-covers of the set system $(V, \mathcal{I})$, where $\mathcal{I}$ is the set of all independent sets in G.

Lemma (Fürer and Kasiviswanathan, 2007)

The independent sets of an n-vertex graph can be enumerated in polynomial space in time $O^*(1.2461^n)$.

Hence the chromatic number of an n-vertex graph can be obtained in time

$$O^*\left(\sum_{i=0}^{n} \binom{n}{n-i} \cdot 1.2461^{n-i}\right) = O^*(2.2461^n)$$

using polynomial space.

Poly Space Algorithms for k-Partitions

Recall that the number of ordered k-partitions of a set system $(\mathcal{U}, \mathcal{F})$ is

$$p_k = \sum_{X \subseteq \mathcal{U}} (-1)^{|X|} a_k[X],$$

where $a_k[X]$ is the number of k-tuples $(S_1, \ldots, S_k) \in \mathcal{F}[X]^k$ s.t. $\sum_i |S_i| = |\mathcal{U}| := n$.

Theorem

The number of ordered k-partitions p_k of a set system $(\mathcal{U}, \mathcal{F})$ can be computed in polynomial space and

- *$O^*(2^n|\mathcal{F}|)$ time if $\mathcal{F}$ can be enumerated in polynomial space with polynomial delay.*
- *$O^*(3^n)$ time if membership in $\mathcal{F}$ can be tested in polynomial time.*

Proof

Exercise!

Overview

1 Dynamic Programming

2 Inclusion-exclusion

3 Subset Convolution

Fast Subset Convolution

Let $U = \{1, \ldots, n\}$ and let 2^U denote the set of all subsets of U.

Definition

For functions $f, g \colon 2^U \to \mathbf{Z}$, the **convolution** of f and g, denoted by $f * g$, is a function from 2^U to $\mathbf{Z}$ defined as follows: for $S \subseteq U$

$$(f * g)(S) = \sum_{X \subseteq S} f(X) \cdot g(S \setminus X) = \sum_{\substack{X \cup Y = S \\ X \cap Y = \emptyset}} f(X) \cdot g(Y).$$

Note that for functions $f, g, h \colon 2^U \to \mathbf{Z}$,

- *Commutativity.* $f * g = g * f$.
- *Associativity.* $(f * g) * h = f * (g * h)$.

Therefore for $f_1, \ldots, f_k \colon 2^U \to \mathbf{Z}$, the expression

$$f_{\pi(1)} * f_{\pi(2)} * \cdots * f_{\pi(k)},$$

where π is a permutation of $[k]$, is well-defined and denotes the same function.

Examples of Convolution Products

Given a graph $G = (V, E)$, define $f : 2^V \to \mathbf{Z}$ as follows:

$$f(S) = \begin{cases} 1 & \text{if } S \subseteq V \text{ is an independent set;} \\ 0 & \text{otherwise.} \end{cases}$$

- What does $(f * f)(V)$ represent?

Examples of Convolution Products

Given a graph $G = (V, E)$, define $f : 2^V \to \mathbf{Z}$ as follows:

$$f(S) = \begin{cases} 1 & \text{if } S \subseteq V \text{ is an independent set;} \\ 0 & \text{otherwise.} \end{cases}$$

- What does $(f * f)(V)$ represent? The number of two-colorings of G.

Examples of Convolution Products

Given a graph $G = (V, E)$, define $f : 2^V \to \mathbf{Z}$ as follows:

$$f(S) = \begin{cases} 1 & \text{if } S \subseteq V \text{ is an independent set;} \\ 0 & \text{otherwise.} \end{cases}$$

- What does $(f * f)(V)$ represent? The number of two-colorings of G.
- What does

$$\underbrace{(f * f * \cdots * f}_{k \text{ times}})(V)$$

represent?

Examples of Convolution Products

Given a graph $G = (V, E)$, define $f : 2^V \to \mathbf{Z}$ as follows:

$$f(S) = \begin{cases} 1 & \text{if } S \subseteq V \text{ is an independent set;} \\ 0 & \text{otherwise.} \end{cases}$$

- What does $(f * f)(V)$ represent? The number of two-colorings of G.
- What does

$$\underbrace{(f * f * \cdots * f}_{k \text{ times}})(V)$$

represent? The number of k-colorings.

Examples of Convolution Products

Given a graph $G = (V, E)$, define $f : 2^V \to \mathbf{Z}$ as follows:

$$f(S) = \begin{cases} 1 & \text{if } S \subseteq V \text{ is a dominating set of } G; \\ 0 & \text{otherwise.} \end{cases}$$

Then

$$(\underbrace{f * f * \cdots * f}_{k \text{ times}})(V)$$

represents the number of k-domatic partitions of G.

Examples of Convolution Products

Given a set system $(U, \mathcal{F})$, define $f : 2^U \to \mathbf{Z}$ as follows: for $S \subseteq U$,

$$f(S) = \begin{cases} 1 & \text{if } S \in \mathcal{F}; \\ 0 & \text{otherwise.} \end{cases}$$

Then what does

$$(\underbrace{f * f * \cdots * f}_{k \text{ times}})(V)$$

represent?

Examples of Convolution Products

Given a set system $(U, \mathcal{F})$, define $f : 2^U \to \mathbf{Z}$ as follows: for $S \subseteq U$,

$$f(S) = \begin{cases} 1 & \text{if } S \in \mathcal{F}; \\ 0 & \text{otherwise.} \end{cases}$$

Then what does

$$\underbrace{(f * f * \cdots * f)}_{k \text{ times}}(V)$$

represent?

The number of k-partitions of $(U, \mathcal{F})$.

Computing the Convolution Product

By definition,

$$(f * g)(S) = \sum_{X \subseteq S} f(X) \cdot g(S \setminus X).$$

- For $S \subseteq U$, the time taken to compute $(f * g)(S)$ is $O^*(2^{|S|})$.
- Hence the total time taken to compute $f * g$ is

$$\sum_{i=0}^{n} \binom{n}{i} 2^i = O^*(3^n).$$

The fast subset convolution algorithm computes the convolution in time $O^*(2^n)$. This can be applied to obtain algorithms running in time $O^*(2^n)$ for a number of problems, including those mentioned previously.

The fast subset convolution algorithm relies on the zeta transform ...

The Zeta Transform

Definition

Given a function $f\colon 2^{[n]} \to \mathbf{Z}$, the **zeta transform** of f is a function $\zeta(f)\colon 2^{[n]} \to \mathbf{Z}$ defined as follows: for all $S \subseteq [n]$,

$$\zeta(f)(S) = \sum_{X \subseteq S} f(X).$$

If we are given $\zeta(f)$, we can "recover" the original function by the so-called **Möbius Inversion**.

Lemma

Let $f\colon 2^{[n]} \to \mathbf{Z}$. Then for all $S \subseteq [n]$,

$$f(S) = \sum_{X \subseteq S} (-1)^{|S \setminus X|} \zeta(f)(X).$$

The Möbius Inversion

Lemma

Let $f : 2^{[n]} \to \mathbf{Z}$. *Then for all* $S \subseteq [n]$,

$$f(S) = \sum_{X \subseteq S} (-1)^{|S \setminus X|} \zeta(f)(X).$$

Proof

The right hand side of the identity equals:

$$\begin{aligned} \sum_{X \subseteq S} (-1)^{|S \setminus X|} \zeta(f)(X) &= \sum_{X \subseteq S} (-1)^{|S \setminus X|} \sum_{Y \subseteq X} f(Y) \\ &= \sum_{Y \subseteq S} \left(\sum_{Y \subseteq X \subseteq S} (-1)^{|S \setminus X|} \right) \cdot f(Y). \end{aligned}$$

For $Y \neq S$, the inner summand is $\sum_{i=0}^{k} \binom{k}{i} (-1)^i = 0$, where $k = |S \setminus Y|$. This shows that the right hand side equals $f(S)$.

The Möbius Transform

The last result motivates the following definition:

Definition

Given a function $f\colon 2^{[n]} \to \mathbf{Z}$, the **Möbius transform** of f is the function $\mu(f)\colon 2^{[n]} \to \mathbf{Z}$ defined as follows: for all $S \subseteq [n]$,

$$\mu(f)(S) = \sum_{X \subseteq S} (-1)^{|S \setminus X|} f(X).$$

The previous lemma showed that $\mu(\zeta(f)) = f$. One can also show that $\zeta(\mu(f)) = f$. Hence

Lemma

The zeta and Möbius transforms are mutually inverse functions.

Computing the Zeta and Möbius Transforms

Lemma

Let $f : 2^{[n]} \to \mathbf{Z}$ s.t. for all $S \subseteq [n]$ we have $|f(S)| \leq m$ for some $m \in \mathbf{N}$. If f can be computed in time $O^*(2^n \cdot \log m)$ then $\zeta(f)$ and $\mu(f)$ can be computed in time $O^*(2^n \cdot \log m)$.

Proof

Compute $f(S)$ for all $S \subseteq [n]$ and store them in a table. By hypothesis, this takes time $O^*(2^n \cdot \log m)$ and space $O^*(2^n \cdot \log m)$.

- For $0 \leq i \leq n$ define

$$\zeta^{(i)}(f)(S) = \sum_{\substack{X \subseteq S \\ S \setminus [i] \subseteq X}} f(X).$$

- Note that $\zeta^{(n)}(f)(S) = \zeta(f)(S)$ and $\zeta^{(0)}(f)(S) = f(S)$.

Lemma

Let $f\colon 2^{[n]} \to \mathbf{Z}$ s.t. for all $S \subseteq [n]$ we have $|f(S)| \leq m$ for some $m \in \mathbf{N}$. If f can be computed in time $O^*(2^n \cdot \log m)$ then $\zeta(f)$ and $\mu(f)$ can be computed in time $O^*(2^n \cdot \log m)$.

Proof (continued)

Also note that if $i \in S$ then

$$\zeta^{(i)}(f)(S) = \sum_{\substack{X \subseteq S \\ S\setminus[i] \subseteq X}} f(X) = \sum_{\substack{X \subseteq S\setminus i \\ (S\setminus i)\setminus[i-1] \subseteq X}} f(X) + \sum_{\substack{X \subseteq S \\ S\setminus[i-1] \subseteq X}} f(X).$$

The first term on the RHS sums over all X that do not contain i while the second term sums over all X containing i.

Computing the Zeta and Möbius Transforms

Lemma

Let $f : 2^{[n]} \to \mathbf{Z}$ s.t. for all $S \subseteq [n]$ we have $|f(S)| \leq m$ for some $m \in \mathbf{N}$. If f can be computed in time $O^*(2^n \cdot \log m)$ then $\zeta(f)$ and $\mu(f)$ can be computed in time $O^*(2^n \cdot \log m)$.

Proof (continued)
Hence for $i \geq 1$,

$$\zeta^{(i)}(f)(S) = \begin{cases} \zeta^{(i-1)}(f)(S) & \text{if } i \notin S; \\ \zeta^{(i-1)}(f)(S \setminus i) + \zeta^{(i-1)}(f)(S) & \text{if } i \in S. \end{cases}$$

Hence the total time taken to compute $\zeta^{(n)}(f)(S)$ for all $S \subseteq [n]$ is $O^*(2^n \cdot \log m)$.

Computing the Zeta and Möbius Transforms . . .

Lemma

Let $f : 2^{[n]} \to \mathbf{Z}$ s.t. for all $S \subseteq [n]$ we have $|f(S)| \leq m$ for some $m \in \mathbf{N}$. If f can be computed in time $O^*(2^n \cdot \log m)$ then $\zeta(f)$ and $\mu(f)$ can be computed in time $O^*(2^n \cdot \log m)$.

Proof (For the Möbius transform)
Compute $f(S)$ for all $S \subseteq [n]$ and store them in a table. By hypothesis, this takes time $O^*(2^n \cdot \log m)$ and space $O^*(2^n \cdot \log m)$.

- For $0 \leq i \leq n$ define

$$\mu^{(i)}(f)(S) = \sum_{\substack{X \subseteq S \\ S \setminus [i] \subseteq X}} (-1)^{|S \setminus X|} f(X).$$

- Note that $\mu^{(n)}(f)(S) = \mu(f)(S)$ and $\mu^{(0)}(f)(S) = f(S)$.

Lemma

Let $f : 2^{[n]} \to \mathbf{Z}$ s.t. for all $S \subseteq [n]$ we have $|f(S)| \leq m$ for some $m \in \mathbf{N}$. If f can be computed in time $O^*(2^n \cdot \log m)$ then $\zeta(f)$ and $\mu(f)$ can be computed in time $O^*(2^n \cdot \log m)$.

Proof (For the Möbius transform, continued)
Similar to the recurrence for $\zeta^{(i)}(f)(S)$, we obtain the following recurrence for $\mu^{(i)}(f)(S)$ where $i \geq 1$,

$$\mu^{(i)}(f)(S) = \begin{cases} \mu^{(i-1)}(f)(S), & \text{if } i \notin S; \\ -\mu^{(i-1)}(f)(S \setminus i) + \mu^{(i-1)}(f)(S), & \text{if } i \in S. \end{cases}$$

Thus the total time taken to compute $\mu(f)(S)$ for all $S \subseteq [n]$ is $O^*(2^n \cdot \log m)$.

Ranked Zeta and Möbius Transforms

For the fast subset convolution algorithm, we first need fast algorithms for the *ranked versions* of the zeta and Möbius transforms.

Definition

Given a function $f : 2^{[n]} \to \mathbf{Z}$, the **ranked zeta transform** of f is a function $\zeta_k(f)$ from $2^{[n]}$ to $\mathbf{Z}$ defined for $0 \leq k \leq n$ as follows:

$$\zeta_k(f)(S) = \sum_{\substack{X \subseteq S \\ |X|=k}} f(X).$$

The Ranked Zeta Transform

Note that

- If $|S| < k$ then $\zeta_k(f)(S) = 0$.
- If we know $\zeta_k(f)$ for all k then one can obtain the zeta transform of f easily as follows:

$$\zeta(f)(S) = \sum_{k=0}^{|S|} \zeta_k(f)(S).$$

- If we are given the ranked zeta transform of f then one can compute f as follows:

$$\begin{aligned} f(S) &= \sum_{X \subseteq S} (-1)^{|S \setminus X|} \zeta(f)(X) \\ &= \sum_{X \subseteq S} (-1)^{|S \setminus X|} \sum_{k=0}^{|X|} \zeta_k(f)(X). \end{aligned}$$

The Ranked Möbius Transform

Definition

Given a function $f : 2^{[n]} \to \mathbf{Z}$, the **ranked Möbius transform** of f is a function $\mu_k(f)$ from $2^{[n]}$ to $\mathbf{Z}$ defined for $0 \leq k \leq n$ as follows:

$$\mu_k(f)(S) = \sum_{\substack{X \subseteq S \\ |X|=k}} (-1)^{|S \setminus X|} f(X).$$

The Ranked Möbius Transform

Note that

- If $|S| < k$ then $\mu_k(f)(S) = 0$.
- One can obtain the Möbius transform of f from the ranked Möbius transform as follows:

$$\mu(f)(S) = \sum_{k=0}^{|S|} \mu_k(f)(S).$$

- Given $\mu_k(f)$ for all k, one can obtain f as follows:

$$\begin{aligned} f(S) &= \sum_{X \subseteq S} \mu(f)(X) \\ &= \sum_{X \subseteq S} \sum_{k=0}^{|X|} \mu_k(f)(X). \end{aligned}$$

Computing the Ranked Zeta and Möbius Transforms

Lemma

Let $f : 2^{[n]} \to \mathbf{Z}$ s.t. for all $S \subseteq [n]$ we have $|f(S)| \leq m$ for some $m \in \mathbf{N}$. If f can be computed in time $O^*(2^n \cdot \log m)$ then the ranked zeta and Möbius transforms of f can be computed in time $O^*(2^n \cdot \log m)$.

Proof

This proof proceeds along the same lines as the one for computing the (unranked) zeta and Möbius transforms and is left as an exercise.

Fast Subset Convolution

In the design of the fast subset convolution algorithm, the main idea is to use a *ranked convolution* of ranked zeta transforms.

Definition

Let $\tilde{\zeta}(f)$ and $\tilde{\zeta}(g)$ be the ranked zeta transforms of functions f and g. The **ranked convolution** of $\tilde{\zeta}(f)$ and $\tilde{\zeta}(g)$, denoted by $(\tilde{\zeta}(f) \circledast \tilde{\zeta}(g))_k$, is defined for $0 \leq k \leq n$ and all $S \subseteq [n]$ by

$$(\tilde{\zeta}(f) \circledast \tilde{\zeta}(g))_k(S) = \sum_{i=0}^{k} \zeta_i(f)(S) \cdot \zeta_{k-i}(g)(S).$$

Note

The ranked convolution can be computed in time $O^*(2^n)$ if the ranked zeta transforms of f and g are known.

Convolutions and Ranked Convolutions

Here's the connection between the convolution of two functions and the ranked convolutions of their ranked zeta transforms.

Theorem

Let $f : 2^U \to \mathbf{Z}$ and let $\tilde{\zeta}(f)$ and $\tilde{\zeta}(g)$ be their ranked zeta transforms. Then $f * g$ is the Möbius transform of $\tilde{\zeta}(f) \circledast \tilde{\zeta}(g)$. That is, for all $S \subseteq U$,

$$(f * g)(S) = \sum_{X \subseteq S} (-1)^{|S \setminus X|} (\tilde{\zeta}(f) \circledast \tilde{\zeta}(g))_{|S|}(X).$$

Implications of the Theorem

Since

$$(f * g)(S) = \sum_{X \subseteq S} (-1)^{|S \setminus X|} (\tilde{\zeta}(f) \circledast \tilde{\zeta}(g))_{|S|}(X),$$

one can compute the convolution $f * g$ as follows.

Assume that there exists $m \in \mathbf{N}$ s.t. $|f(S)|, |g(S)| \leq m$ for all $S \subseteq U$.

- First compute the ranked zeta transforms of f and g in time $O^*(2^n \cdot \log m)$.
- Next compute the ranked convolution of the ranked zeta transforms of f and g in time $O^*(2^n)$.
- Finally compute the Möbius transform of the function $\tilde{\zeta}(f) \circledast \tilde{\zeta}(g)$ in time $O^*(2^n \cdot \log m)$.

Thus the convolution of f and g can be computed in time $O^*(2^n \cdot \log m)$.

Proof of Theorem

Theorem

Let $f : 2^U \to \mathbf{Z}$ and let $\tilde{\zeta}(f)$ and $\tilde{\zeta}(g)$ be their ranked zeta transforms. Then $f * g$ is the Möbius transform of $\tilde{\zeta}(f) \circledast \tilde{\zeta}(g)$. That is, for all $S \subseteq U$,

$$(f * g)(S) = \sum_{X \subseteq S} (-1)^{|S \setminus X|} (\tilde{\zeta}(f) \circledast \tilde{\zeta}(g))_{|S|}(X).$$

Proof

Let $|S| = k$. By writing out the expression for the ranked convolution on the rhs, we obtain the following expression:

$$\sum_{X \subseteq S} (-1)^{|S \setminus X|} \sum_{i=0}^{k} \zeta_i(f)(X) \cdot \zeta_{k-i}(f)(X). \tag{1}$$

Proof of Theorem (continued) . . .

Expression 1 may be written as

$$\sum_{X \subseteq S} (-1)^{|S \setminus X|} \sum_{i=0}^{k} \left(\sum_{\substack{Y \subseteq X \\ |Y|=i}} f(Y) \right) \left(\sum_{\substack{Z \subseteq X \\ |Z|=k-i}} g(Z) \right)$$

$$= \sum_{X \subseteq S} (-1)^{|S \setminus X|} \sum_{i=0}^{k} \sum_{\substack{Y,Z \subseteq X \\ |Y|=i;|Z|=k-i}} f(Y)g(Z)$$

The last expression may be written as

$$\sum_{i=0}^{k} \sum_{\substack{Y,Z \subseteq S \\ |Y|=i;|Z|=k-i}} \left(\sum_{\substack{X \subseteq S \\ Y \cup Z \subseteq X}} (-1)^{|S \setminus X|} \right) f(Y)g(Z). \tag{2}$$

Proof of Theorem (continued) . . .

In expression 2

$$\sum_{i=0}^{k} \sum_{\substack{Y,Z\subseteq S \\ |Y|=i;|Z|=k-i}} \left(\sum_{\substack{X\subseteq S \\ Y\cup Z\subseteq X}} (-1)^{|S\setminus X|} \right) f(Y)g(Z),$$

the outer sum is over all subsets Y, Z of S such that $|Y| = i$ and $|Z| = k - i$; the inner sum is over all X s.t.

$$Y \cup Z \subseteq X \subseteq S.$$

Let $|Y \cup Z| = p$. Since $Y \cup Z \subseteq X \subseteq S$, we have that $p \leq |X| \leq k$ and $0 \leq |S \setminus X| \leq k - p$.

Proof of Theorem (continued) . . .

Consider the case when $k - p > 0$. The number of sets X of size $p + j$ that satisfy $Y \cup Z \subseteq X \subseteq S$ is

$$\binom{k-p}{j} = \binom{k-p}{k-p-j}.$$

Hence the inner sum equals

$$\sum_{i=0}^{k-p} (-1)^i \binom{k-p}{i} = (1-1)^{k-p} = 0.$$

If $k - p = 0$, then $Y \cup Z = S$ and Y, Z form a bipartition of S. The inner sum is 1 in this case and expression 2 works out to

$$\sum_{\substack{Y \cup Z = S \\ Y \cap Z = \emptyset}} f(Y)g(Z) = (f * g)(S).$$

This completes the proof.

Final Stand: Some Applications

We now know that the convolution of two functions $f, g \colon 2^U \to \mathbf{Z}$ can be computed in time $O^*(2^{|U|} \cdot \log m)$, where m is a positive integer s.t. $|f(S)|, |g(S)| \leq m$ for all $S \subseteq U$.

We saw that

- k-Chromatic Number,
- k-Partition,
- k-Domatic Partition, etc.

may be expressed as (k iterated) convolutions of a function $f \colon 2^V \to \{0, 1\}$ from vertex subsets of a graph to the set $\{0, 1\}$.

Hence all these problems can all be solved in $O^*(2^n \cdot \log k)$ time.