Algebraic approach to exact algorithms

Łukasz Kowalik

University of Warsaw

Będlewo, 21.09.2012

Introduction

Exact algorithms for NP-hard problems: motivation

Ways of coping with NP-hardness

- Approximation (for optimization problems),
- Restricted inputs,
- Heuristics

Exact algorithms for NP-hard problems: motivation

Ways of coping with NP-hardness

- Approximation (for optimization problems),
- Restricted inputs,
- ~~Heuristics~~

Exact algorithms for NP-hard problems: motivation

Ways of coping with NP-hardness

- Approximation (for optimization problems),
- Restricted inputs,
- ~~Heuristics~~

Unfortunately these methods have limitations

- Many important problems do not approximate well, unless $\mathbf{P} \neq \mathbf{NP}$ (e.g. TSP, coloring, clique)
- Sometimes we have to solve an instance which is not restricted

Exact algorithms for NP-hard problems: motivation

Ways of coping with NP-hardness

- Approximation (for optimization problems),
- Restricted inputs,
- ~~Heuristics~~

And even if they work, they offer a compromise:

They are fast, but

- not exact,
- fast only for special instances,
- ~~you never know what exactly your heuristics returns~~

This tutorial is on

Algorithms with no compromises

given an NP-hard problem we want to solve it and we aim at the best possible **asymptotic worst-case** time (for general instances).

Motivation, cont'd

- We will investigate how much we can improve over the naive algorithm for the problem.

Motivation, cont'd

- We will investigate how much we can improve over the naive algorithm for the problem.
- **Goal:** give an algorithm of $O(c^n)$ time complexity, for c as small as possible.

Motivation, cont'd

- We will investigate how much we can improve over the naive algorithm for the problem.
- **Goal:** give an algorithm of $O(c^n)$ time complexity, for c as small as possible.
- If instead of $O(2^n)$-time algorithm we use a $O(1.189^n) = O(2^{n/4})$-time algorithm, it means (roughly) that using the same machine we can solve instances 4 **times** bigger. Note that accelerating the processor 16-times means (roughly), that we can solve instances with n bigger by 4.

~~Absurds~~ Properties of asymptotic notation

- $(n+m)2^n = o(2.0001^n)$,

~~Absurds~~ Properties of asymptotic notation

- $(n+m)2^n = o(2.0001^n)$,
- $n^{100}2^n = o(2.0001^n)$,

~~Absurds~~ Properties of asymptotic notation

- $(n+m)2^n = o(2.0001^n)$,
- $n^{100}2^n = o(2.0001^n)$,
- $n^{\log n}2^n = o(2.0001^n)$.

~~Absurds~~ Properties of asymptotic notation

- $(n+m)2^n = o(2.0001^n)$,
- $n^{100}2^n = o(2.0001^n)$,
- $n^{\log n}2^n = o(2.0001^n)$.

Motivated by the above we introduce the following notation:

Definition

$f(n) = O^*(g(n))$, when $f(n) = p(n)g(n)$ for some polynomial p.

E.g. $(n+m)2^n = O^*(2^n)$, $n^{100}2^n = O^*(2^n)$.

Agenda

In this tutorial I focus on algebraic approaches.
We will discuss

1. Algorithms based on Fast Matrix Multiplication,
2. Algorithms based on Inclusion-Exclusion principle,
3. Algorithms based on Schwartz-Zippel lemma.

Part I: Fast Matrix Multiplication

(Square) matrix multiplication

Problem

Given two matrices $n \times n$: A and B.
Compute the matrix $C = A \cdot B$.

Naive algorithm

$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$.
Time: $O(n^3)$ arithmetical operations.

Matrix multiplication: Divide and conquer (1)

W.l.o.g. $n = 2^k$.
Let us partition **A**, **B**, **C** into blocks of size $(n/2) \times (n/2)$:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}$$

Then

$$\mathbf{C} = \left[\begin{array}{c|c} \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} & \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2} \\ \hline \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1} & \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2} \end{array}\right]$$

We get the recurrence $T(n) = 8T(n/2) + O(n^2)$, hence $T(n) = O(n^3)$.
(The last level dominates, it has $8^{\log_2 n} = n^3$ nodes.)

Matrix multiplication: Divide and conquer (2)

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix}$$

A new approach (Strassen 1969):

$$\mathbf{M}_1 := (\mathbf{A}_{1,1} + \mathbf{A}_{2,2})(\mathbf{B}_{1,1} + \mathbf{B}_{2,2}) \qquad \mathbf{M}_2 := (\mathbf{A}_{2,1} + \mathbf{A}_{2,2})\mathbf{B}_{1,1}$$

$$\mathbf{M}_3 := \mathbf{A}_{1,1}(\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \qquad \mathbf{M}_4 := \mathbf{A}_{2,2}(\mathbf{B}_{2,1} - \mathbf{B}_{1,1})$$

$$\mathbf{M}_5 := (\mathbf{A}_{1,1} + \mathbf{A}_{1,2})\mathbf{B}_{2,2} \qquad \mathbf{M}_6 := (\mathbf{A}_{2,1} - \mathbf{A}_{1,1})(\mathbf{B}_{1,1} + \mathbf{B}_{1,2})$$

$$\mathbf{M}_7 := (\mathbf{A}_{1,2} - \mathbf{A}_{2,2})(\mathbf{B}_{2,1} + \mathbf{B}_{2,2}).$$

Then:

$$\mathbf{C} = \left[\begin{array}{c|c} \mathbf{A}_{1,1}\mathbf{B}_{1,1} + \mathbf{A}_{1,2}\mathbf{B}_{2,1} & \mathbf{A}_{1,1}\mathbf{B}_{1,2} + \mathbf{A}_{1,2}\mathbf{B}_{2,2} \\ \hline \mathbf{A}_{2,1}\mathbf{B}_{1,1} + \mathbf{A}_{2,2}\mathbf{B}_{2,1} & \mathbf{A}_{2,1}\mathbf{B}_{1,2} + \mathbf{A}_{2,2}\mathbf{B}_{2,2} \end{array}\right]$$

$$= \left[\begin{array}{c|c} \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 & \mathbf{M}_3 + \mathbf{M}_5 \\ \hline \mathbf{M}_2 + \mathbf{M}_4 & \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6 \end{array}\right]$$

We get the recurrence $T(n) = 7T(n/2) + O(n^2)$ hence $T(n) = O(7^{\log_2 n}) = O(n^{\log_2 7}) = O(n^{2.81})$.

A few facts

Let $M(n)$ be the time needed to multiply two matrices $n \times n$.
We know that

- $M(n) = O(n^{\omega})$, where $\omega < 2.38$ (Coppersmith and Winograd 1990, Vassilevska-Williams 2011).
- One can invert a matrix in $O(M(n))$ time (Bunch and Hopcroft).
- One can compute the determinant of a matrix in $O(M(n))$ time (Bunch and Hopcroft).

A standard exercise

Problem

Given a directed/undirected n-vertex graph G

- find a triangle in G, if it exists.
- Compute the number of triangles in G

MAX-2-SAT

Problem MAX-2-SAT

Given a 2-CNF formula ϕ with n variables, find an assignment which maximizes the number of satisfied clauses.

Example: $(x_1 \vee \neg x_2) \wedge (x_3 \vee x_2) \wedge (x_2 \vee \neg x_5) \wedge \cdots$

MAX-2-SAT

Problem MAX-2-SAT

Given a 2-CNF formula ϕ with n variables, find an assignment which maximizes the number of satisfied clauses.

Example: $(x_1 \vee \neg x_2) \wedge (x_3 \vee x_2) \wedge (x_2 \vee \neg x_5) \wedge \cdots$

In what follows we deal with the equivalent (up to a #clauses factor) problem:

MAX-2-SAT, decision version

Input: A 2-CNF formula ϕ with n variables, a number $k \in \mathbb{N}$.

Question: Is there an assignment which satisfies exactly k clauses?

MAX-2-SAT

Problem MAX-2-SAT

Given a 2-CNF formula ϕ with n variables, find an assignment which maximizes the number of satisfied clauses.

Example: $(x_1 \vee \neg x_2) \wedge (x_3 \vee x_2) \wedge (x_2 \vee \neg x_5) \wedge \cdots$
In what follows we deal with the equivalent (up to a #clauses factor) problem:

MAX-2-SAT, decision version

Input: A 2-CNF formula ϕ with n variables, a number $k \in \mathbb{N}$.
Question: Is there an assignment which satisfies exactly k clauses?

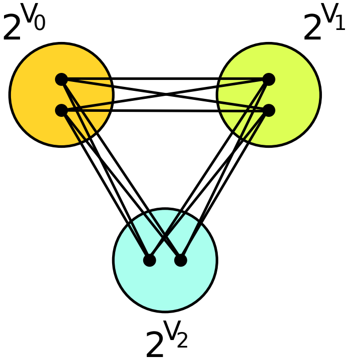
Complexity

MAX-2-SAT is NP-complete.
The naive algorithm works in $O^*(2^n)$ time.
Question: Can we do better? E.g. $O(1.9^n)$?

MAX-2-SAT (Williams 2004)

We construct an undirected graph G on $O(2^{n/3})$ vertices.

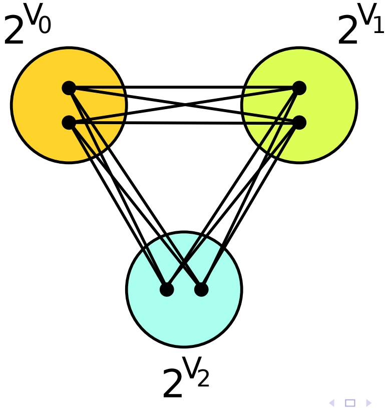
- Let us fix an arbitrary partition $V = V_0 \cup V_1 \cup V_2$ into three equal parts (as equal as possible...).
- $V(G)$ is the set of all assignments $v_i : V_i \to \{0,1\}$ for $i = 0, 1, 2$.
- For every $v \in V_i$, $w \in V_{(i+1) \bmod 3}$ graph G contains the edge vw.



MAX-2-SAT (Williams 2004)

Solution idea

- We assign weights to edges so that the weight of the vwu triangle in G equals the number of clauses satisfied with the assignment (v, w, u).
- Then it is sufficient to check if there is a triangle of weight k in G.



MAX-2-SAT (Williams 2004)

Solution idea

- We assign weights to edges so that the weight of the vwu triangle in G equals the number of clauses satisfied with the assignment (v, w, u).
- Then it is sufficient to check if there is a triangle of weight k in G.

Problem 1 How should we assign weights?
Let $c(v)$ = all the clauses satisfied under the (partial) assignment v.
Then the number of clauses satisfied under the assignment (v, w, u) amounts to:

$$|c(v) \cup c(w) \cup c(u)| = |c(v)| + |c(w)| + |c(u)| \\ - |c(v) \cap c(w)| - |c(v) \cap c(u)| - |c(w) \cap c(u)| \\ + |c(v) \cap c(w) \cap c(u)|.$$

MAX-2-SAT (Williams 2004)

Solution idea

- We assign weights to edges so that the weight of the vwu triangle in G equals the number of clauses satisfied with the assignment (v, w, u).
- Then it is sufficient to check if there is a triangle of weight k in G.

Problem 1 How should we assign weights?
Let $c(v)$ = all the clauses satisfied under the (partial) assignment v.
Then the number of clauses satisfied under the assignment (v, w, u) amounts to:

$$\begin{aligned}|c(v) \cup c(w) \cup c(u)| = & |c(v)| + |c(w)| + |c(u)| \\ & - |c(v) \cap c(w)| - |c(v) \cap c(u)| - |c(w) \cap c(u)| \\ & + \underbrace{|c(v) \cap c(w) \cap c(u)|}_{0}.\end{aligned}$$

MAX-2-SAT (Williams 2004)

Solution idea

- We assign weights to edges so that the weight of the *vwu* triangle in G equals the number of clauses satisfied with the assignment (v, w, u).
- Then it is sufficient to check if there is a triangle of weight k in G.

Problem 1 How should we assign weights?
Let $c(v)$ = all the clauses satisfied under the (partial) assignment v.
Then the number of clauses satisfied under the assignment (v, w, u) amounts to:

$$|c(v) \cup c(w) \cup c(u)| = |c(v)| + |c(w)| + |c(u)| - |c(v) \cap c(w)| - |c(w) \cap c(u)| - |c(u) \cap c(v)| + \underbrace{|c(v) \cap c(w) \cap c(u)|}_{0}.$$

So, we put $\text{weight}(xy) = |c(x)| - |c(x) \cap c(y)|$.

MAX-2-SAT (Williams 2004)

We are left with verifying whether there is a triangle of weight k in G.

A trick

Consider all $O(m^2) = O(n^4)$ partitions ($m =$ the number of clauses) $k = k_0 + k_1 + k_2$. For every partition we build a graph G_{k_0,k_1,k_2} which consists only of:

- edges of weight k_0 between 2^{V_0} and 2^{V_1},
- edges of weight k_1 between 2^{V_1} and 2^{V_2},
- edges of weight k_2 between 2^{V_2} and 2^{V_0},

Then it suffices to...

MAX-2-SAT (Williams 2004)

We are left with verifying whether there is a triangle of weight k in G.

A trick

Consider all $O(m^2) = O(n^4)$ partitions (m = the number of clauses) $k = k_0 + k_1 + k_2$. For every partition we build a graph G_{k_0,k_1,k_2} which consists only of:

- edges of weight k_0 between 2^{V_0} and 2^{V_1},
- edges of weight k_1 between 2^{V_1} and 2^{V_2},
- edges of weight k_2 between 2^{V_2} and 2^{V_0},

Then it suffices to... check whether there is a triangle.

Checking whether G_{k_0,k_1,k_2} contains a triangle

Corollary

- Graph G_{k_0,k_1,k_2} has $3 \cdot 2^{n/3}$ vertices.
- We can verify whether G_{k_0,k_1,k_2} contains a triangle in $O(2^{\omega n/3}) = O(1.732^n)$ time and $O(2^{2/3n})$ space.
- Hence we can check whether G contains a triangle of weight k in $O(n^4 \cdot 2^{\omega n/3}) = O(n^4 \cdot 1.732^n) = O(1.733^n)$ time.

MAX-2-SAT (Williams 2004): Conclusion

Corollary

There is an algorithm for MAX-2-SAT running in $O^*(1.733^n)$ time and $O(2^{2/3n})$ space.

MAX-2-SAT (Williams 2004): Conclusion

Corollary

There is an algorithm for MAX-2-SAT running in $O^*(1.733^n)$ time and $O(2^{2/3n})$ space.

It is easy to modify the algorithm (how?) to get

Corollary

There is an algorithm which counts the number of optimum MAX-2-SAT solutions running in $O^*(1.733^n)$ time and $O(2^{2/3n})$ space.

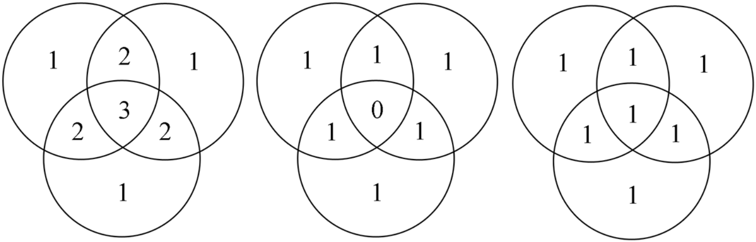
Part II: Inclusion-Exclusion

Inclusion-Exclusion Principle

Twierdzenie (Inclusion-Exclusion Principle, version I)

$$|\bigcup_{i\in\{1,\ldots,n\}} A_i| = \sum_{\emptyset\neq X\subseteq\{1,\ldots,n\}} (-1)^{|X|-1}|\bigcap_{i\in X} A_i|$$

e.g. $|A\cup B| = |A| + |B| - |A\cap B|$,
$|A\cup B\cup C| = |A| + |B| + |C| - |A\cap B| - |B\cap C| - |A\cap C| + |A\cap B\cap C|$.



$|A|+|B|+|C|$

$|A|+|B|+|C|$
$-(|A\cap B|+|A\cap C|+|B\cap C|)$

$|A|+|B|+|C|$
$-(|A\cap B|+|A\cap C|+|B\cap C|)$
$+|A\cap B\cap C|$

Inclusion-Exclusion Principle, rewriting

Let $A_1, \ldots, A_n \subseteq U$, where U is a finite set.

$$|\bigcup_{i\in\{1,\ldots,n\}} A_i| = \sum_{\emptyset\neq X\subseteq\{1,\ldots,n\}} (-1)^{|X|-1} |\bigcap_{i\in X} A_i|$$

Inclusion-Exclusion Principle, rewriting

Let $A_1, \ldots, A_n \subseteq U$, where U is a finite set.

$$|\bigcup_{i\in\{1,\ldots,n\}} A_i| = \sum_{\emptyset\neq X\subseteq\{1,\ldots,n\}} (-1)^{|X|-1} |\bigcap_{i\in X} A_i|$$

$$|U| - |\bigcup_{i\in\{1,\ldots,n\}} A_i| = |U| - \sum_{\emptyset\neq X\subseteq\{1,\ldots,n\}} (-1)^{|X|-1} |\bigcap_{i\in X} A_i|$$

Inclusion-Exclusion Principle, rewriting

Let $A_1, \ldots, A_n \subseteq U$, where U is a finite set.

$$|\bigcup_{i\in\{1,\ldots,n\}} A_i| = \sum_{\emptyset\neq X\subseteq\{1,\ldots,n\}} (-1)^{|X|-1} |\bigcap_{i\in X} A_i|$$

$$|U| - |\bigcup_{i\in\{1,\ldots,n\}} A_i| = |U| - \sum_{\emptyset\neq X\subseteq\{1,\ldots,n\}} (-1)^{|X|-1} |\bigcap_{i\in X} A_i|$$

$$|U - \bigcup_{i\in\{1,\ldots,n\}} A_i| = |U| - \sum_{\emptyset\neq X\subseteq\{1,\ldots,n\}} (-1)^{|X|-1} |\bigcap_{i\in X} A_i|$$

Inclusion-Exclusion Principle, rewriting

Let $A_1, \ldots, A_n \subseteq U$, where U is a finite set.

$$|\bigcup_{i\in\{1,\ldots,n\}} A_i| = \sum_{\emptyset\neq X\subseteq\{1,\ldots,n\}} (-1)^{|X|-1} |\bigcap_{i\in X} A_i|$$

$$|U| - |\bigcup_{i\in\{1,\ldots,n\}} A_i| = |U| - \sum_{\emptyset\neq X\subseteq\{1,\ldots,n\}} (-1)^{|X|-1} |\bigcap_{i\in X} A_i|$$

$$|U - \bigcup_{i\in\{1,\ldots,n\}} A_i| = |U| - \sum_{\emptyset\neq X\subseteq\{1,\ldots,n\}} (-1)^{|X|-1} |\bigcap_{i\in X} A_i|$$

Denote $\overline{A_i} = U - A_i$ and $\bigcap_{i\in\emptyset} \overline{A_i} = U$. Then:

$$|\bigcap_{i\in\{1,\ldots,n\}} \overline{A_i}| = \sum_{X\subseteq\{1,\ldots,n\}} (-1)^{|X|} |\bigcap_{i\in X} A_i|$$

Inclusion-Exclusion Principle, rewriting

Let $A_1, \ldots, A_n \subseteq U$, where U is a finite set.

$$|\bigcup_{i\in\{1,\ldots,n\}} A_i| = \sum_{\emptyset\neq X\subseteq\{1,\ldots,n\}} (-1)^{|X|-1} |\bigcap_{i\in X} A_i|$$

$$|U| - |\bigcup_{i\in\{1,\ldots,n\}} A_i| = |U| - \sum_{\emptyset\neq X\subseteq\{1,\ldots,n\}} (-1)^{|X|-1} |\bigcap_{i\in X} A_i|$$

$$|U - \bigcup_{i\in\{1,\ldots,n\}} A_i| = |U| - \sum_{\emptyset\neq X\subseteq\{1,\ldots,n\}} (-1)^{|X|-1} |\bigcap_{i\in X} A_i|$$

Denote $\overline{A_i} = U - A_i$ and $\bigcap_{i\in\emptyset} \overline{A_i} = U$. Then:

$$|\bigcap_{i\in\{1,\ldots,n\}} \overline{A_i}| = \sum_{X\subseteq\{1,\ldots,n\}} (-1)^{|X|} |\bigcap_{i\in X} A_i|$$

$$|\bigcap_{i\in\{1,\ldots,n\}} A_i| = \sum_{X\subseteq\{1,\ldots,n\}} (-1)^{|X|} |\bigcap_{i\in X} \overline{A_i}|$$

Inclusion-Exclusion Principle, intersection version

We get:

Twierdzenie (Inclusion-Exclusion Principle, intersection version)

Let $A_1, \ldots, A_n \subseteq U$, where U is a finite set.
Denote $\overline{A_i} = U - A_i$ and $\bigcap_{i\in\emptyset} \overline{A_i} = U$.
Then:

$$|\bigcap_{i\in\{1,\ldots,n\}} A_i| = \sum_{X\subseteq\{1,\ldots,n\}} (-1)^{|X|} \underbrace{|\bigcap_{i\in X} \overline{A_i}|}_{\text{``simplified problem''}}$$

A classic example: derangements

Task

Permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ is a derangement, when $\pi(i) \neq i$ for each $i = 1, \ldots, n$.
Find a formula for $d(n)$, the number of n-element derangements.

- U is a set of n-element permutations.

A classic example: derangements

Task

Permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ is a derangement, when $\pi(i) \neq i$ for each $i = 1, \ldots, n$.
Find a formula for $d(n)$, the number of n-element derangements.

- U is a set of n-element permutations.
- For $i = 1, \ldots, n$ we define $A_i = \{\pi \in U \ : \ \pi(i) \neq i\}$. „requirements”

A classic example: derangements

Task

Permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ is a derangement, when $\pi(i) \neq i$ for each $i = 1, \ldots, n$.
Find a formula for $d(n)$, the number of n-element derangements.

- U is a set of n-element permutations.
- For $i = 1, \ldots, n$ we define $A_i = \{\pi \in U \ : \ \pi(i) \neq i\}$. „requirements"
- Then $d(n) = |\bigcap_{i=1,\ldots,n} A_i|$.

A classic example: derangements

Task

Permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ is a derangement, when $\pi(i) \neq i$ for each $i = 1, \ldots, n$.
Find a formula for $d(n)$, the number of n-element derangements.

- U is a set of n-element permutations.
- For $i = 1, \ldots, n$ we define $A_i = \{\pi \in U \ : \ \pi(i) \neq i\}$. „requirements"
- Then $d(n) = |\bigcap_{i=1,\ldots,n} A_i|$.
- $|\bigcap_{i \in X} \overline{A_i}| = (n - |X|)!$. „simplified problem"

A classic example: derangements

Task

Permutation $\pi : \{1, \dots, n\} \to \{1, \dots, n\}$ is a derangement, when $\pi(i) \neq i$ for each $i = 1, \dots, n$.
Find a formula for $d(n)$, the number of n-element derangements.

- U is a set of n-element permutations.
- For $i = 1, \dots, n$ we define $A_i = \{\pi \in U \ : \ \pi(i) \neq i\}$. „requirements”
- Then $d(n) = |\bigcap_{i=1,\dots,n} A_i|$.
- $|\bigcap_{i \in X} \overline{A_i}| = (n - |X|)!$. „simplified problem”

Corollary

$$d(n) = \sum_{X \subseteq \{1,\dots,n\}} (-1)^{|X|}(n - |X|)! = \sum_{i=1}^{n} (-1)^i \binom{n}{i} (n - i)!$$

A toy algorithmic example

Problem

Given a CNF-formula with m clauses, compute the number of satisfying assignments.

Example: $(x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_5) \wedge \cdots$

A toy algorithmic example

Problem

Given a CNF-formula with m clauses, compute the number of satisfying assignments.

Example: $(x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_5) \wedge \cdots$

- U is a set of all assignments.

A toy algorithmic example

Problem

Given a CNF-formula with m clauses, compute the number of satisfying assignments.

Example: $(x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_5) \wedge \cdots$

- U is a set of all assignments.
- A_i = the set of assignments with clause C_i satisfied, $i = 1, \ldots, m$.

A toy algorithmic example

Problem

Given a CNF-formula with m clauses, compute the number of satisfying assignments.

Example: $(x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_5) \wedge \cdots$

- U is a set of all assignments.
- $A_i =$ the set of assignments with clause C_i satisfied, $i = 1, \ldots, m$.
- Then the solution is $|\bigcap_{i=1,\ldots,n} A_i|$.

A toy algorithmic example

Problem

Given a CNF-formula with m clauses, compute the number of satisfying assignments.

Example: $(x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_5) \wedge \cdots$

- U is a set of all assignments.
- A_i = the set of assignments with clause C_i satisfied, $i = 1, \ldots, m$.
- Then the solution is $|\bigcap_{i=1,\ldots,n} A_i|$.
- $|\bigcap_{i \in X} \overline{A_i}| =$
$$\begin{cases} 0 & \text{when } X \text{ contains two (numbers of) clauses with opposite literals,} \\ 2^v & \text{where } v \text{ is the number of variables outside clauses from } X \end{cases}$$

A toy algorithmic example

Problem

Given a CNF-formula with m clauses, compute the number of satisfying assignments.

Example: $(x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee \neg x_5) \wedge \cdots$

- U is a set of all assignments.
- A_i = the set of assignments with clause C_i satisfied, $i = 1, \ldots, m$.
- Then the solution is $|\bigcap_{i=1,\ldots,n} A_i|$.
- $|\bigcap_{i \in X} \overline{A_i}| =$
$$\begin{cases} 0 & \text{when } X \text{ contains two (numbers of) clauses with opposite literals,} \\ 2^v & \text{where } v \text{ is the number of variables outside clauses from } X \end{cases}$$
- The simplified problem can be solved in polynomial (even linear) time, so we get an $O^*(2^m)$-time algorithm.

The number of Hamiltonian cycles (Karp 1982)

Hamiltonian cycle: a simple cycle that contains all the vertices.

Problem

Given an n-vertex undirected graph $G = (V, E)$ compute the number of Hamiltonian cycles.

The number of Hamiltonian cycles (Karp 1982)

Hamiltonian cycle: a simple cycle that contains all the vertices.

Problem

Given an n-vertex undirected graph $G = (V, E)$ compute the number of Hamiltonian cycles.

- A walk of length k in G (shortly, a k-walk) is a sequence of vertices $v_0, v_1, \ldots, v_k$ such that $v_i v_{i+1} \in E$ for each $i = 0, \ldots, k-1$.
- A walk is closed, when $v_0 = v_k$.

The number of Hamiltonian cycles (Karp 1982)

Hamiltonian cycle: a simple cycle that contains all the vertices.

Problem

Given an n-vertex undirected graph $G = (V, E)$ compute the number of Hamiltonian cycles.

- A walk of length k in G (shortly, a k-walk) is a sequence of vertices $v_0, v_1, \ldots, v_k$ such that $v_i v_{i+1} \in E$ for each $i = 0, \ldots, k-1$.
- A walk is closed, when $v_0 = v_k$.
- U is the set of closed n-walks from vertex 1.

The number of Hamiltonian cycles (Karp 1982)

Hamiltonian cycle: a simple cycle that contains all the vertices.

Problem

Given an n-vertex undirected graph $G = (V, E)$ compute the number of Hamiltonian cycles.

- A walk of length k in G (shortly, a k-walk) is a sequence of vertices $v_0, v_1, \ldots, v_k$ such that $v_i v_{i+1} \in E$ for each $i = 0, \ldots, k-1$.
- A walk is closed, when $v_0 = v_k$.
- U is the set of closed n-walks from vertex 1.
- A_v = the walks from U that visit v, $v \in V$.

The number of Hamiltonian cycles (Karp 1982)

Hamiltonian cycle: a simple cycle that contains all the vertices.

Problem

Given an n-vertex undirected graph $G = (V, E)$ compute the number of Hamiltonian cycles.

- A walk of length k in G (shortly, a k-walk) is a sequence of vertices $v_0, v_1, \ldots, v_k$ such that $v_i v_{i+1} \in E$ for each $i = 0, \ldots, k-1$.
- A walk is closed, when $v_0 = v_k$.
- U is the set of closed n-walks from vertex 1.
- A_v = the walks from U that visit v, $v \in V$.
- Then the solution is $|\bigcap_{v \in V} A_v|$.

The number of Hamiltonian cycles (Karp 1982)

Hamiltonian cycle: a simple cycle that contains all the vertices.

Problem

Given an n-vertex undirected graph $G = (V, E)$ compute the number of Hamiltonian cycles.

- A walk of length k in G (shortly, a k-walk) is a sequence of vertices $v_0, v_1, \ldots, v_k$ such that $v_i v_{i+1} \in E$ for each $i = 0, \ldots, k-1$.
- A walk is closed, when $v_0 = v_k$.
- U is the set of closed n-walks from vertex 1.
- $A_v =$ the walks from U that visit v, $v \in V$.
- Then the solution is $|\bigcap_{v \in V} A_v|$.
- The simplified problem: $|\bigcap_{v \in X} \overline{A_v}| =$ the number of closed walks from U in $G' = G[V - X]$.

The number of Hamiltonian cycles, cont'd

The simplified problem

Compute the number of closed n-walks in G' that start at vertex 1.

Dynamic programming

- $T(d, x) =$ the number of length d walks from 1 to x.
- $T(d, x) = \sum_{y \in V} T(d-1, y) \cdot [yx \in E(G')]$.
- We return $T(n, 1)$, DP works in $O(n^3)$ time.

The number of Hamiltonian cycles, cont'd

The simplified problem

Compute the number of closed n-walks in G' that start at vertex 1.

Dynamic programming

- $T(d, x) =$ the number of length d walks from 1 to x.
- $T(d, x) = \sum_{y \in V} T(d-1, y) \cdot [yx \in E(G')]$.
- We return $T(n, 1)$, DP works in $O(n^3)$ time.

Another approach: we return $M^n_{1,1}$, $M =$ adjacency matrix; $O(n^\omega \log n)$ time.

The number of Hamiltonian cycles, cont'd

The simplified problem

Compute the number of closed n-walks in G' that start at vertex 1.

Dynamic programming

- $T(d, x)$ = the number of length d walks from 1 to x.
- $T(d, x) = \sum_{y \in V} T(d-1, y) \cdot [yx \in E(G')]$.
- We return $T(n, 1)$, DP works in $O(n^3)$ time.

Another approach: we return $M^n_{1,1}$, M = adjacency matrix; $O(n^\omega \log n)$ time.

Corollary

We can solve the Hamiltonian Cycle problem (and even find the number of such cycles) in $O^*(2^n)$ time and **polynomial space**.

TSP

Problem

Given a weight matrix in the complete graph $w : V^2 \to \{1, \ldots, C\}$, compute the number of Hamiltonian cycles **of weight** α, $\alpha = 1, \ldots, nC$?

TSP

Problem

Given a weight matrix in the complete graph $w : V^2 \to \{1, \ldots, C\}$, compute the number of Hamiltonian cycles **of weight** α, $\alpha = 1, \ldots, nC$?

The simplified problem

Compute the number of closed n-walks **of weight** α in G' that start at vertex 1.

TSP

Problem

Given a weight matrix in the complete graph $w : V^2 \to \{1, \ldots, C\}$, compute the number of Hamiltonian cycles **of weight** α, $\alpha = 1, \ldots, nC$?

The simplified problem

Compute the number of closed n-walks **of weight** α in G' that start at vertex 1.

Dynamic programming

- let $C =$ the maximum edge weight in G'.
- $T(d, x, \beta) =$ the number of length d walks from 1 to x and of weight β, $\beta = 1, \ldots, \alpha$.
- $T(d, x, \beta) = \sum_{y \in V} T(d-1, y, \beta - w(x, y))$.
- We return $T(n, 1, \alpha)$, time $O(n^3 C)$.

TSP

Corollary

We can solve the (decision) TSP problem in $O^*(2^n \cdot C)$ time and **polynomial space**.

Corollary

We can solve the optimization TSP problem in $O^*(2^n \cdot C \log C)$ time and **polynomial space**.

Coloring in $O^*(2^n)$, Björklund, Husfeldt, Koivisto 2006

k-coloring

k-coloring of a graph $G = (V, E)$ is a function $c : V \to \{1, \ldots, k\}$ such that for every edge $xy \in E$, $c(x) \neq c(y)$.

Problem

Given a graph $G = (V, E)$ and $k \in \mathbb{N}$ decide whether there is a k-coloring of G. (If we can do it in $O^*(c^n)$ time then we can also **find** the coloring in $O^*(c^n)$ time when it exists, due to self-reducibility).

Coloring in $O^*(2^n)$, Björklund, Husfeldt, Koivisto 2006

k-coloring

k-coloring of a graph $G = (V, E)$ is a function $c : V \to \{1, \ldots, k\}$ such that for every edge $xy \in E$, $c(x) \neq c(y)$.

Problem

Given a graph $G = (V, E)$ and $k \in \mathbb{N}$ decide whether there is a k-coloring of G. (If we can do it in $O^*(c^n)$ time then we can also **find** the coloring in $O^*(c^n)$ time when it exists, due to self-reducibility).

Observations

- (trivial) every k-coloring is a partition of V into k independent sets.
- (interesting) There is a partition of V into k independent sets iff there is a **cover** of V by k independent sets, i.e. k independent sets $I_1, \ldots, I_k$ such that $\bigcup_{j=1}^{k} I_j = V$.

Coloring in 2^n, cont'd

- U is the set of tuples $(I_1, \ldots, I_k)$, where I_j are independent sets (not necessarily disjoint nor even different!)

Coloring in 2^n, cont'd

- U is the set of tuples $(I_1, \ldots, I_k)$, where I_j are independent sets (not necessarily disjoint nor even different!)
- $A_v = \{(I_1, \ldots, I_k) \in U \;:\; v \in \bigcup_{j=1}^{k} I_j\}$

Coloring in 2^n, cont'd

- U is the set of tuples $(I_1, \ldots, I_k)$, where I_j are independent sets (not necessarily disjoint nor even different!)
- $A_v = \{(I_1, \ldots, I_k) \in U \;:\; v \in \bigcup_{j=1}^{k} I_j\}$
- Then $|\bigcap_{v \in V} A_v| \neq 0$ iff G is k-colorable.

Coloring in 2^n, cont'd

- U is the set of tuples $(I_1, \ldots, I_k)$, where I_j are independent sets (not necessarily disjoint nor even different!)
- $A_v = \{(I_1, \ldots, I_k) \in U \ : \ v \in \bigcup_{j=1}^{k} I_j\}$
- Then $|\bigcap_{v \in V} A_v| \neq 0$ iff G is k-colorable.
- The simplified problem:

$$|\bigcap_{v \in X} \overline{A_v}| =$$

Coloring in 2^n, cont'd

- U is the set of tuples $(I_1, \ldots, I_k)$, where I_j are independent sets (not necessarily disjoint nor even different!)
- $A_v = \{(I_1, \ldots, I_k) \in U \ : \ v \in \bigcup_{j=1}^{k} I_j\}$
- Then $|\bigcap_{v \in V} A_v| \neq 0$ iff G is k-colorable.
- The simplified problem:

$$|\bigcap_{v \in X} \overline{A_v}| = |\{(I_1, \ldots, I_k) \in U \ : \ I_1, \ldots, I_k \subseteq V - X\}|$$

Coloring in 2^n, cont'd

- U is the set of tuples $(I_1, \ldots, I_k)$, where I_j are independent sets (not necessarily disjoint nor even different!)
- $A_v = \{(I_1, \ldots, I_k) \in U \ : \ v \in \bigcup_{j=1}^k I_j\}$
- Then $|\bigcap_{v \in V} A_v| \neq 0$ iff G is k-colorable.
- The simplified problem:

$$|\bigcap_{v \in X} \overline{A_v}| = |\{(I_1, \ldots, I_k) \in U \ : \ I_1, \ldots, I_k \subseteq V - X\}| = s(V - X)^k$$

where $s(Y)$ = the number of independent sets in $G[Y]$.

Coloring in 2^n, cont'd

- U is the set of tuples $(I_1, \ldots, I_k)$, where I_j are independent sets (not necessarily disjoint nor even different!)
- $A_v = \{(I_1, \ldots, I_k) \in U \ : \ v \in \bigcup_{j=1}^{k} I_j\}$
- Then $|\bigcap_{v \in V} A_v| \neq 0$ iff G is k-colorable.
- The simplified problem:

$$|\bigcap_{v \in X} \overline{A_v}| = |\{(I_1, \ldots, I_k) \in U \ : \ I_1, \ldots, I_k \subseteq V - X\}| = s(V - X)^k$$

  where $s(Y)$ = the number of independent sets in $G[Y]$.
- $s(Y)$ can be computed at the beginning **for all subsets** $Y \subseteq V$: $s(Y) = s(Y - \{y\}) + s(Y - N[y])$. This takes time (**and space**) $O^*(2^n)$, since the number of covers takes $O(n \log k)$ bits.

Coloring in 2^n, cont'd

- U is the set of tuples $(I_1, \ldots, I_k)$, where I_j are independent sets (not necessarily disjoint nor even different!)
- $A_v = \{(I_1, \ldots, I_k) \in U \ : \ v \in \bigcup_{j=1}^{k} I_j\}$
- Then $|\bigcap_{v \in V} A_v| \neq 0$ iff G is k-colorable.
- The simplified problem:

$$|\bigcap_{v \in X} \overline{A_v}| = |\{(I_1, \ldots, I_k) \in U \ : \ I_1, \ldots, I_k \subseteq V - X\}| = s(V - X)^k$$

  where $s(Y) =$ the number of independent sets in $G[Y]$.
- $s(Y)$ can be computed at the beginning **for all subsets** $Y \subseteq V$: $s(Y) = s(Y - \{y\}) + s(Y - N[y])$. This takes time (**and space**) $O^*(2^n)$, since the number of covers takes $O(n \log k)$ bits.
- Next, we compute $|\bigcap_{v \in X} \overline{A_v}|$ easily in $O^*(1)$ time, so we get $|\bigcap_{v \in V} A_v|$ in $O^*(2^n)$ time.

Coloring in 2^n, cont'd

Theorem

In $O^*(2^n)$ time and space we can

- find a k-coloring or conclude it does not exist,
- find the chromatic number.

Coloring in 2^n, cont'd

Theorem

In $O^*(2^n)$ time and space we can

- find a k-coloring or conclude it does not exist,
- find the chromatic number.

Theorem

In $O^*(2.25^n)$ time and **polynomial space** we can find a k-coloring of a given graph G or conclude that it does not exist.

Proof

We compute $s(Y)$ in $O(1.2461^n)$ time and **polynomial space** by the algorithm of Fürer, Kasiviswanathan (2005). Total time:

$$\sum_{X \subseteq V} 1.2461^{|X|} = \sum_{k=0}^{n} \binom{n}{k} 1.2461^k = (1 + 1.2461)^n = O(2.25^n).$$

Remarks

Remark 1

By using a bit more complicated dynamic programming we can compute the „real" number of k-colorings (and not the number of covers) within the same time and space bound.

Remarks

Remark 1

By using a bit more complicated dynamic programming we can compute the „real” number of k-colorings (and not the number of covers) within the same time and space bound.

Remark 2

The presented algorithm can be extended to handle the general problem of covering/partitioning a set V by a family of subsets.

Steiner Tree in 2^k, Nederlof 2009

Unweighted version

Given graph $G = (V, E)$, the set of terminals $K \subseteq V$ and a number $c \in \mathbb{N}$.
Is there a tree $T \subseteq G$ such that $K \subseteq V(T)$ and $|E(T)| \leq c$?

Steiner Tree in 2^k, Nederlof 2009

Unweighted version

Given graph $G = (V, E)$, the set of terminals $K \subseteq V$ and a number $c \in \mathbb{N}$. Is there a tree $T \subseteq G$ such that $K \subseteq V(T)$ and $|E(T)| \leq c$?

Weighted version

Additionally: weights on edges $w : E \to \mathbb{N}$. Is there a tree $T \subseteq G$ such that $K \subseteq V(T)$ and $w(E(T)) \leq c$?

Steiner Tree in 2^k, Nederlof 2009

Unweighted version

Given graph $G = (V, E)$, the set of terminals $K \subseteq V$ and a number $c \in \mathbb{N}$. Is there a tree $T \subseteq G$ such that $K \subseteq V(T)$ and $|E(T)| \leq c$?

Weighted version

Additionally: weights on edges $w : E \to \mathbb{N}$. Is there a tree $T \subseteq G$ such that $K \subseteq V(T)$ and $w(E(T)) \leq c$?

Denote $n = |V|$, $k = |K|$.

The classical algorithm [Dreyfus, Wagner 1972]

Dynamic programming, works in $O^*(3^k)$ time and $O^*(2^k)$ space, even in the weighted version.

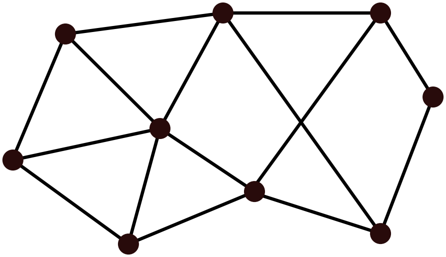
Branching walks

Definition

Let $G = (V, E)$ be an undirected graph and let $s \in V$. **A branching walk** is a pair $B = (T, h)$, where T is an ordered rooted tree and $h : V(T) \to V$ is a homomorphism, i.e. if $(x, y) \in E(T)$ then $h(x)h(y) \in E(G)$. We say that B is from s, when $h(r) = s$, where r is the root of T. The length of B is defined as $|E(T)|$.

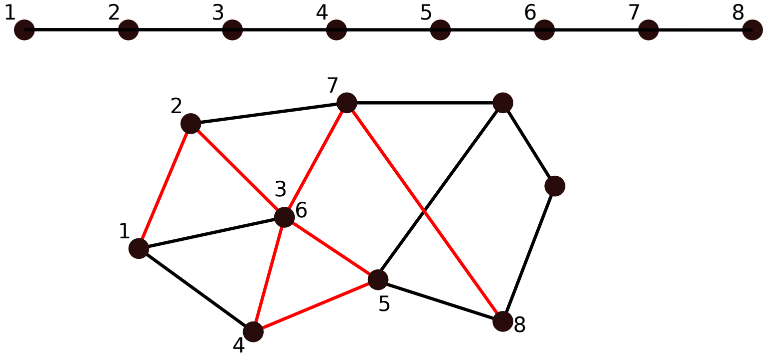
Branching walks

Example 1 Every walk is a branching walk



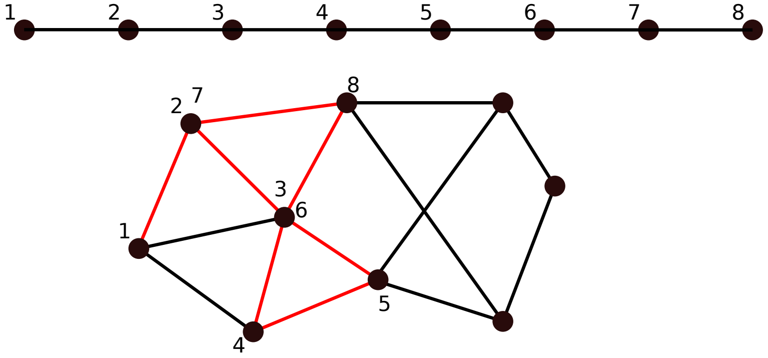
Branching walks

Example 1 Every walk is a branching walk

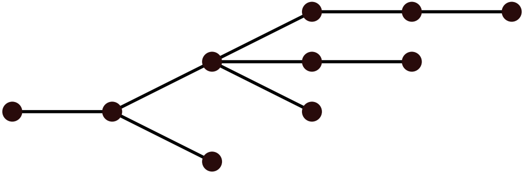


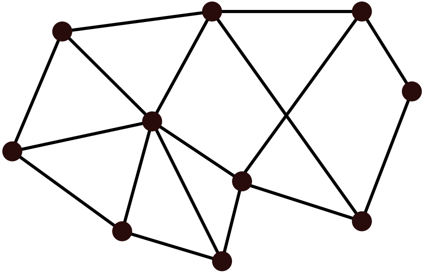
Branching walks

Example 2 Even this one.

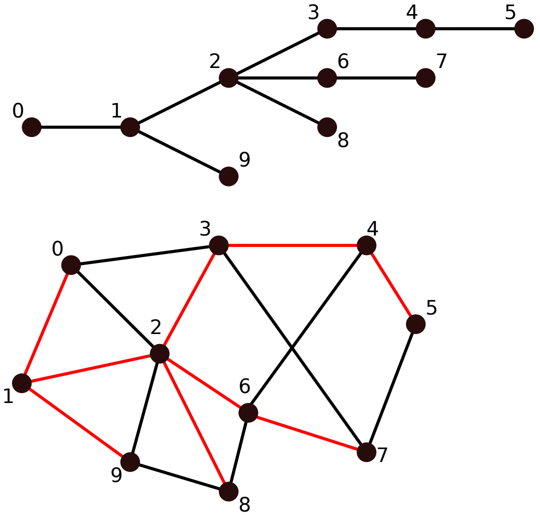


Branching walks

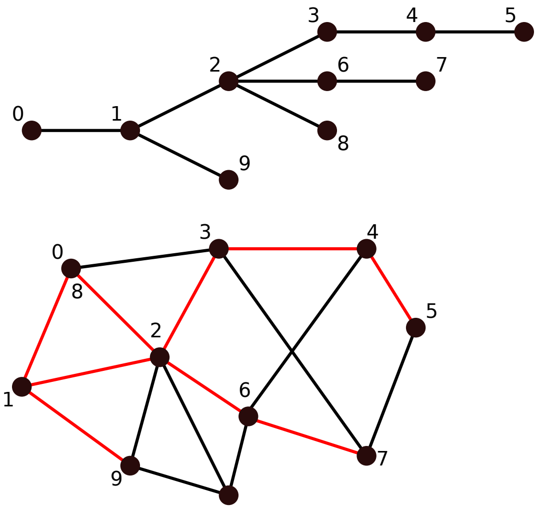




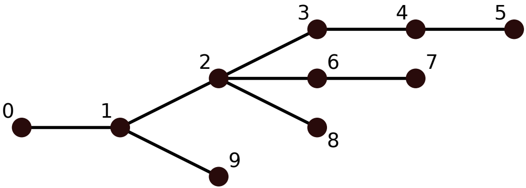
Example 3 An injective homomorphism.

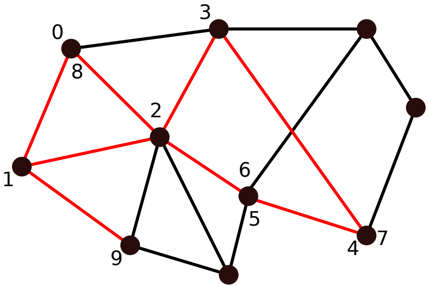


Example 4 A non-injective homomorphism.



Example 5 An even more non-injective homomorphism.





Steiner Tree, unweighted

Let $s \in K$ be any terminal.

Observation

G contains a tree T such that $K \subseteq V(T)$ and $|E(T)| \leq c$ iff there is a branching walk $B = (T_B, h)$ from s in G such that $K \subseteq h(V(T_B))$.

Steiner Tree, unweighted

Let $s \in K$ be any terminal.

Observation

G contains a tree T such that $K \subseteq V(T)$ and $|E(T)| \leq c$ iff there is a branching walk $B = (T_B, h)$ from s in G such that $K \subseteq h(V(T_B))$.

- U is the set of all length c branching walks from s.

Steiner Tree, unweighted

Let $s \in K$ be any terminal.

Observation

G contains a tree T such that $K \subseteq V(T)$ and $|E(T)| \leq c$ iff there is a branching walk $B = (T_B, h)$ from s in G such that $K \subseteq h(V(T_B))$.

- U is the set of all length c branching walks from s.
- $A_v = \{B \in U \ : \ v \in V(B)\}$ for $v \in K$.

Let $s \in K$ be any terminal.

Observation

G contains a tree T such that $K \subseteq V(T)$ and $|E(T)| \leq c$ iff there is a branching walk $B = (T_B, h)$ from s in G such that $K \subseteq h(V(T_B))$.

- U is the set of all length c branching walks from s.
- $A_v = \{B \in U \ : \ v \in V(B)\}$ for $v \in K$.
- Then $|\bigcap_{v \in K} A_v| \neq 0$ iff there is the desired Steiner Tree.

Steiner Tree, unweighted

Let $s \in K$ be any terminal.

Observation

G contains a tree T such that $K \subseteq V(T)$ and $|E(T)| \leq c$ iff there is a branching walk $B = (T_B, h)$ from s in G such that $K \subseteq h(V(T_B))$.

- U is the set of all length c branching walks from s.
- $A_v = \{B \in U \ : \ v \in V(B)\}$ for $v \in K$.
- Then $|\bigcap_{v \in K} A_v| \neq 0$ iff there is the desired Steiner Tree.
- For every $R \subseteq K$ let us denote $R' = R \cup (V - K)$.
- The simplified problem:

$$|\bigcap_{v \in K} \overline{A_v}| = b_c^{K-X}(s),$$

where $b_j^R(a) =$ is the number of length j branching walks from a in $G[R']$.

Steiner Tree, the simplified problem

For $R \subseteq K$ denote $R' = R \cup (V - K)$.

The simplified problem

$$|\bigcap_{v \in K} \overline{A_v}| = b_c^{K-X}(s),$$

where $b_j^R(a) =$ is the number of length j branching walks from a in $G[R']$.

Steiner Tree, the simplified problem

For $R \subseteq K$ denote $R' = R \cup (V - K)$.

The simplified problem

$$|\bigcap_{v \in K} \overline{A_v}| = b_c^{K-X}(s),$$

where $b_j^R(a) =$ is the number of length j branching walks from a in $G[R']$.

- we compute $b_j^R(a)$ for all $j = 0, \ldots, c$ and $a \in R'$ using DP:

$$\begin{cases} 1 & \text{when } j = 0, \\ \sum_{t \in N(a) \cap R'} \sum_{j_1 + j_2 = j-1} b_{j_1}^R(a) b_{j_2}^R(t) & \text{otherwise.} \end{cases}$$

Steiner Tree, the simplified problem

For $R \subseteq K$ denote $R' = R \cup (V - K)$.

The simplified problem

$$|\bigcap_{v \in K} \overline{A_v}| = b_c^{K-X}(s),$$

where $b_j^R(a) =$ is the number of length j branching walks from a in $G[R']$.

- we compute $b_j^R(a)$ for all $j = 0, \ldots, c$ and $a \in R'$ using DP:

$$\begin{cases} 1 & \text{when } j = 0, \\ \sum_{t \in N(a) \cap R'} \sum_{j_1 + j_2 = j-1} b_{j_1}^R(a) b_{j_2}^R(t) & \text{otherwise.} \end{cases}$$

- Note that $b_j^R = O((nj)^j)$ — by easy induction; hence b_j^R takes $O(n \log n) = O^*(1)$ bits.
- It follows that the the simplified problem can be solved in $O(n^4 \cdot n \log n) = O(n^5 \log n)$ time and $O(n^3 \log n)$ space.

Steiner Tree, finish

Corollary [Nederlof 2009]

The unweighted Steiner Tree problem can be solved in $O^*(2^k)$ time and polynomial space.

Steiner Tree, finish

Corollary [Nederlof 2009]

The unweighted Steiner Tree problem can be solved in $O^*(2^k)$ time and polynomial space.

Twierdzenie [Nederlof 2009]

The **weighted** Steiner Tree problem can be solved in $O^*(C \cdot 2^k)$ time and $O^*(C)$ space. (We skip the proof here)

Part III: Multi-linear detection in polynomials

The Schwartz-Zippel Lemma

Lemma [Schwartz 1980, Zippel 1979]

Let $p(x_1, x_2, \ldots, x_n)$ be a non-zero polynomial of degree at most d over a field F and let S be a finite subset of F. Then the probability that p evaluates to 0 on a random element $(a_1, a_2, \ldots, a_n) \in S^n$ is bounded by $d/|S|$.

The Schwartz-Zippel Lemma

Lemma [Schwartz 1980, Zippel 1979]

Let $p(x_1, x_2, \ldots, x_n)$ be a non-zero polynomial of degree at most d over a field F and let S be a finite subset of F. Then the probability that p evaluates to 0 on a random element $(a_1, a_2, \ldots, a_n) \in S^n$ is bounded by $d/|S|$.

Proof: Induction, for $n = 1$ we use the known result that a degree d polynomial has at most d zeroes.

The Schwartz-Zippel Lemma

Lemma [Schwartz 1980, Zippel 1979]

Let $p(x_1, x_2, \ldots, x_n)$ be a non-zero polynomial of degree at most d over a field F and let S be a finite subset of F. Then the probability that p evaluates to 0 on a random element $(a_1, a_2, \ldots, a_n) \in S^n$ is bounded by $d/|S|$.

A typical application

- We can efficiently evaluate a polynomial p of degree d.
- We want to test whether p is a non-zero polynomial.
- Then, we pick S so that $|S| \geq 2d$ and we evaluate p on a random element $e \in S$. We answer YES iff we got $p(e) \neq 0$.
- If p is the zero polynomial we always get NO, otherwise we get YES with probability at least $\frac{1}{2}$.
- This is called a Monte-Carlo algorithm with one-sided error.

The Schwartz-Zippel Lemma: Example 1

Corollary [Schwartz, Zippel]

Let P be a multivariate polynomial of degree d over a finite field F. If we can evaluate P in a given point in time T then we can check whether $P \equiv 0$ by a Monte-Carlo algorithm with one-sided error in time $T + O(1)$.

Polynomial equality testing

Input: Two multivariate polynomials P, Q given as an arithmetic circuit.
Question: Does $P \equiv Q$?

Note: A polynomial described by an arithmetic circuit of size s can have $2^{\Omega(s)}$ different monomials: $(x_1 + x_2)(x_1 - x_3)(x_2 + x_4) \cdots$.

Solution

The Schwartz-Zippel Lemma: Example 1

Corollary [Schwartz, Zippel]

Let P be a multivariate polynomial of degree d over a finite field F. If we can evaluate P in a given point in time T then we can check whether $P \equiv 0$ by a Monte-Carlo algorithm with one-sided error in time $T + O(1)$.

Polynomial equality testing

Input: Two multivariate polynomials P, Q given as an arithmetic circuit.
Question: Does $P \equiv Q$?

Note: A polynomial described by an arithmetic circuit of size s can have $2^{\Omega(s)}$ different monomials: $(x_1 + x_2)(x_1 - x_3)(x_2 + x_4) \cdots$.

Solution

Test whether the polynomial $P - Q$ is non-zero using the Schwartz-Zippel Lemma.

The Schwartz-Zippel Lemma: Example 2

Testing for perfect matching

Input: Bipartite graph $G = (A, B, E)$, $|A| = |B| = n$.
Question: Does G contain a perfect matching?

Lemma

Let M be the bipartite symbolic adjacency matrix of G, i.e. for $a \in A$, $b \in B$:

$$M_{a,b} = \begin{cases} x_{ab} & \text{when } ab \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Then $\det M \not\equiv 0$ iff G has a perfect matching.

Note that $\det M$ is a polynomial, each monomial corresponds to a p.m.

$$\det M = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} M_{i,\sigma_i}$$

The Schwartz-Zippel Lemma: Example 2, cont'd

Algorithm

1. Choose values of variables x_{ab} from a finite field F of size at least $2n$ uniformly at random,
2. We get a matrix $\tilde{M}$ over F.
3. Compute $\det \tilde{M}$ and return YES iff we $\det \tilde{M} \neq 0$.

Corollary

Existence of a perfect matching can be tested by a Monte-Carlo one-sided error algorithm by a single $n \times n$ matrix determinant evaluation.

Combining the blocks

- Bunch, Hopcroft: We can multiply two $n \times n$ matrices in time $O(n^\omega)$ $\Rightarrow$ we can compute the determinant of an $n \times n$ matrix in time $O(n^\omega)$.
- Coppersmith, Winograd: $\omega < 2.376$.
- Lovasz: So, we can test perfect matching in randomized $O(n^\omega)$ time!

Question

Question

What if the bound of $1/2$ for the probability of success is not enough for us?

Question

Question

What if the bound of $1/2$ for the probability of success is not enough for us?

Answer

Repeat the algorithm 1000 times and answer YES if there was **at least one** YES. Then,

$$Pr[error] \leq \frac{1}{2^{1000}}$$

Note

The probability that an earthquake destroys the computer is probably higher than $\frac{1}{2^{1000}} \ldots$

Finite fields of characteristic 2

In what follows, we use finite fields of size 2^k.
We need to know just three things about such fields:

- They exist,
- We can perform arithmetic operations fast, in $O(k^{O(1)})$ time,
- They are of characteristic two, i.e. $1 + 1 = 0$.
  (In particular, for any element a, we have $a + a = 0$.)

k-path problem

Problem

Input: directed/undirected graph G, integer k.
Question: Does G contain a simple path of length k?

A few facts

- NP-complete (why?)

k-path problem

Problem

Input: directed/undirected graph G, integer k.
Question: Does G contain a simple path of length k?

A few facts

- NP-complete (why?)
- even $O(f(k)n^{O(1)})$-time algorithm is non-trivial,

k-path problem

Problem

Input: directed/undirected graph G, integer k.
Question: Does G contain a simple path of length k?

A few facts

- NP-complete (why?)
- even $O(f(k)n^{O(1)})$-time algorithm is non-trivial,
- Monien 1985: $O(k!n^{O(1)})$

k-path problem

Problem

Input: directed/undirected graph G, integer k.
Question: Does G contain a simple path of length k?

A few facts

- NP-complete (why?)
- even $O(f(k)n^{O(1)})$-time algorithm is non-trivial,
- Monien 1985: $O(k!n^{O(1)})$
- Alon, Yuster, Zwick 1994: $O((2e)^k n^{O(1)})$

k-path problem

Problem

Input: directed/undirected graph G, integer k.
Question: Does G contain a simple path of length k?

A few facts

- NP-complete (why?)
- even $O(f(k)n^{O(1)})$-time algorithm is non-trivial,
- Monien 1985: $O(k!n^{O(1)})$
- Alon, Yuster, Zwick 1994: $O((2e)^k n^{O(1)})$
- Chen, Lu, She, Zhang 2007: $O(4^k n^{O(1)})$

k-path problem

Problem

Input: directed/undirected graph G, integer k.
Question: Does G contain a simple path of length k?

A few facts

- NP-complete (why?)
- even $O(f(k)n^{O(1)})$-time algorithm is non-trivial,
- Monien 1985: $O(k!n^{O(1)})$
- Alon, Yuster, Zwick 1994: $O((2e)^k n^{O(1)})$
- Chen, Lu, She, Zhang 2007: $O(4^k n^{O(1)})$
- Koutis 2008: $O(2^{3/2k} n^{O(1)})$

k-path problem

Problem

Input: directed/undirected graph G, integer k.
Question: Does G contain a simple path of length k?

A few facts

- NP-complete (why?)
- even $O(f(k)n^{O(1)})$-time algorithm is non-trivial,
- Monien 1985: $O(k!n^{O(1)})$
- Alon, Yuster, Zwick 1994: $O((2e)^k n^{O(1)})$
- Chen, Lu, She, Zhang 2007: $O(4^k n^{O(1)})$
- Koutis 2008: $O(2^{3/2k} n^{O(1)})$
- Williams 2009: $O(2^k)$

k-path problem

Problem

Input: directed/undirected graph G, integer k.
Question: Does G contain a simple path of length k?

A few facts

- NP-complete (why?)
- even $O(f(k)n^{O(1)})$-time algorithm is non-trivial,
- Monien 1985: $O(k!n^{O(1)})$
- Alon, Yuster, Zwick 1994: $O((2e)^k n^{O(1)})$
- Chen, Lu, She, Zhang 2007: $O(4^k n^{O(1)})$
- Koutis 2008: $O(2^{3/2k} n^{O(1)})$
- Williams 2009: $O(2^k)$
- Björklund, Husfeldt, Kaski, Koivisto 2010: $O(1.66^k)$, undirected graphs.

k-path problem

Problem

Input: directed/undirected graph G, integer k.
Question: Does G contain a simple path of length k?

A few facts

- NP-complete (why?)
- even $O(f(k)n^{O(1)})$-time algorithm is non-trivial,
- Monien 1985: $O(k!n^{O(1)})$
- Alon, Yuster, Zwick 1994: $O((2e)^k n^{O(1)})$
- Chen, Lu, She, Zhang 2007: $O(4^k n^{O(1)})$
- Koutis 2008: $O(2^{3/2k} n^{O(1)})$
- Williams 2009: $O(2^k)$
- Björklund, Husfeldt, Kaski, Koivisto 2010: $O(1.66^k)$, undirected graphs.

$O^*(2^k)$-time algorithm for k-path

Rough idea

- Want to construct a polynomial P^s, $P^s \not\equiv 0$ iff G has a k-path from s.

$O^*(2^k)$-time algorithm for k-path

Rough idea

- Want to construct a polynomial P^s, $P^s \not\equiv 0$ iff G has a k-path from s.
- First try: $$P^s(\cdots) = \sum_{k\text{-path } P \text{ from } s \text{ in } G} \text{monomial}(P).$$
  Seems good, but how to evaluate it?

$O^*(2^k)$-time algorithm for k-path

Rough idea

- Want to construct a polynomial P^s, $P^s \not\equiv 0$ iff G has a k-path from s.
- First try: $$P^s(\cdots) = \sum_{k\text{-path } P \text{ from } s \text{ in } G} \text{monomial}(P).$$
  Seems good, but how to evaluate it?
- Second try: $$P^s(\cdots) = \sum_{k\text{-walk } W \text{ from } s \text{ in } G} \text{monomial}(W).$$
  Now we **can** evaluate it but we may get false positives.

$O^*(2^k)$-time algorithm for k-path

Rough idea

- Want to construct a polynomial P^s, $P^s \not\equiv 0$ iff G has a k-path from s.
- First try: $$P^s(\cdots) = \sum_{k\text{-path } P \text{ from } s \text{ in } G} \text{monomial}(P).$$
  Seems good, but how to evaluate it?
- Second try: $$P^s(\cdots) = \sum_{k\text{-walk } W \text{ from } s \text{ in } G} \text{monomial}(W).$$
  Now we **can** evaluate it but we may get false positives.
- Final try:
  $$P^s(\cdots) = \sum_{k\text{-walk } W \text{ from } s \text{ in } G} \; \sum_{\substack{\ell:\{1,\dots,k\}\to\{1,\dots,k\} \\ \ell \text{ is bijective}}} \text{monomial}(w, \ell).$$
  - We still can evaluate it,
  - It turns out that every monomial corresponding to a walk which is not a path appears even number of times so it cancels-out!

Our Hero

$$P^s(\mathbf{x},\mathbf{y}) = \sum_{\substack{\text{walk } W = v_1,\ldots,v_k \\ v_1 = s}} \sum_{\substack{\ell:\{1,\ldots,k\}\to\{1,\ldots,k\} \\ \ell \text{ is bijective}}} \underbrace{\prod_{i=1}^{k-1} x_{v_i,v_{i+1}} \prod_{i=1}^{k} y_{v_i,\ell(i)}}_{\mathrm{mon}(W,\ell)}$$

Monomials corresponding to non-simple walks cancel-out

- Let $W = v_1, \ldots, v_k$ be a walk from s, and a bijection $\ell \in S_k$.

Monomials corresponding to non-simple walks cancel-out

- Let $W = v_1, \ldots, v_k$ be a walk from s, and a bijection $\ell \in S_k$.
- Assume $v_a = v_b$ for some $a < b$, if many such pairs take the lexicographically first.

Monomials corresponding to non-simple walks cancel-out

- Let $W = v_1, \ldots, v_k$ be a walk from s, and a bijection $\ell \in S_k$.
- Assume $v_a = v_b$ for some $a < b$, if many such pairs take the lexicographically first.
- We define $\ell' : \{1, \ldots, k\} \to \{1, \ldots, k\}$ as follows:

$$\ell'(x) = \begin{cases} \ell(b) & \text{if } x = a, \\ \ell(a) & \text{if } x = b, \\ \ell(x) & \text{otherwise.} \end{cases}$$

Monomials corresponding to non-simple walks cancel-out

- Let $W = v_1, \ldots, v_k$ be a walk from s, and a bijection $\ell \in S_k$.
- Assume $v_a = v_b$ for some $a < b$, if many such pairs take the lexicographically first.
- We define $\ell' : \{1, \ldots, k\} \to \{1, \ldots, k\}$ as follows:

$$\ell'(x) = \begin{cases} \ell(b) & \text{if } x = a, \\ \ell(a) & \text{if } x = b, \\ \ell(x) & \text{otherwise.} \end{cases}$$

- $(W, \ell) \neq (W, \ell')$ since ℓ is injective.

Monomials corresponding to non-simple walks cancel-out

- Let $W = v_1, \ldots, v_k$ be a walk from s, and a bijection $\ell \in S_k$.
- Assume $v_a = v_b$ for some $a < b$, if many such pairs take the lexicographically first.
- We define $\ell' : \{1, \ldots, k\} \to \{1, \ldots, k\}$ as follows:

$$\ell'(x) = \begin{cases} \ell(b) & \text{if } x = a, \\ \ell(a) & \text{if } x = b, \\ \ell(x) & \text{otherwise.} \end{cases}$$

- $(W, \ell) \neq (W, \ell')$ since ℓ is injective.
- $\text{mon}(W, \ell) = \prod_{i=1}^{k-1} x_{v_i, v_{i+1}} \prod_{i=1}^{k} y_{v_i, \ell(i)} =$

$$\prod_{i=1}^{k-1} x_{v_i, v_{i+1}} \prod_{i \in \{1,\ldots,k\} \setminus \{a,b\}} y_{v_i, \ell(i)} \underbrace{y_{v_a, \ell(a)}}_{y_{v_b \ell'(b)}} \underbrace{y_{v_b, \ell(b)}}_{y_{v_a \ell'(a)}} = \text{mon}(W, \ell')$$

Monomials corresponding to non-simple walks cancel-out

- Let $W = v_1, \ldots, v_k$ be a walk from s, and a bijection $\ell \in S_k$.
- Assume $v_a = v_b$ for some $a < b$, if many such pairs take the lexicographically first.
- We define $\ell' : \{1, \ldots, k\} \to \{1, \ldots, k\}$ as follows:

$$\ell'(x) = \begin{cases} \ell(b) & \text{if } x = a, \\ \ell(a) & \text{if } x = b, \\ \ell(x) & \text{otherwise.} \end{cases}$$

- $(W, \ell) \neq (W, \ell')$ since ℓ is injective.
- $\mathrm{mon}(W, \ell) = \mathrm{mon}(W, \ell')$

Monomials corresponding to non-simple walks cancel-out

- Let $W = v_1, \ldots, v_k$ be a walk from s, and a bijection $\ell \in S_k$.
- Assume $v_a = v_b$ for some $a < b$, if many such pairs take the lexicographically first.
- We define $\ell' : \{1, \ldots, k\} \to \{1, \ldots, k\}$ as follows:

$$\ell'(x) = \begin{cases} \ell(b) & \text{if } x = a, \\ \ell(a) & \text{if } x = b, \\ \ell(x) & \text{otherwise.} \end{cases}$$

- $(W, \ell) \neq (W, \ell')$ since ℓ is injective.
- $\text{mon}(W, \ell) = \text{mon}(W, \ell')$
- If we start from (W, ℓ') and follow the same way of assignment we get (W, ℓ). (Called a fixed-point free involution)

Monomials corresponding to non-simple walks cancel-out

- Let $W = v_1, \ldots, v_k$ be a walk from s, and a bijection $\ell \in S_k$.
- Assume $v_a = v_b$ for some $a < b$, if many such pairs take the lexicographically first.
- We define $\ell' : \{1, \ldots, k\} \to \{1, \ldots, k\}$ as follows:

$$\ell'(x) = \begin{cases} \ell(b) & \text{if } x = a, \\ \ell(a) & \text{if } x = b, \\ \ell(x) & \text{otherwise.} \end{cases}$$

- $(W, \ell) \neq (W, \ell')$ since ℓ is injective.
- $\text{mon}(W, \ell) = \text{mon}(W, \ell')$
- If we start from (W, ℓ') and follow the same way of assignment we get (W, ℓ). (Called a fixed-point free involution)
- Since the field is of characteristic 2, $\text{mon}(W, \ell)$ and $\text{mon}(W, \ell')$ cancel out!

Half the way...

Corollary

If $P^s \neq 0$ then there is a k-path.

The second half

Recall:

$$P^s(\mathbf{x},\mathbf{y}) = \sum_{\substack{\text{walk } W = v_1,\ldots,v_k \\ v_1 = s}} \sum_{\substack{\ell:\{1,\ldots,k\}\to\{1,\ldots,k\} \\ \ell \text{ is bijective}}} \underbrace{\prod_{i=1}^{k-1} x_{v_i,v_{i+1}} \prod_{i=1}^{k} y_{v_i,\ell(i)}}_{\mathrm{mon}(W,\ell)}$$

Question

Why not just $\mathrm{mon}(W,\ell) = x$ for a single variable x?
Why do we need exactly $\mathrm{mon}(W,\ell) = \prod_{i=1}^{k-1} x_{v_i,v_{i+1}} \prod_{i=1}^{k} y_{v_i,\ell(i)}$?

The second half

Recall:

$$P^s(\mathbf{x},\mathbf{y}) = \sum_{\substack{\text{walk } W = v_1,\ldots,v_k \\ v_1 = s}} \sum_{\substack{\ell:\{1,\ldots,k\}\to\{1,\ldots,k\} \\ \ell \text{ is bijective}}} \underbrace{\prod_{i=1}^{k-1} x_{v_i,v_{i+1}} \prod_{i=1}^{k} y_{v_i,\ell(i)}}_{\text{mon}(W,\ell)}$$

Question

Why not just $\text{mon}(W,\ell) = x$ for a single variable x?
Why do we need exactly $\text{mon}(W,\ell) = \prod_{i=1}^{k-1} x_{v_i,v_{i+1}} \prod_{i=1}^{k} y_{v_i,\ell(i)}$?

Answer

Now, every labelled walk which is a path gets a **unique** monomial.

The second half

Recall:

$$P^s(\mathbf{x},\mathbf{y}) = \sum_{\substack{\text{walk } W = v_1,\ldots,v_k \\ v_1 = s}} \sum_{\substack{\ell:\{1,\ldots,k\}\to\{1,\ldots,k\} \\ \ell \text{ is bijective}}} \underbrace{\prod_{i=1}^{k-1} x_{v_i,v_{i+1}} \prod_{i=1}^{k} y_{v_i,\ell(i)}}_{\text{mon}(W,\ell)}$$

Question

Why not just $\text{mon}(W,\ell) = x$ for a single variable x?
Why do we need exactly $\text{mon}(W,\ell) = \prod_{i=1}^{k-1} x_{v_i,v_{i+1}} \prod_{i=1}^{k} y_{v_i,\ell(i)}$?

Answer

Now, every labelled walk which is a path gets a **unique** monomial.

Corollary

If there is a k-path from s then $P^s \not\equiv 0$.

Where are we?

Corollary

There is a k-path from s iff $P^s \not\equiv 0$.

The missing element

How to evaluate P^s efficiently?
($O^*(2^k)$ is efficiently enough.)

Weighted inclusion-exclusion

Let $A_1, \ldots, A_n \subseteq U$, where U is a finite set.
Let $w : U \to F$ be a weight function.
For any $X \subseteq U$ denote $w(X) = \sum_{x \in X} w(x)$.
Let us also denote $\bigcap_{i \in \emptyset}(U - A_i) = U$.

Then,

$$w\left(\bigcap_{i \in \{1,\ldots,n\}} A_i\right) = \sum_{X \subseteq \{1,\ldots,n\}} (-1)^{|X|} w\left(\bigcap_{i \in X}(U - A_i)\right).$$

Weighted inclusion-exclusion

Let $A_1, \ldots, A_n \subseteq U$, where U is a finite set.
Let $w : U \to F$ be a weight function.
For any $X \subseteq U$ denote $w(X) = \sum_{x \in X} w(x)$.
Let us also denote $\bigcap_{i \in \emptyset}(U - A_i) = U$.

Then,

$$w\left(\bigcap_{i \in \{1,\ldots,n\}} A_i\right) = \sum_{X \subseteq \{1,\ldots,n\}} (-1)^{|X|} w\left(\bigcap_{i \in X}(U - A_i)\right).$$

Counting over a field of characteristic 2 we know that $-1 = 1$ so we can remove the $(-1)^{|X|}$:

$$w\left(\bigcap_{i \in \{1,\ldots,n\}} A_i\right) = \sum_{X \subseteq \{1,\ldots,n\}} w\left(\bigcap_{i \in X}(U - A_i)\right).$$

Evaluating $P^s(\mathbf{x},\mathbf{y}) = \sum_{\substack{\text{walk } W \\ \text{from } s}} \sum_{\substack{\ell:\{1,\dots,k\}\to\{1,\dots,k\} \\ \ell \text{ is bijective}}} \mathrm{mon}(W,\ell)$

Fix a walk W from s.

- $U = \{\ell : \{1,\dots,k\} \to \{1,\dots,k\}\}$ (all functions)
- for $\ell \in U$, define the weight $w(\ell) = \mathrm{mon}(W,\ell)$.
- for $i = 1,\dots,k$ let $A_i = \{\ell \in U \;:\; \ell^{-1}(i) \neq \emptyset\}$.
- Then,

$$\sum_{\substack{\ell:\{1,\dots,k\}\to\{1,\dots,k\} \\ \ell \text{ is bijective}}} \mathrm{mon}(W,\ell) = \sum_{\substack{\ell:\{1,\dots,k\}\to\{1,\dots,k\} \\ \ell \text{ is surjective}}} \mathrm{mon}(W,\ell) = w(\bigcap_{i=1}^{k} A_i).$$

- By weighted I-E,

$$\sum_{\substack{\ell:\{1,\dots,k\}\to\{1,\dots,k\} \\ \ell \text{ is bijective}}} \mathrm{mon}(W,\ell) = \sum_{X\subseteq\{1,\dots,k\}} w\left(\bigcap_{i\in X}(U - A_i)\right) =$$

$$\sum_{X\subseteq\{1,\dots,k\}} \sum_{\ell:\{1,\dots,k\}\to\{1,\dots,k\}\setminus X} \mathrm{mon}(W,\ell)$$

Evaluating $P^s(\mathbf{x},\mathbf{y}) = \sum_{\substack{\text{walk } W \\ \text{from } s}} \sum_{\substack{\ell:\{1,\ldots,k\}\to\{1,\ldots,k\} \\ \ell \text{ is bijective}}} \text{mon}(W,\ell)$

Fix a walk W from s.

- $U = \{\ell : \{1,\ldots,k\} \to \{1,\ldots,k\}\}$ (all functions)
- for $\ell \in U$, define the weight $w(\ell) = \text{mon}(W,\ell)$.
- for $i = 1,\ldots,k$ let $A_i = \{\ell \in U \ : \ \ell^{-1}(i) \neq \emptyset\}$.
- Then,

$$\sum_{\substack{\ell:\{1,\ldots,k\}\to\{1,\ldots,k\} \\ \ell \text{ is bijective}}} \text{mon}(W,\ell) = \sum_{\substack{\ell:\{1,\ldots,k\}\to\{1,\ldots,k\} \\ \ell \text{ is surjective}}} \text{mon}(W,\ell) = w(\bigcap_{i=1}^{k} A_i).$$

- By weighted I-E,

$$\sum_{\substack{\ell:\{1,\ldots,k\}\to\{1,\ldots,k\} \\ \ell \text{ is bijective}}} \text{mon}(W,\ell) = \sum_{X\subseteq\{1,\ldots,k\}} w\left(\bigcap_{i\in X}(U - A_i)\right) =$$

$$\sum_{X\subseteq\{1,\ldots,k\}} \sum_{\ell:\{1,\ldots,k\}\to X} \text{mon}(W,\ell)$$

Evaluating $P^s(\mathbf{x},\mathbf{y}) = \sum_{\substack{\text{walk } W \\ \text{from } s}} \sum_{\substack{\ell:\{1,\ldots,k\}\to\{1,\ldots,k\} \\ \ell \text{ is bijective}}} \mathrm{mon}(W,\ell)$

We got

$$\sum_{\substack{\ell:\{1,\ldots,k\}\to\{1,\ldots,k\} \\ \ell \text{ is bijective}}} \mathrm{mon}(W,\ell) = \sum_{X\subseteq\{1,\ldots,k\}} \sum_{\ell:\{1,\ldots,k\}\to X} \mathrm{mon}(W,\ell)$$

Hence,

$$\begin{aligned} P^s(\mathbf{x},\mathbf{y}) &= \sum_{\substack{\text{walk } W \\ \text{from } s}} \sum_{X\subseteq\{1,\ldots,k\}} \sum_{\ell:\{1,\ldots,k\}\to X} \mathrm{mon}(W,\ell) \\ &= \sum_{X\subseteq\{1,\ldots,k\}} \underbrace{\sum_{\substack{\text{walk } W \\ \text{from } s}} \sum_{\ell:\{1,\ldots,k\}\to X} \mathrm{mon}(W,\ell)}_{P^s_X(\mathbf{x},\mathbf{y})} \end{aligned}$$

Evaluating $P_X^s(\mathbf{x},\mathbf{y}) = \sum_{\substack{\text{walk } W \\ \text{from } s \\ \text{of length } k}} \sum_{\ell:\{1,\ldots,k\}\to X} \text{mon}(W,\ell)$ in poly-time

We use dynamic programming. (How?)

Evaluating $P_X^s(\mathbf{x},\mathbf{y}) = \sum_{\substack{\text{walk } W \\ \text{from } s \\ \text{of length } k}} \sum_{\ell:\{1,\ldots,k\}\to X} \text{mon}(W,\ell)$ in poly-time

We use dynamic programming. (How?)
Fill the 2-dimensional table T,

$$T[v,d] = \sum_{\substack{\text{walk } W = v_1,\ldots,v_d \\ v_1 = v \\ \text{of length } d}} \sum_{\ell:\{1,\ldots,k\}\to X} \prod_{i=1}^{k-1} x_{v_i,v_{i+1}} \prod_{i=1}^{k} y_{v_i,\ell(i)}$$

Then,

$$T[v,d] = \begin{cases} 1 & \text{when } d = 1, \\ \sum_{(v,w)\in E} x_{vw} \sum_{l\in X} y_{wl} \cdot T[w,d-1] & \text{otherwise.} \end{cases}$$

Hence, $P_X^s(\mathbf{x},\mathbf{y}) = T[s,k]$ can be computed in $O(kn)$ time and space.

The last slide

Corollary

The k-path problem can be solved by a $O^*(2^k)$-time polynomial space one-sided error Monte-Carlo algorithm.

Bibliography I

A book:

F. Fomin, D. Kratsch.
Exact Exponential Algorithms.
Springer, 2010.

Articles:

A. Björklund, T. Husfeldt, and M. Koivisto.
Set Partitioning via Inclusion-Exclusion.
SIAM J. Comput., 39(2):546–563, 2009.

A. Björklund.
Determinant sums for undirected hamiltonicity.
In Proc. FOCS'10, pages 173–182, 2010.

A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto.
Narrow sieves for parameterized paths and packings.
CoRR, abs/1007.1161, 2010.

Bibliography II

I. Koutis.
Faster algebraic algorithms for path and packing problems.
In Proc. ICALP'08, volume 5125 of LNCS, pages 575–586, 2008.

J. Nederlof.
Fast polynomial-space algorithms using Möbius inversion: Improving on steiner tree and related problems.
In Proc. ICALP'09, volume 5555 of LNCS, pages 713–725, 2009.

R. Williams.
Finding paths of length k in $O^*(2^k)$ time.
Inf. Process. Lett., 109(6):315–318, 2009.