enumeration algorithms

Arnaud Mary

October 7, 2015

zoo of problems

For a problem A and an instance X of A we denote by $A(X)$ the set of solutions of A on X.

zoo of problems

For a problem A and an instance X of A we denote by $A(X)$ the set of solutions of A on X.

- Decision problem: decide whether $A(X) = \emptyset$

zoo of problems

For a problem A and an instance X of A we denote by $A(X)$ the set of solutions of A on X.

- Decision problem: decide whether $A(X) = \emptyset$
- Optimisation problem: find $y \in A(X)$ with optimal cost $\omega(y)$ for a given cost function ω

zoo of problems

For a problem A and an instance X of A we denote by $A(X)$ the set of solutions of A on X.

- Decision problem: decide whether $A(X) = \emptyset$
- Optimisation problem: find $y \in A(X)$ with optimal cost $\omega(y)$ for a given cost function ω
- Counting problem: compute $|A(X)|$

zoo of problems

For a problem A and an instance X of A we denote by $A(X)$ the set of solutions of A on X.

- Decision problem: decide whether $A(X) = \emptyset$
- Optimisation problem: find $y \in A(X)$ with optimal cost $\omega(y)$ for a given cost function ω
- Counting problem: compute $|A(X)|$
- Enumeration problem: Output $A(X)$

example

A clique C of a graph is said to be maximal if it is not included in any other clique.

example

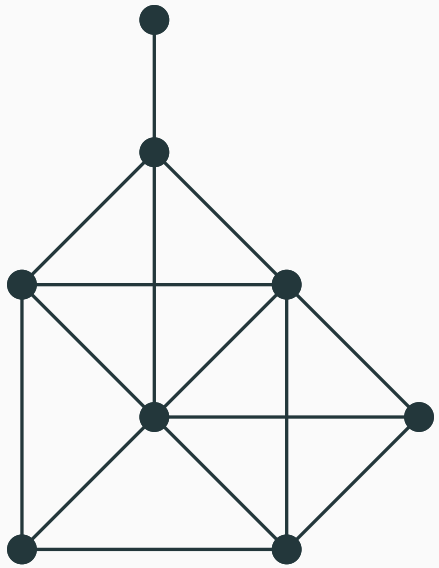
A clique C of a graph is said to be maximal if it is not included in any other clique.

Problem: Maximal Cliques Enumeration

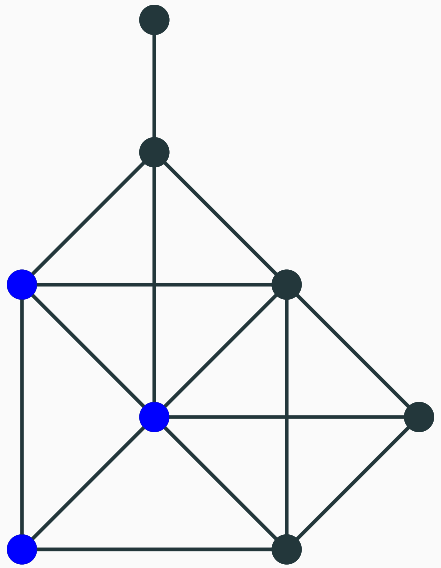
Input: A graph G

Output: All maximal cliques of G

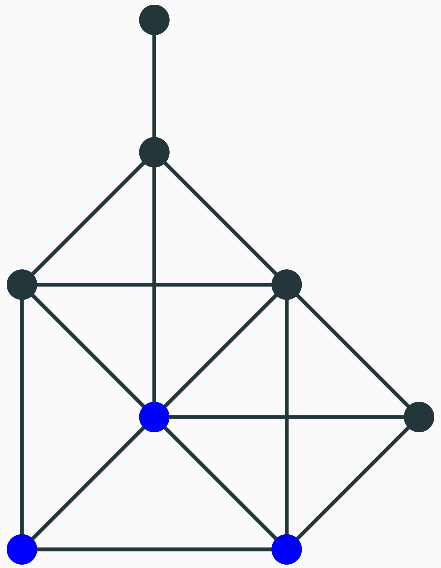
example



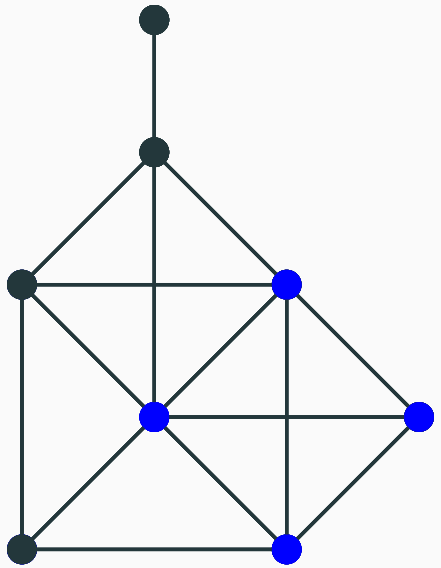
example



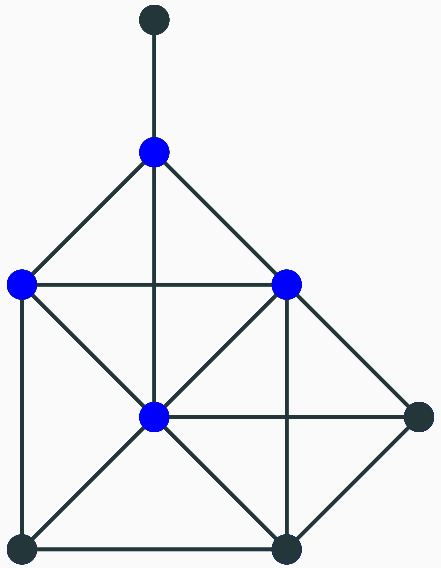
example



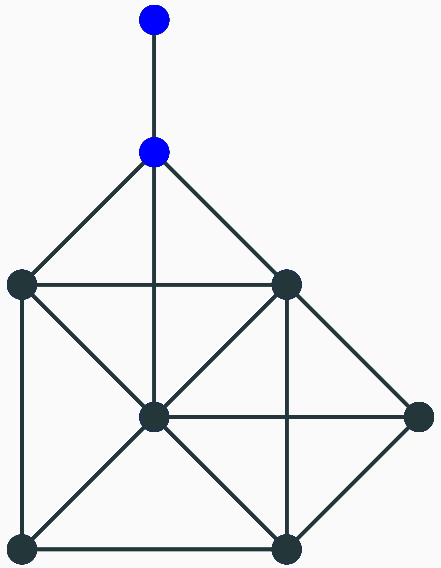
example



example



example



complexity

- The number of outputs of an enumeration problem may be exponential in the size of the input.

complexity

- The number of outputs of an enumeration problem may be exponential in the size of the input.
- For instance: the number of maximal cliques of a graph can be exponential on n (a graph can have $3^{n/3}$ maximal cliques)

complexity

- The number of outputs of an enumeration problem may be exponential in the size of the input.
- For instance: the number of maximal cliques of a graph can be exponential on n (a graph can have $3^{n/3}$ maximal cliques)
- We cannot hope to have a algorithm whose running time is polynomial in the input (you need at least one operation per output).

dilemma

- What is an efficient enumeration algorithm?

dilemma

- What is an efficient enumeration algorithm?
- How to define tractability for enumeration problems?

dilemma

- What is an efficient enumeration algorithm?
- How to define tractability for enumeration problems?
- How to measure the complexity of an enumeration algorithm.

two schools

two schools

Two points of view

Input sensitive

- The complexity is measured in term of the input size only.

Output sensitive

two schools

Two points of view

Input sensitive

- The complexity is measured in term of the input size only.
- For most of problems the complexity are exponential.

Output sensitive

two schools

Two points of view

Input sensitive

- The complexity is measured in term of the input size only.
- For most of problems the complexity are exponential.

Output sensitive

- The complexity is measured in term of the input size and the output size.

two schools

Two points of view

Input sensitive

- The complexity is measured in term of the input size only.
- For most of problems the complexity are exponential.

Output sensitive

- The complexity is measured in term of the input size and the output size.
- We then can talk about "polynomial" algorithm even if the number of solutions can be exponentially large in the input size.

two schools

Two points of view

Input sensitive

- The goal is to find an exponential algorithm with the smallest complexity

Output sensitive

two schools

Two points of view

Input sensitive

- The goal is to find an exponential algorithm with the smallest complexity
- Typical example is to find an algorithm of complexity α^n with alpha as small as possible (usually smaller than 2)

Output sensitive

two schools

Two points of view

Input sensitive

- The goal is to find an exponential algorithm with the smallest complexity
- Typical example is to find an algorithm of complexity α^n with alpha as small as possible (usually smaller than 2)

Output sensitive

- We try to find an algorithm polynomial in the input size and output size

two schools

Two points of view

Input sensitive

- The goal is to find an exponential algorithm with the smallest complexity
- Typical example is to find an algorithm of complexity α^n with alpha as small as possible (usually smaller than 2)

Output sensitive

- We try to find an algorithm polynomial in the input size and output size
- ideally we want a complexity linear in the output size.

Exemple

Input sensitive

Given a graph G with n vertices, there exists an algorithm that enumerates all maximal cliques in $O(3^{\frac{n}{3}}) \sim O(1.442^n)$.

Output sensitive

Exemple

Input sensitive

Given a graph G with n vertices, there exists an algorithm that enumerates all maximal cliques in $O(3^{\frac{n}{3}}) \sim O(1.442^n)$.

Output sensitive

Given a graph G with n vertices, there exists an algorithm that enumerates all maximal cliques in $O(n^3|\mathcal{C}|)$ where $|\mathcal{C}|$ denotes the number of maximal cliques of G.

Input sensitive

Pros

- We know in advance the running time of the algorithm.

Cons

Input sensitive

Pros

- We know in advance the running time of the algorithm.
- The complexity of the algorithm gives a combinatorial bound on the number of possible solutions

Cons

Input sensitive

Pros

- We know in advance the running time of the algorithm.
- The complexity of the algorithm gives a combinatorial bound on the number of possible solutions
- The algorithm gives an exact exponential algorithm for the optimisation problem

Cons

Input sensitive

Pros

- We know in advance the running time of the algorithm.
- The complexity of the algorithm gives a combinatorial bound on the number of possible solutions
- The algorithm gives an exact exponential algorithm for the optimisation problem

Cons

- We don't know the relative efficiency of the algorithm. Is it far to be optimal or not?

Input sensitive

Pros

- We know in advance the running time of the algorithm.
- The complexity of the algorithm gives a combinatorial bound on the number of possible solutions
- The algorithm gives an exact exponential algorithm for the optimisation problem

Cons

- We don't know the relative efficiency of the algorithm. Is it far to be optimal or not?
- The complexity measure depends on the number of solutions that one instance can have.

Input sensitive

Pros

- We know in advance the running time of the algorithm.
- The complexity of the algorithm gives a combinatorial bound on the number of possible solutions
- The algorithm gives an exact exponential algorithm for the optimisation problem

Cons

- We don't know the relative efficiency of the algorithm. Is it far to be optimal or not?
- The complexity measure depends on the number of solutions that one instance can have.
- We don't know the behavior of the algorithm on instances with few solutions.

output sensitive

Output sensitive

Pros	Cons
• The efficiency does not depend on the problem.	

output sensitive

Output sensitive

Pros

- The efficiency does not depend on the problem.
- We can see if an algorithm is far from the (theoretical) optimal complexity.

Cons

output sensitive

Output sensitive

Pros	Cons
• The efficiency does not depend on the problem. • We can see if an algorithm is far from the (theoretical) optimal complexity. • We can classify enumeration problems by their output sensitive complexity.	

Output sensitive

Pros

- The efficiency does not depend on the problem.
- We can see if an algorithm is far from the (theoretical) optimal complexity.
- We can classify enumeration problems by their output sensitive complexity.

Cons

- We don't know in advance how many times the algorithm will run.

Output sensitive

Pros

- The efficiency does not depend on the problem.
- We can see if an algorithm is far from the (theoretical) optimal complexity.
- We can classify enumeration problems by their output sensitive complexity.

Cons

- We don't know in advance how many times the algorithm will run.
- Doesn't give any information on the maximal number of solutions an instance can have.

input sensitive enumeration

input sensitive algorithm

Comes from exact exponential algorithm community

Methods used:

- Branch and reduce algorithm
- Measure and conquer algorithm
- Classical dynamic programming algorithm.

branch and reduce algorithm

A branch and reduce algorithm is a recursive algorithm that divides the instance in some smaller instances based on a set of rules. There are two kinds of rules:

- **Reduction rules:** They just reduce the size of the instance. They are usually based on easy observations.
- **Branching rules:** The rules that effectively divide the problem in smaller sub-problems.

toy example

Enumeration of maximal independent sets of a graph G of maximum degree 2

- S: The current set of vertices taken in the solution.
- F: The set of "free vertices"; the vertices that are allowed to be taken in the solution.

toy example

Enumeration of maximal independent sets of a graph G of maximum degree 2

- S: The current set of vertices taken in the solution.
- F: The set of "free vertices"; the vertices that are allowed to be taken in the solution.

Reduction Rules:

- If F contains a degree 0 vertex, then add it to S and remove it from F

Enumeration of maximal independent sets of a graph G of maximum degree 2

Branching Rules:

- If F contains a degree 1 vertex u with neighbour v, then branch in the following ways:
  - Add u to S and remove u and v from F.
  - Add v to S and remove u, v and all neighbours of v from F.

Enumeration of maximal independent sets of a graph G of maximum degree 2

Branching Rules:

- If F contains a degree 2 vertex u with neighbour v, w, then branch in the following ways:
  - Add u to S and remove u, v and w from F.
  - Add v to S and remove u, v and all neighbours of v from F.
  - Add w to S and remove u, w and all neighbours of w from F.

complexity analysis

- To determine the complexity of the previous algorithm, let us analyse its recursion tree.

complexity analysis

- To determine the complexity of the previous algorithm, let us analyse its recursion tree.
- We want to determine the number of leaves of the tree.

complexity analysis

- To determine the complexity of the previous algorithm, let us analyse its recursion tree.
- We want to determine the number of leaves of the tree.
- In each internal node of the tree we count the number of sub-problems we create and how the size of these sub-problems decreases.

branching vectors

Branching Rules:

- If F contains a degree 1 vertex u with neighbour v, then branch in the following ways:
  - Add u to S and remove u and v from F. $|F|$ decreases by 2
  - Add v to S and remove u, v and all neighbours of v from F. $|F|$ decreases by 2

We branch in two sub-problems each whose sizes decrease by 2.

It is described by the Branching vector $(2, 2)$

branching vectors

Branching Rules:

- If F contains a degree 2 vertex u with neighbour v, w, then branch in the following ways:
  - Add u to S and remove u, v and w from F. $|F|$ decreases by 3
  - Add v to S and remove u, v and all neighbours of v from F. $|F|$ decreases by 3
  - Add w to S and remove u, w and all neighbours of w from F. $|F|$ decreases by 3

We branch in three sub-problems each whose sizes decrease by 3.

Branching vector: $(3, 3, 3)$

example of tree with branching vector $(3, 3)$

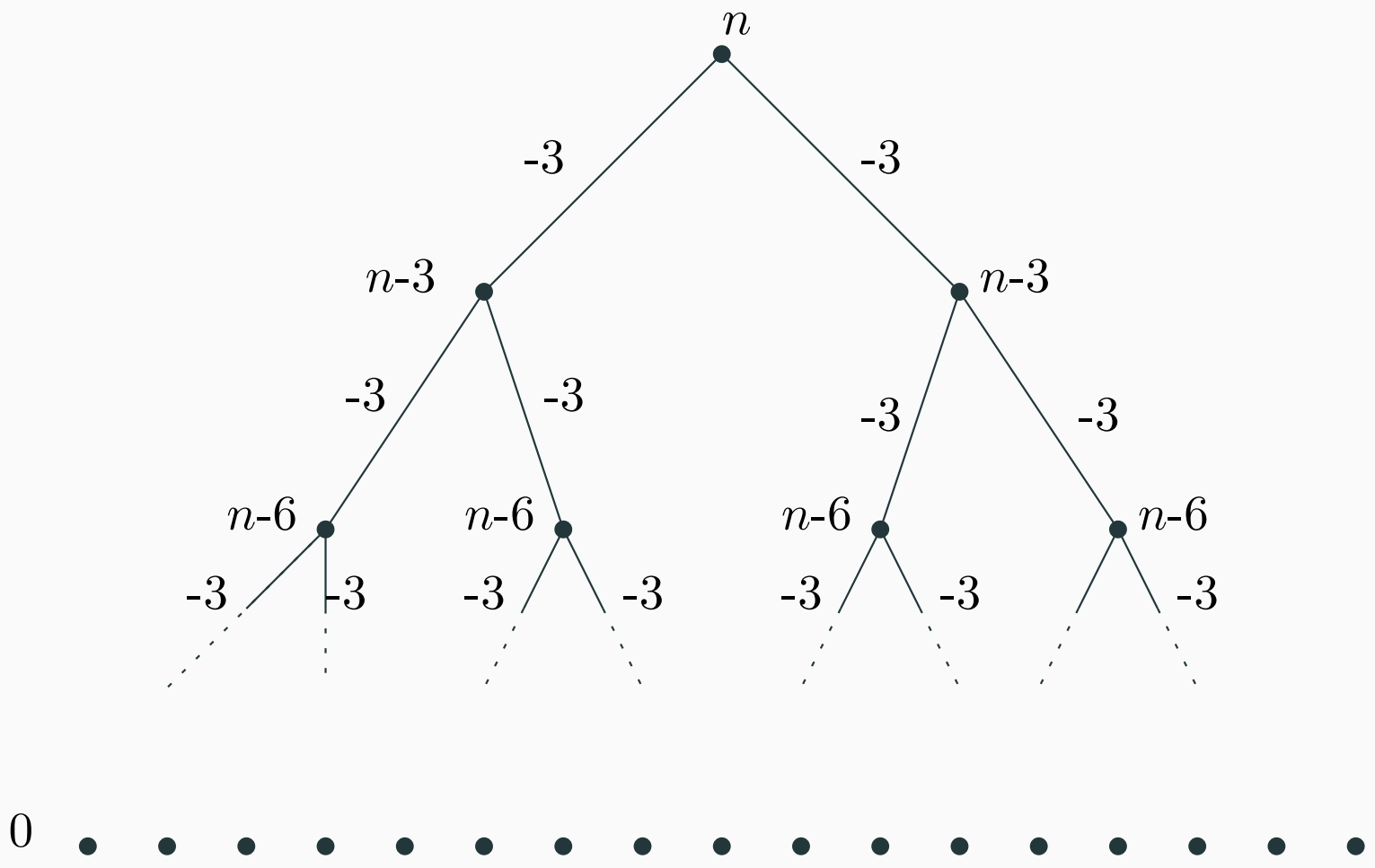


Figure 1: Tree with branching vector $(3, 3)$ at each node

example of tree with branching vector $(3, 3)$

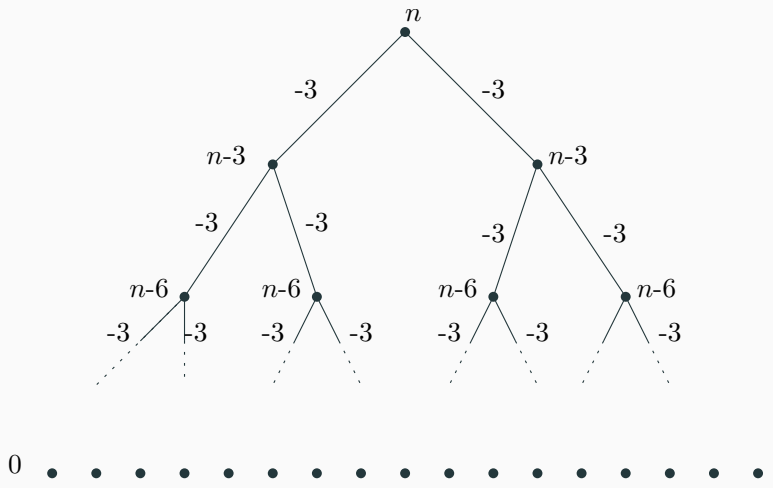


Figure 2: Tree with branching vector $(3, 3)$ at each node

How many leaves has this tree?

example of tree with branching vector $(3, 3)$

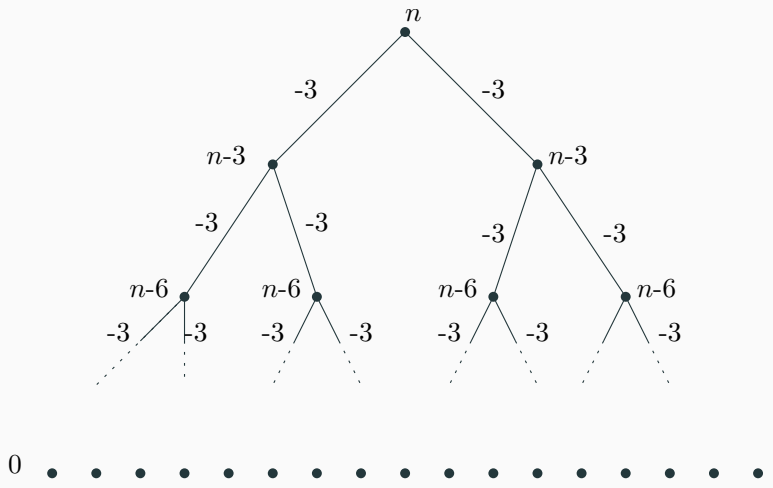


Figure 2: Tree with branching vector $(3, 3)$ at each node

How many leaves has this tree?

Satisfies the recurrence $T(n) = T(n-3) + T(n-3)$

example of tree with branching vector $(3, 3)$

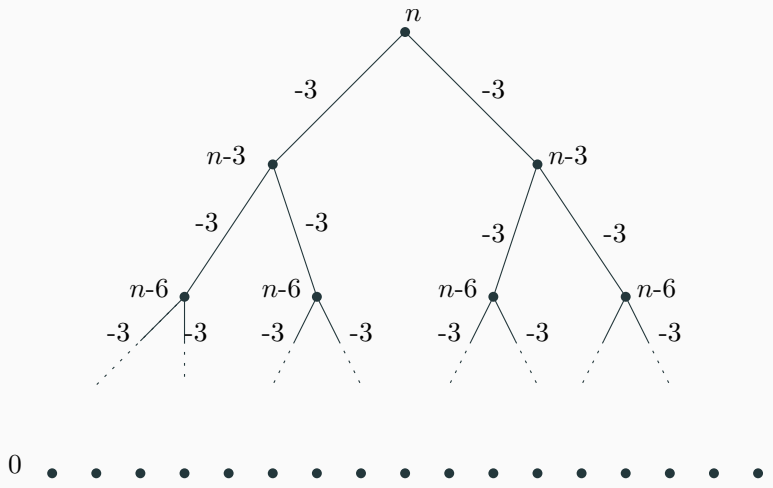


Figure 2: Tree with branching vector $(3, 3)$ at each node

How many leaves has this tree?

Satisfies the recurrence $T(n) = T(n-3) + T(n-3)$

$T(n) = 2^{n/3} \sim 1.259^n$

Question: What is the number of leaves of a tree with a given branching vector $(x_1, x_2, ..., x_k)$

Question: What is the number of leaves of a tree with a given branching vector $(x_1, x_2, ..., x_k)$

The number of leaves $T(n)$ is the solution of the recurrence

$$T(n) = T(n - x_1) + T(n - x_2) + ... + T(n - x_k)$$

Question: What is the number of leaves of a tree with a given branching vector $(x_1, x_2, ..., x_k)$
The number of leaves $T(n)$ is the solution of the recurrence

$$T(n) = T(n - x_1) + T(n - x_2) + ... + T(n - x_k)$$

Theorem

A tree with branching vector $(x_1, x_2, ..., x_k)$ has α^n leaves where α is the unique real positive solution of the equation:

$$x^n = x^{n-x_1} + x^{n-x_2} + ... + x^{n-x_k}$$

Theorem

A tree with branching vector $(x_1, x_2, ..., x_k)$ *has* α^n *leaves where* α *is the unique real positive solution of the equation:*

$$x^n = x^{n-x_1} + x^{n-x_2} + ... + x^{n-x_k}$$

α is called the branching factor of the branching vector $(x_1, x_2, ..., x_k)$

back to our toy example

First Branching Rule:

- If F contains a degree 1 vertex u with neighbour v, then branch in the following ways:
  - Add u to S and remove u and v from F. $|F|$ decreases by 2
  - Add v to S and remove u, v and all neighbours of v from F. $|F|$ decreases by 2

Branching vector: $(2, 2)$

Branching factor of $(2, 2)$: $\sqrt{2} \sim 1.414$

back to our toy example

Second Branching Rule:

- If F contains a degree 2 vertex u with neighbour v, w, then branch in the following ways:
  - Add u to S and remove u, v and w from F. $|F|$ decreases by 3
  - Add v to S and remove u, v and all neighbours of v from F. $|F|$ decreases by 3
  - Add w to S and remove u, w and all neighbours of w from F. $|F|$ decreases by 3

Branching vector: $(3, 3, 3)$

Branching factor of $(3, 3, 3)$: $\sqrt[3]{3} \sim 1.442$

back to our toy example

To estimate the number of leaves $T(n)$ of the recursion tree (and then the complexity of the algorithm), we keep the worst branching factor

back to our toy example

To estimate the number of leaves $T(n)$ of the recursion tree (and then the complexity of the algorithm), we keep the worst branching factor

- Branching vector: $(2, 2)$ with branching factor $\sqrt{2} \sim 1.414$
- Branching vector: $(3, 3, 3)$ with branching factor $\sqrt[3]{3} \sim 1.442$

back to our toy example

To estimate the number of leaves $T(n)$ of the recursion tree (and then the complexity of the algorithm), we keep the worst branching factor

- Branching vector: $(2, 2)$ with branching factor $\sqrt{2} \sim 1.414$
- Branching vector: $(3, 3, 3)$ with branching factor $\sqrt[3]{3} \sim 1.442$

$T(n) \leq (\sqrt[3]{3})^n = 3^{n/3} \sim 1.442^n$

exercise

Problem: Exact 3-Sat enumeration

Input: A 3-CNF formula with m clauses and n variables

Output: All satisfying assignments of variables with exactly one true literal per clause

exercise

Problem: Exact 3-Sat enumeration

Input: A 3-CNF formula with m clauses and n variables

Output: All satisfying assignments of variables with exactly one true literal per clause

Try to find an algorithm to solve this problem.

output sensitive enumeration

Total-Polynomial algorithm

output sensitive enumeration

Here the goal is to find algorithms whose complexity in term of input and output size is small.

output sensitive enumeration

Here the goal is to find algorithms whose complexity in term of input and output size is small.

Definition

An enumeration algorithm is said to be total-polynomial (or output-polynomial) if its running time is

$$poly(m+n)$$

where $poly$ is a polynomial, n is the size of the input and m is the size of the output.

TotalP

Definition

We call TotalP the class of enumeration problems that admit a total-polynomial algorithm.

TotalP

Definition

We call TotalP the class of enumeration problems that admit a total-polynomial algorithm.

How to prove that a problem is not in TotalP?

decision enumeration

To a given enumeration problem A, one can associate a decision problem called the DecisionEnumeration problem.

decision enumeration

To a given enumeration problem A, one can associate a decision problem called the DecisionEnumeration problem.

DecisionEnumeration problem

input: A set of solutions $\mathcal{S} := \{s_1, ..., s_k\}$

Question: Is $\mathcal{S}$ the set of all solution?

decision enumeration

To a given enumeration problem A, one can associate a decision problem called the DecisionEnumeration problem.

DecisionEnumeration problem

input: A set of solutions $\mathcal{S} := \{s_1, ..., s_k\}$
Question: Is $\mathcal{S}$ the set of all solution?

Theorem

Let A be enumeration problem. If the DecisionEnumeration problem associated to A is NP-hard, then $P \notin \text{TotalP}$ unless $P = NP$.

proof

- Assume that there exists a polynomial P such that A can be solved in $P(n+m)$. And let $\mathcal{S} := \{s_1, ..., s_k\}$ be a set of solutions.

proof

- Assume that there exists a polynomial P such that A can be solved in $P(n+m)$. And let $\mathcal{S} := \{s_1, ..., s_k\}$ be a set of solutions.
- Run the algorithm for $P(n,k)+1$ times.

proof

- Assume that there exists a polynomial P such that A can be solved in $P(n+m)$. And let $\mathcal{S} := \{s_1, ..., s_k\}$ be a set of solutions.
- Run the algorithm for $P(n,k)+1$ times.
- If the algorithm stops during this period, answer "Yes" to the question, all solutions are in $\mathcal{S}$.

proof

- Assume that there exists a polynomial P such that A can be solved in $P(n+m)$. And let $\mathcal{S} := \{s_1, ..., s_k\}$ be a set of solutions.
- Run the algorithm for $P(n,k)+1$ times.
- If the algorithm stops during this period, answer "Yes" to the question, all solutions are in $\mathcal{S}$.
- Otherwise, stop it and answer "No".

decision enumeration

Theorem

Let A be enumeration problem. If the DecisionEnumeration problem associated to A is NP-hard, then $P \notin \text{TotalP}$ unless $P = NP$.

By abuse of terminology, we say sometimes that an enumeration problem is NP-hard when its associated DecisionEnumeration problem is so.

tractability?

Is the TotalP notion a good definition of tractable enumeration problems?

tractability?

Is the TotalP notion a good definition of tractable enumeration problems?

Issue:
We may wait long time before the output of the first solution.
They may all be given at the end of the algorithm.

tractability?

Is the TotalP notion a good definition of tractable enumeration problems?

Issue:
We may wait long time before the output of the first solution.
They may all be given at the end of the algorithm.

Can we refine the class TotalP to solve this issue ?

Incremental-Polynomial algorithm

incremental algorithm

An incremental-polynomial algorithm is a special case of total-polynomial algorithm.

Definition

An enumeration algorithm is said to be incremental-polynomial if the delay between the output of the k^{th} solution and the $(k+1)^{\text{th}}$ solution is bounded by $poly(n+k)$ where $poly$ is a polynomial and n is the input size.

incremental algorithm

Definition

We call IncP the class of enumeration problems that admit a incremental-polynomial algorithm.

incremental algorithm

Definition

We call IncP the class of enumeration problems that admit a incremental-polynomial algorithm.

Proposition

$\textsc{IncP} \subseteq \textsc{TotalP}$

proof of the proposition

- If A is an incremental-polynomial algorithm, there exists a polynomial P such that the delay between the output of the k^{th} solution and the $(k+1)^{\text{th}}$ solution is bounded by $P(n+k)$
- If m is the total number of solutions, the total running time of the algorithm is

$$\sum_{i=1}^{m} P(n+i) \leq mP(n+m)$$

  which is polynomial in $n+m$

The classes IncP and TotalP are well separated

The classes IncP and TotalP are well separated

Some problems are known to be in TotalP but not in IncP.

The classes $\textsc{IncP}$ and $\textsc{TotalP}$ are well separated

Some problems are known to be in $\textsc{TotalP}$ but not in $\textsc{IncP}$.

Proposition

Unless $P = NP$, $\textsc{IncP} \subsetneq \textsc{TotalP}$ ($\textsc{IncP} \neq \textsc{TotalP}$).

NewSolution problem

A classical way to obtain an incremental-polynomial algorithm is to be able to solve a slightly different version of the DecisionEnumeration problem.

NEWSOLUTION problem

A classical way to obtain an incremental-polynomial algorithm is to be able to solve a slightly different version of the DECISIONENUMERATION problem.

NEWSOLUTION problem

input: A set of solutions $\{s_1, ..., s_k\}$

output: "No" if $\{s_1, ..., s_k\}$ contains all solutions

A new solution $s \notin \{s_1, ..., s_k\}$ otherwise

new solution problem

To obtain a incremental-polynomial algorithm, one can apply successively the NewSolution problem by adding the new solution found at each step to the set of already generated solutions until the answer is "No".

new solution problem

To obtain a incremental-polynomial algorithm, one can apply successively the NewSolution problem by adding the new solution found at each step to the set of already generated solutions until the answer is "No".

Proposition

Let A be an enumeration problem. Then $A \in \text{IncP}$ if and only if the NewSolution problem associated to A is polynomial.

new solution problem

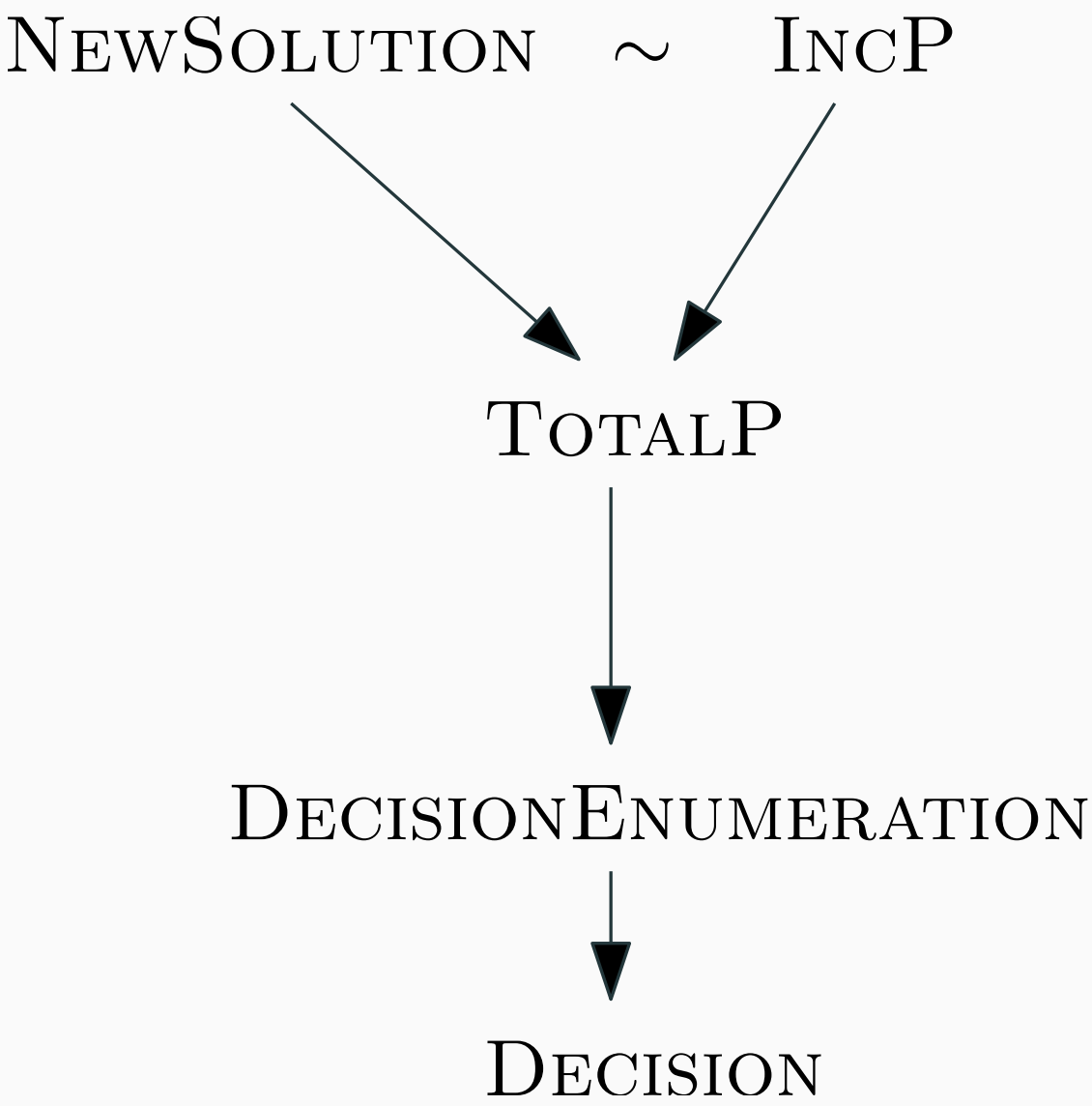
To obtain a incremental-polynomial algorithm, one can apply successively the NewSolution problem by adding the new solution found at each step to the set of already generated solutions until the answer is "No".

Proposition

Let A be an enumeration problem. Then $A \in \text{IncP}$ if and only if the NewSolution problem associated to A is polynomial.

Proof: Left as exercise.

summarize



example: spanning trees of a graph

Let G be a graph and let T be a spanning tree of G. The fundamental cycle of T with respect to an edge $e \in E(G) \setminus E(T)$, is the unique simple cycle $F(T, e)$ of $T \cup \{e\}$.

example: spanning trees of a graph

Let G be a graph and let T be a spanning tree of G. The fundamental cycle of T with respect to an edge $e \in E(G) \setminus E(T)$, is the unique simple cycle $F(T, e)$ of $T \cup \{e\}$.
Let us denote by $F(T)$ the set of fundamental cycles of T.

example: spanning trees of a graph

Let G be a graph and let T be a spanning tree of G. The fundamental cycle of T with respect to an edge $e \in E(G) \setminus E(T)$, is the unique simple cycle $F(T, e)$ of $T \cup \{e\}$.
Let us denote by $F(T)$ the set of fundamental cycles of T.

Proposition

A set $\mathcal{T}$ is the set of the spanning trees of a graph G, if and only if for all $T \in \mathcal{T}$ and for all edge $e \notin T$, the tree $(T \cup \{e\}) \setminus \{y\}$ belongs to $\mathcal{T}$ for all $y \in F(T, e)$.

homework: cycles of a graph

Let G be a graph, T one of its spanning tree and let $F(T)$ be the set of fundamental cycles of T.

Algorithm:

- output $F(T)$
- $S = F(T)$
- While $\exists\ C_1, C_2 \in S$, $C_1 \cap C_2 \neq \emptyset$ and $C_1 \Delta C_2 \notin S$:
  - output $C_1 \Delta C_2$
  - Add $C_1 \Delta C_2$ to S

homework: cycles of a graph

Let G be a graph, T one of its spanning tree and let $F(T)$ be the set of fundamental cycles of T.

Algorithm:

- output $F(T)$
- $S = F(T)$
- While $\exists\ C_1, C_2 \in S$, $C_1 \cap C_2 \neq \emptyset$ and $C_1 \Delta C_2 \notin S$:
  - output $C_1 \Delta C_2$
  - Add $C_1 \Delta C_2$ to S

1. Does this algorithm enumerate all cycles of the graph?

homework: cycles of a graph

Let G be a graph, T one of its spanning tree and let $F(T)$ be the set of fundamental cycles of T.

Algorithm:

- output $F(T)$
- $S = F(T)$
- While $\exists\ C_1, C_2 \in S$, $C_1 \cap C_2 \neq \emptyset$ and $C_1 \Delta C_2 \notin S$:
  - output $C_1 \Delta C_2$
  - Add $C_1 \Delta C_2$ to S

1. Does this algorithm enumerate all cycles of the graph?
2. Prove or disprove it.

homework: cycles of a graph

Let G be a graph, T one of its spanning tree and let $F(T)$ be the set of fundamental cycles of T.

Algorithm:

- output $F(T)$
- $S = F(T)$
- While $\exists\ C_1, C_2 \in S$, $C_1 \cap C_2 \neq \emptyset$ and $C_1 \Delta C_2 \notin S$:
  - output $C_1 \Delta C_2$
  - Add $C_1 \Delta C_2$ to S

1. Does this algorithm enumerate all cycles of the graph?
2. Prove or disprove it.
3. Is it total-polynomial? Incremental-polynomial?

an open problem

- A hypergraph $\mathcal{H} := (V, \mathcal{E})$ is a set family $\mathcal{E}$ on a ground set V.

an open problem

- A hypergraph $\mathcal{H} := (V, \mathcal{E})$ is a set family $\mathcal{E}$ on a ground set V.
- A transversal (or hitting set) of $\mathcal{H}$ is a subset T of V that intersects every $E \in \mathcal{E}$. It is minimal, if it does not contain any other transversal.

an open problem

- A hypergraph $\mathcal{H} := (V, \mathcal{E})$ is a set family $\mathcal{E}$ on a ground set V.
- A transversal (or hitting set) of $\mathcal{H}$ is a subset T of V that intersects every $E \in \mathcal{E}$. It is minimal, if it does not contain any other transversal.

an open problem

- A hypergraph $\mathcal{H} := (V, \mathcal{E})$ is a set family $\mathcal{E}$ on a ground set V.
- A transversal (or hitting set) of $\mathcal{H}$ is a subset T of V that intersects every $E \in \mathcal{E}$. It is minimal, if it does not contain any other transversal.

Transversal DecisionEnumeration problem

input: A hypergraph $\mathcal{H}$ and a set of minimal transversals of $\mathcal{H}$ $\mathcal{S} := \{T_1, ..., T_k\}$

Question: Is $\mathcal{S}$ the set of all minimal transversals of $\mathcal{H}$?

transversal problem

Transversal DecisionEnumeration problem

input: A hypergraph $\mathcal{H}$ and a set of minimal transversals of $\mathcal{H}$ $\mathcal{S} := \{T_1, ..., T_k\}$

Question: Is $\mathcal{S}$ the set of all minimal transversals of $\mathcal{H}$?

The complexity of the above problem is open.

transversal problem

Transversal DecisionEnumeration problem

input: A hypergraph $\mathcal{H}$ and a set of minimal transversals of $\mathcal{H}$ $\mathcal{S} := \{T_1, ..., T_k\}$

Question: Is $\mathcal{S}$ the set of all minimal transversals of $\mathcal{H}$?

The complexity of the above problem is open.

But we know that the NewSolution problem is quasi-polynomial (there is an algorithm of complexity N^{logN} where $N = k + |\mathcal{H}|$)

transversal DECISIONENUMERATION problem

Therefore :

- The minimal transversal of a hypergraph can be enumerated in incremental quasi-polynomial time
- Unless every NP-complete problem can be solved in quasi-polynomial time, the transversal DecisionEnumeration problem is not Np-complete.
- Open: Is the transversal (DecisionEnumeration/ NewSolution) problem polynomial?

example of problems in INCP

The enumeration of

- Spanning trees
- (Simple) Cycles
- (maximal) Paths, induced paths ...
- (maximal/maximum) matching
- maximal Independent sets
- maximal cliques/bicliques
- minimal vertex covers
- minimal feedback vertex/arc set
- ...

example np-hard enumeration problems

The enumeration of

- Minimal dicut of an oriented graph.
- Vertices of a polyhedron given by a set of inequalities.
- Maximal elements of an independent system given by an oracle.
- Any problem for which the (classical) decision problem is NP-hard.
- ...

equivalent to minimal transversals enumeration

The enumeration of

- Maximal stable sets of a hypergraph.
- All satisfying assignments of a monotone CNF.
- Minimal dominating sets of a graph.
- Minimal subsets that are not included in any set from a given family of sets.
- ...

Polynomial delay algorithm

issues

Issues of the Incremental algorithms:

- The delay between two consecutive solutions is increasing as more solutions are outputted.
- The space used by the algorithm is usually exponential in the input size.

polynomial delay algorithm

Definition

A polynomial delay algorithm is an enumeration algorithm such that the delay between two consecutive outputs is bounded by $poly(n)$ where $poly$ is a polynomial and n is the input size.

Supergraph method

supergraph method

A common way to obtain polynomial algorithms is the so called supergraph method.

Idea:

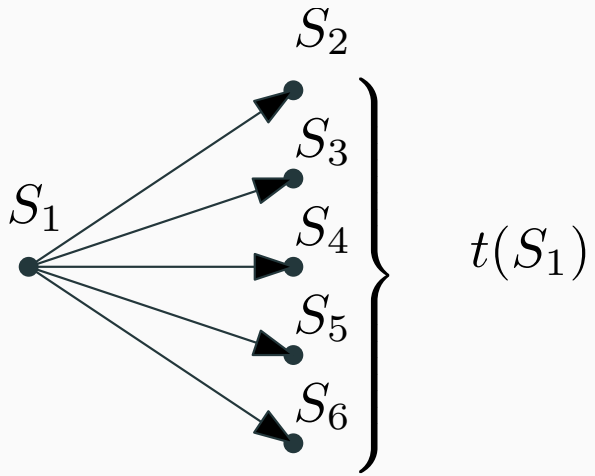
- Make small modifications on solutions to obtain new ones.
- Prove that all solutions can be obtained in this way starting from a special solution (or set of solutions).
- Try to avoid redundancy (output several times the same solution) using only polynomial space.

supergraph method

Let $\mathcal{S}$ be the set of solutions of a given enumeration problem.

The supergraph method consists on defining a transition function $t : \mathcal{S} \to 2^{\mathcal{S}}$.

It defines the arcs of an oriented graph with vertex set $\mathcal{S}$

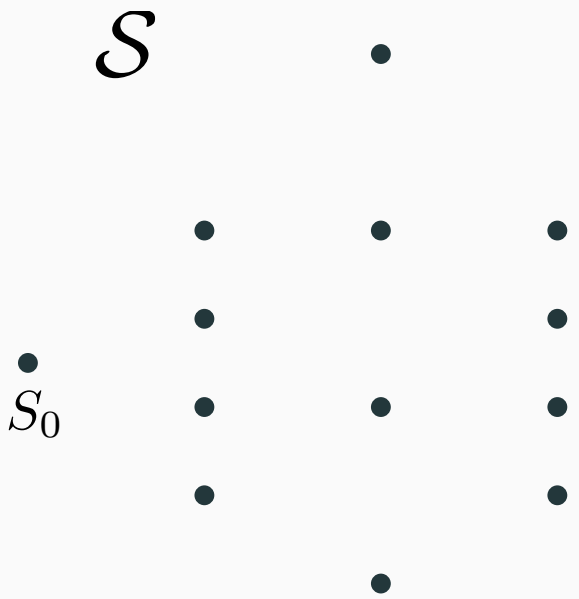


supergraph method

Let $\mathcal{S}$ be the set of solutions of a given enumeration problem.

The supergraph method consists on defining a transition function $t : \mathcal{S} \to 2^{\mathcal{S}}$.

This function defines the arcs of an oriented graph with vertex set $\mathcal{S}$

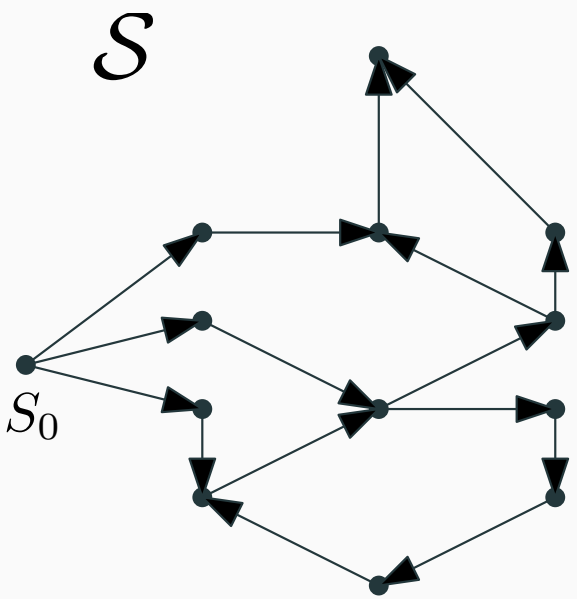


supergraph method

Let $\mathcal{S}$ be the set of solutions of a given enumeration problem.

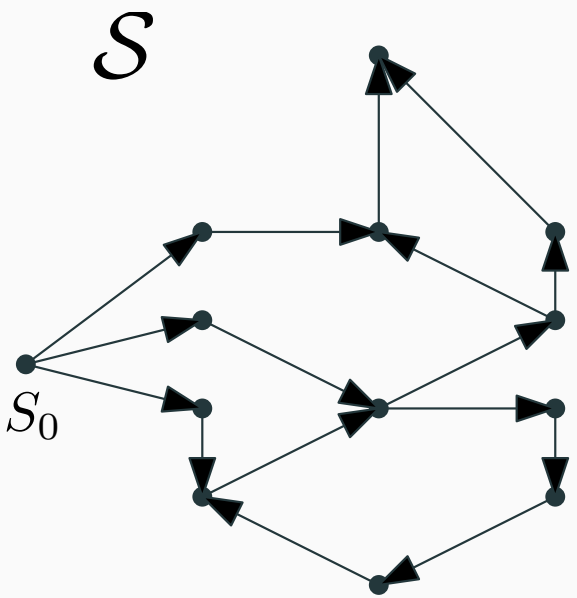
The supergraph method consists on defining a transition function $t : \mathcal{S} \to 2^{\mathcal{S}}$.

This function defines the arcs of an oriented graph with vertex set $\mathcal{S}$



supergraph method

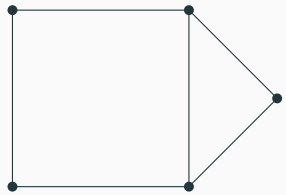
Assuming that the transition function is such that every solution is accessible from the starting solution, we would like to explore the graph (for instance with a DFS or a BFS) to find all solutions.

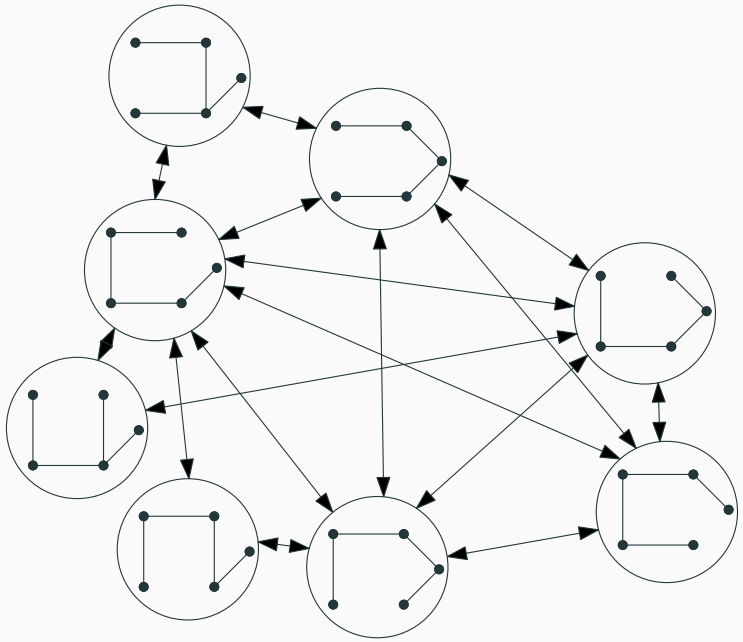


exemple: spanning trees

For a spanning tree T of a graph G, we define the following transition function:

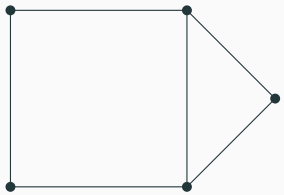
$$t(T) := \{(T \cup \{e\}) \setminus \{e'\} : e \notin T,\ e' \in F(T, e)\}$$

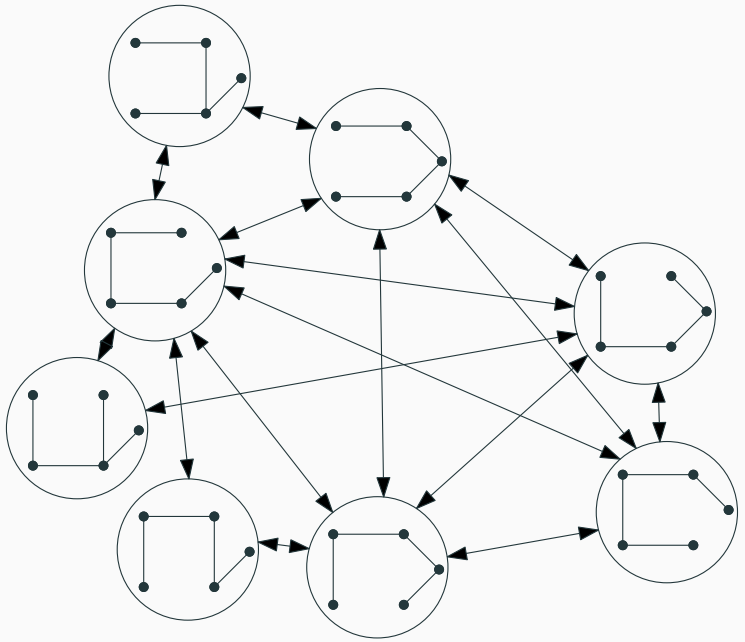




Issues:

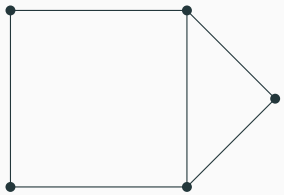
- When you explore a graph, you have to store the set of already visited vertices.
- Here the number of vertices of a supergraph corresponds to the number of solutions which is huge in general.

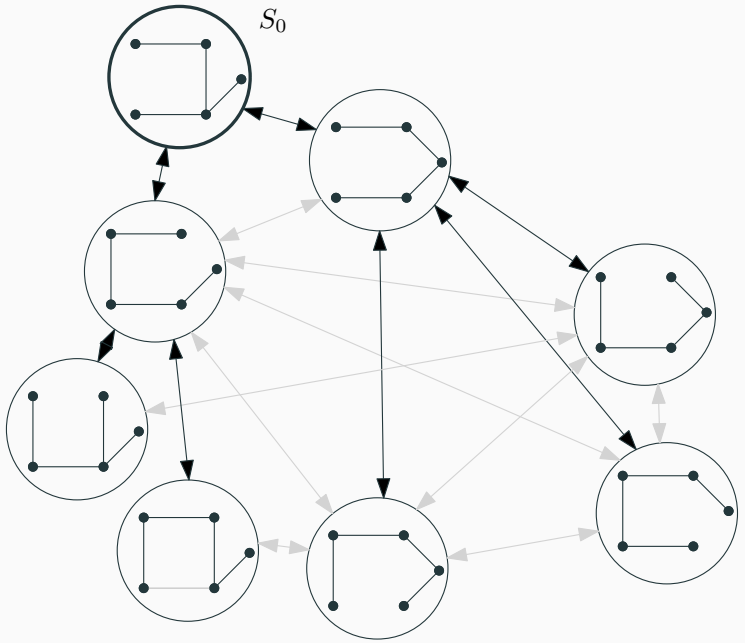




Solution:

- Restrict the transition function so that it defines an arborescence.
- The goal is to define a parent-child relation between solutions so that each solution will be "produced" only by its father





reverse search

To define an arborescence, we usually use the reverse search method.

- Define a total ordering over solutions.
- The father $P(S)$ of a solution S is defined as the smallest solution S' such that $S \in t(S')$

Then we restrict the transition function t to t' :

$$t'(T) := \{T' \in t(T) : P(T') = T\}$$

spanning trees

- Consider an ordering over the edges of G : $(e_1, ..., e_k)$.
- The ordering over spanning trees is given by the lexicographic order.
- The father $P(T)$ of a spanning tree T is the spanning tree $(T \cup \{e\}) \setminus \{e'\}$ where e is smallest edge that does not belong to T and e' is the largest edge of $F(T, e)$

algorithm

We call the algorithm with the starting solution S_0.

ReverseSearch(S):

- Output S
- For all $S' \in t(S)$
  - If $P(S') = S$
    - ReverseSearch(S')

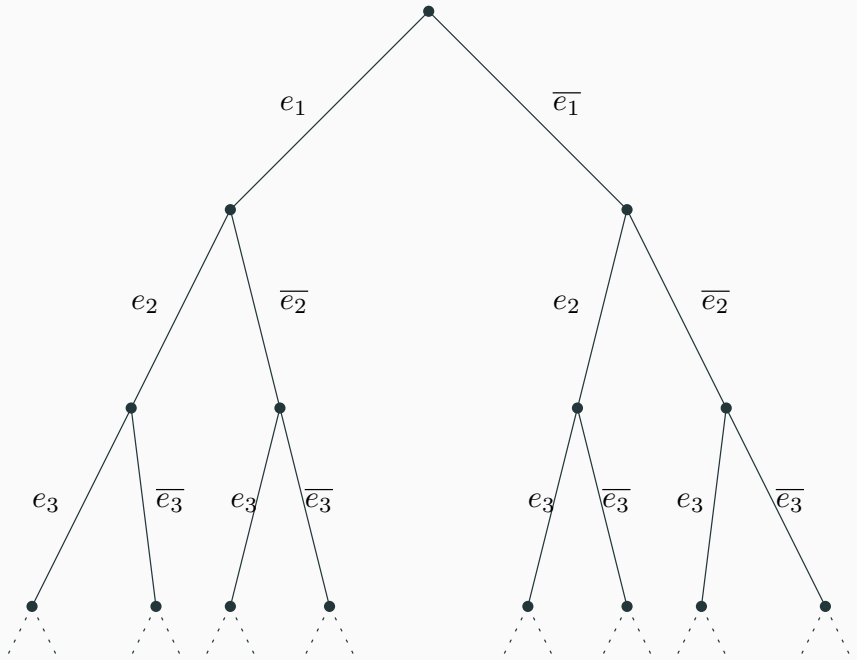
Backtracking method

flashlight

Here we assume that the solutions of the problem are subsets of a ground set E.

Steps

- The solutions corresponds to leaves of the tree.

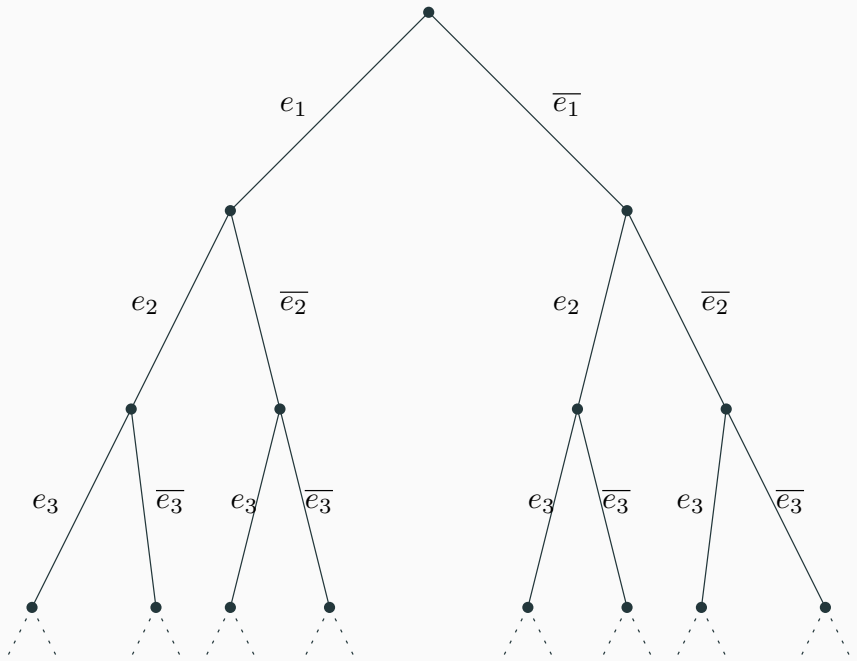


flashlight

Here we assume that the solutions of the problem are subsets of a ground set E.

Steps

- The solutions corresponds to leaves of the tree.
- Each level i corresponds to the choice of the selection of e_i in the solution.

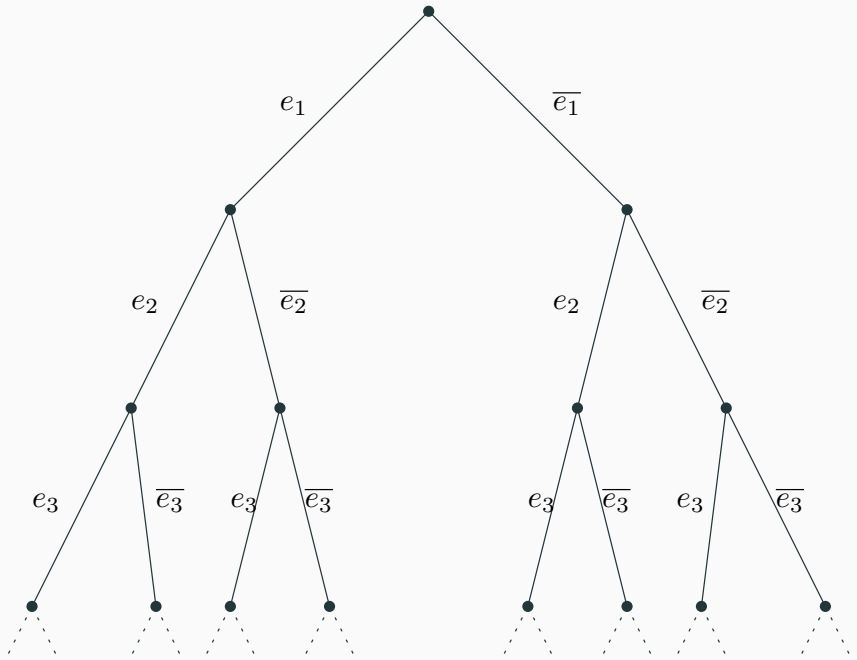


flashlight

Here we assume that the solutions of the problem are subsets of a ground set E.

Steps

- The solutions corresponds to leaves of the tree.
- Each level i corresponds to the choice of the selection of e_i in the solution.
- All the solutions in the left tree contain e_i and the ones of the right tree do not contain e_i.



extension problem

To use the backtracking method, we need to be able to solve the following problem.

Extension problem

Input: Two sets $X, Y \subseteq E$

Question: Is there a solution S such that $X \subseteq S$ and $Y \cap S = \emptyset$?

If the extension problem is polynomial, then the enumeration problem can be solved with polynomial delay.

algorithm

$\textsc{BacktrackAlgorithm}(i, X, Y)$:

- if $i = n$:
  - output X
- if there is a solution S such that $(X \cup \{e_i\}) \subseteq S$ and $Y \cap S = \emptyset$:
  - $\textsc{BacktrackAlgorithm}(i + 1, X \cup \{e_i\}, Y)$
- if there is a solution S such that $X \subseteq S$ and $(Y \cup \{e_i\}) \cap S = \emptyset$:
  - $\textsc{BacktrackAlgorithm}(i + 1, X, Y \cup \{e_i\})$

exercise

- Why the previous algorithm enumerates all solutions with polynomial delay (Assuming that the extension problem is polynomial)?
- Is the Extension problem is polynomial for spanning trees?
- Is it polynomial for the cycles of a graph?