Input-Sensitive Enumerations

Petr Golovach

Department of Informatics, University of Bergen

SGT 2018, Sète, France, 13.06.2018

Enumerations

An *enumeration problem* require to list all wanted objects for a given input.

Enumerations

An *enumeration problem* require to list all wanted objects for a given input.

For example:

Enumerations

An *enumeration problem* require to list all wanted objects for a given input.

For example:

- list all subset of vertices or edges of a given graph that satisfy a certain condition:
  - List all maximal independent sets.
  - List all perfect matchings.
  - ...

Enumerations

An *enumeration problem* require to list all wanted objects for a given input.

For example:

- list all subset of vertices or edges of a given graph that satisfy a certain condition:
  - List all maximal independent sets.
  - List all perfect matchings.
  - ...
- List all satisfying assignments of variables for a Boolean formula.

Enumerations

An *enumeration problem* require to list all wanted objects for a given input.

For example:

- list all subset of vertices or edges of a given graph that satisfy a certain condition:
    - List all maximal independent sets.
    - List all perfect matchings.
    - ...
- List all satisfying assignments of variables for a Boolean formula.
- List all triangulation for a set of point on the plane.

Enumerations

An *enumeration problem* require to list all wanted objects for a given input.

For example:

- list all subset of vertices or edges of a given graph that satisfy a certain condition:
  - List all maximal independent sets.
  - List all perfect matchings.
  - ...
- List all satisfying assignments of variables for a Boolean formula.
- List all triangulation for a set of point on the plane.
- ...

Input-sensitive vs. output-sensitive enumerations

1. *Output-sensitive*

Input-sensitive vs. output-sensitive enumerations

1. *Output-sensitive*
   - The complexity is measured as a function of both the input size and the size of the output.

Input-sensitive vs. output-sensitive enumerations

1. *Output-sensitive*
   - The complexity is measured as a function of both the input size and the size of the output.
   - The aim is to find an algorithm that is polynomial for this measure.

Input-sensitive vs. output-sensitive enumerations

1. *Output-sensitive*
   - The complexity is measured as a function of both the input size and the size of the output.
   - The aim is to find an algorithm that is polynomial for this measure.
2. *Input-sensitive*

Input-sensitive vs. output-sensitive enumerations

1. *Output-sensitive*
   - The complexity is measured as a function of both the input size and the size of the output.
   - The aim is to find an algorithm that is polynomial for this measure.
2. *Input-sensitive*
   - The complexity is measured as a function of the input size only.

Input-sensitive enumeration

- The running time depends on the length of the input only (e.g., the number of vertices of the input graph).

Input-sensitive enumeration

- The running time depends on the length of the input only (e.g., the number of vertices of the input graph).
- We use the classical worst case running time analysis.

Input-sensitive enumeration

- The running time depends on the length of the input only (e.g., the number of vertices of the input graph).
- We use the classical worst case running time analysis.
- If the number of objects to be enumerated is *exponential* (in the worst case), then an input-sensitive enumeration algorithm runs in *exponential* time.

Input-sensitive enumeration

- The running time depends on the length of the input only (e.g., the number of vertices of the input graph).
- We use the classical worst case running time analysis.
- If the number of objects to be enumerated is *exponential* (in the worst case), then an input-sensitive enumeration algorithm runs in *exponential* time.
- We use *exact exponential-time algorithms*,

Input-sensitive enumeration

- The running time depends on the length of the input only (e.g., the number of vertices of the input graph).
- We use the classical worst case running time analysis.
- If the number of objects to be enumerated is *exponential* (in the worst case), then an input-sensitive enumeration algorithm runs in *exponential* time.
- We use *exact exponential-time algorithms*, in particular *branching algorithms*.

Input-sensitive enumeration

An input-sensitive algorithm solving an enumeration problem

Input-sensitive enumeration

An input-sensitive algorithm solving an enumeration problem

- Can be used to solve the decision, optimization and counting versions of the problem.

Input-sensitive enumeration

An input-sensitive algorithm solving an enumeration problem

- Can be used to solve the decision, optimization and counting versions of the problem.
- The running time of the algorithm provides an *upper bound for the maximum number of enumerated objects*.

Input-sensitive enumeration

An input-sensitive algorithm solving an enumeration problem

- Can be used to solve the decision, optimization and counting versions of the problem.
- The running time of the algorithm provides an *upper bound for the maximum number of enumerated objects*.
- Moreover, the algorithm can be used to obtain better bounds.

Upper and Lower bounds

Suppose that there is a family of instances $\mathcal{I}$ of an enumeration problem such that for every $n \in \mathbb{N}$, $\mathcal{I}$ contains an instance I with $|I| = n$ and the number of enumerated objects for $I \in \mathcal{I}$ is $f(|I|)$.

Upper and Lower bounds

Suppose that there is a family of instances $\mathcal{I}$ of an enumeration problem such that for every $n \in \mathbb{N}$, $\mathcal{I}$ contains an instance I with $|I| = n$ and the number of enumerated objects for $I \in \mathcal{I}$ is $f(|I|)$.

Then $f(n)$ provides an unconditional *running time lower bound* for every enumeration algorithm.

Upper and Lower bounds

Suppose that there is a family of instances $\mathcal{I}$ of an enumeration problem such that for every $n \in \mathbb{N}$, $\mathcal{I}$ contains an instance I with $|I| = n$ and the number of enumerated objects for $I \in \mathcal{I}$ is $f(|I|)$.

Then $f(n)$ provides an unconditional *running time lower bound* for every enumeration algorithm.

Our aim is

Upper and Lower bounds

Suppose that there is a family of instances $\mathcal{I}$ of an enumeration problem such that for every $n \in \mathbb{N}$, $\mathcal{I}$ contains an instance I with $|I| = n$ and the number of enumerated objects for $I \in \mathcal{I}$ is $f(|I|)$.

Then $f(n)$ provides an unconditional *running time lower bound* for every enumeration algorithm.

Our aim is

- Construct an enumeration algorithm with the "best" running time.

Upper and Lower bounds

Suppose that there is a family of instances $\mathcal{I}$ of an enumeration problem such that for every $n \in \mathbb{N}$, $\mathcal{I}$ contains an instance I with $|I| = n$ and the number of enumerated objects for $I \in \mathcal{I}$ is $f(|I|)$.

Then $f(n)$ provides an unconditional *running time lower bound* for every enumeration algorithm.

Our aim is

- Construct an enumeration algorithm with the “best” running time.
- Construct the “best” lower bound.

Upper and Lower bounds

Suppose that there is a family of instances $\mathcal{I}$ of an enumeration problem such that for every $n \in \mathbb{N}$, $\mathcal{I}$ contains an instance I with $|I| = n$ and the number of enumerated objects for $I \in \mathcal{I}$ is $f(|I|)$.

Then $f(n)$ provides an unconditional *running time lower bound* for every enumeration algorithm.

Our aim is

- Construct an enumeration algorithm with the "best" running time.
- Construct the "best" lower bound.
- Ideally, we wish to get (asymptotically) tight upper and lower bounds for running time and combinatorial bounds for the number of enumerated objects.

Plan of the lectures

- Introduction to branching enumeration algorithms and their analysis.

Plan of the lectures

- Introduction to branching enumeration algorithms and their analysis.
- Advanced analysis of branching algorithms; the “Measure and Conquer” technique.

Plan of the lectures

- Introduction to branching enumeration algorithms and their analysis.
- Advanced analysis of branching algorithms; the "Measure and Conquer" technique.
- Lower bounds.

Plan of the lectures

- Introduction to branching enumeration algorithms and their analysis.
- Advanced analysis of branching algorithms; the “Measure and Conquer” technique.
- Lower bounds.
- Conclusions and open problems.

Plan of the lectures

- Introduction to branching enumeration algorithms and their analysis.
- Advanced analysis of branching algorithms; the “Measure and Conquer” technique.
- Lower bounds.
- Conclusions and open problems.

Branching algorithms

The majority of input-sensitive enumeration algorithms are *Recursive branching algorithms*.

Branching algorithms

The majority of input-sensitive enumeration algorithms are *Recursive branching algorithms*.

These algorithms are also called

Branching algorithms

The majority of input-sensitive enumeration algorithms are *Recursive branching algorithms*.

These algorithms are also called

- Branch & bound algorithms.

Branching algorithms

The majority of input-sensitive enumeration algorithms are *Recursive branching algorithms*.

These algorithms are also called

- Branch & bound algorithms.
- Backtracking algorithms.

Branching algorithms

The majority of input-sensitive enumeration algorithms are *Recursive branching algorithms*.

These algorithms are also called

- Branch & bound algorithms.
- Backtracking algorithms.
- Search tree algorithms.

Branching algorithms

The majority of input-sensitive enumeration algorithms are *Recursive branching algorithms*.

These algorithms are also called

- Branch & bound algorithms.
- Backtracking algorithms.
- Search tree algorithms.
- Branch & reduce algorithms.

Branching algorithms

The majority of input-sensitive enumeration algorithms are *Recursive branching algorithms*.

These algorithms are also called

- Branch & bound algorithms.
- Backtracking algorithms.
- Search tree algorithms.
- Branch & reduce algorithms.
- Splitting algorithms

Branching algorithms

Pros

- Branching algorithms could be simple and easy to implement.

Branching algorithms

Pros

- Branching algorithms could be simple and easy to implement.
- Typically, they use polynomial space.

Branching algorithms

Pros

- Branching algorithms could be simple and easy to implement.
- Typically, they use polynomial space.
- Could be efficient in practice.

Branching algorithms

Pros

- Branching algorithms could be simple and easy to implement.
- Typically, they use polynomial space.
- Could be efficient in practice.

Cons

- Difficult to analyze.

Branching algorithms

Pros

- Branching algorithms could be simple and easy to implement.
- Typically, they use polynomial space.
- Could be efficient in practice.

Cons

- Difficult to analyze.
- Could be difficult to apply for some classes of problems.

Maximal independent sets

A set of vertices X of a graph G is *independent* if the vertices of X are pairwise non-adjacent.

Maximal independent sets

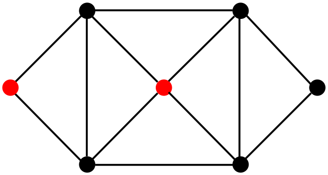
A set of vertices X of a graph G is *independent* if the vertices of X are pairwise non-adjacent.

An independent set X is *(inclusion) maximal* if every $Y \supset X$ is not independent.

Maximal independent sets

A set of vertices X of a graph G is *independent* if the vertices of X are pairwise non-adjacent.

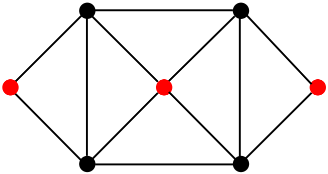
An independent set X is *(inclusion) maximal* if every $Y \supset X$ is not independent.



Maximal independent sets

A set of vertices X of a graph G is *independent* if the vertices of X are pairwise non-adjacent.

An independent set X is *(inclusion) maximal* if every $Y \supset X$ is not independent.



Maximal independent sets

Problem (Maximal Independent Set Enumeration)

Input: *A graph G.*

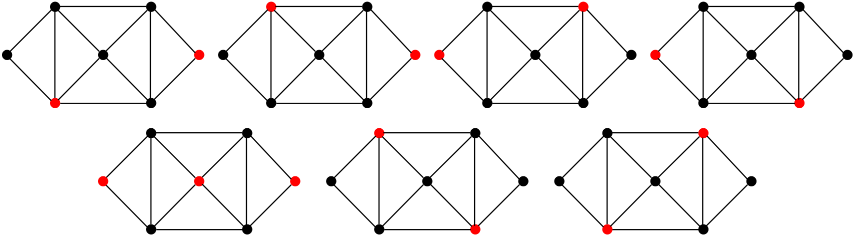
Task: *Enumerate all maximal independent sets of G.*

Maximal independent sets

Problem (Maximal Independent Set Enumeration)

Input: *A graph G.*

Task: *Enumerate all maximal independent sets of G.*



Enumeration of MIS

Enum MIS(G, S)

Input : A graph G, a set S of vertices already selected to be in a MIS, $S \cap V(G) = \emptyset$

Output: All sets $X \cup S$ for MIS S of G.

if $V(G) = \emptyset$ **then**
 output S
else
 find a vertex $v \in V(G)$ of minimum degree d;
 call **Enum MIS**$(G - N_G[v], S \cup \{v\})$;
 for $x \in N_G(v)$ **do**
  call **Enum MIS**$(G - N_G[x], S \cup \{x\})$
 end
end

Enumeration of MIS

Enum MIS(G, S)

Input : A graph G, a set S of vertices already selected to be in a MIS, $S \cap V(G) = \emptyset$
Output: All sets $X \cup S$ for MIS S of G.

if $V(G) = \emptyset$ **then**
  output S
else
  find a vertex $v \in V(G)$ of minimum degree d;
  call **Enum MIS**$(G - N_G[v], S \cup \{v\})$;
  for $x \in N_G(v)$ **do**
    call **Enum MIS**$(G - N_G[x], S \cup \{x\})$
  end
end

Call **Enum MIS**$(G, \emptyset)$ to enumerate all MIS of G.

Correctness

Observation: Let X be an independent set in G, then

Correctness

Observation: Let X be an independent set in G, then

- for every $v \in X$, $N_G(v) \cap X = \emptyset$,

Correctness

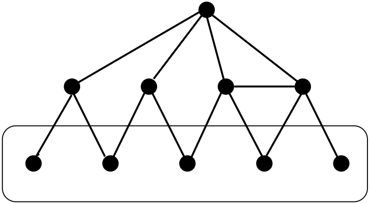
Observation: Let X be an independent set in G, then

- for every $v \in X$, $N_G(v) \cap X = \emptyset$,
- X is maximal if and only if for every $v \in V(G)$, $N_G[v] \cap X \neq \emptyset$.

Correctness

Observation: Let X be an independent set in G, then

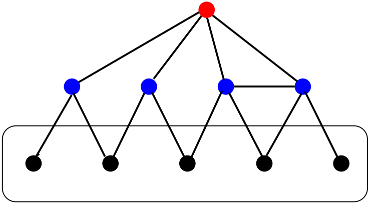
- for every $v \in X$, $N_G(v) \cap X = \emptyset$,
- X is maximal if and only if for every $v \in V(G)$, $N_G[v] \cap X \neq \emptyset$.



Correctness

Observation: Let X be an independent set in G, then

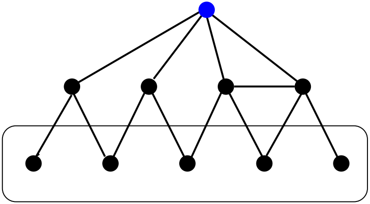
- for every $v \in X$, $N_G(v) \cap X = \emptyset$,
- X is maximal if and only if for every $v \in V(G)$, $N_G[v] \cap X \neq \emptyset$.



Correctness

Observation: Let X be an independent set in G, then

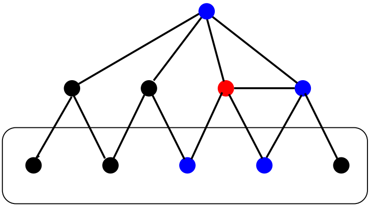
- for every $v \in X$, $N_G(v) \cap X = \emptyset$,
- X is maximal if and only if for every $v \in V(G)$, $N_G[v] \cap X \neq \emptyset$.



Correctness

Observation: Let X be an independent set in G, then

- for every $v \in X$, $N_G(v) \cap X = \emptyset$,
- X is maximal if and only if for every $v \in V(G)$, $N_G[v] \cap X \neq \emptyset$.



Correctness

Enum MIS(G, S)

Input : A graph G, a set S pf vertices already selected to be in a MIS, $S \cap V(G) = \emptyset$
Output: All sets $X \cup S$ for MIS S of G.

```
if V(G) = ∅ then
| output S
else
| find a vertex v ∈ V(G) of minimum degree d;
| call Enum MIS(G − N_G[v], S ∪ {v});
| for x ∈ N_G(v) do
| | call Enum MIS(G − N_G[x], S ∪ {x})
| end
end
```

Correctness

Enum MIS(G, S)

Input : A graph G, a set S pf vertices already selected to be in a MIS, $S \cap V(G) = \emptyset$

Output: All sets $X \cup S$ for MIS S of G.

if $V(G) = \emptyset$ **then**
  output S
else
  find a vertex $v \in V(G)$ of minimum degree d;
  call **Enum MIS**$(G - N_G[v], S \cup \{v\})$;
  for $x \in N_G(v)$ **do**
    call **Enum MIS**$(G - N_G[x], S \cup \{x\})$
  end
end

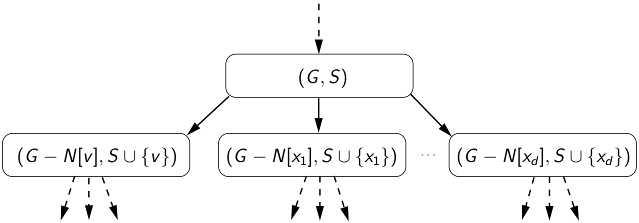
Exercise: Give a formal inductive correctness proof.

Running time

Consider the search tree produced by the algorithm.

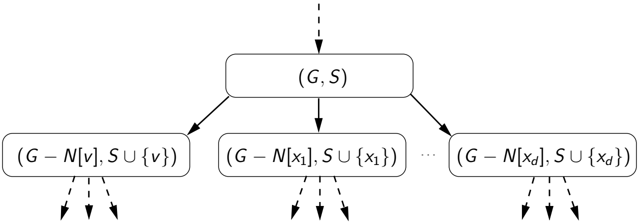
Running time

Consider the search tree produced by the algorithm.



Running time

Consider the search tree produced by the algorithm.



Observe that the algorithm produced maximal independent sets in the leaves of the search tree.

Running time

Denote by $L(n)$ the maximum number of leaves of a search tree for the input graph with n vertices.

Running time

Denote by $L(n)$ the maximum number of leaves of a search tree for the input graph with n vertices.

Note that $L(n)$ is a non-decreasing function and $L(0) = 1$.

Running time

Denote by $L(n)$ the maximum number of leaves of a search tree for the input graph with n vertices.

Note that $L(n)$ is a non-decreasing function and $L(0) = 1$.

Consider G that gives the maximum number of leaves.

Running time

Denote by $L(n)$ the maximum number of leaves of a search tree for the input graph with n vertices.

Note that $L(n)$ is a non-decreasing function and $L(0) = 1$.

Consider G that gives the maximum number of leaves.

Let $x_1, \ldots, x_d$ be the neighbors of v chosen by the algorithm.

Running time

Denote by $L(n)$ the maximum number of leaves of a search tree for the input graph with n vertices.

Note that $L(n)$ is a non-decreasing function and $L(0) = 1$.

Consider G that gives the maximum number of leaves.

Let $x_1, \ldots, x_d$ be the neighbors of v chosen by the algorithm.

$$L(n) \leq L(n - |N_G[v]|) + L(n - |N_G[x_1]|) + \ldots + L(n - |N_G[x_d]|).$$

Running time

Denote by $L(n)$ the maximum number of leaves of a search tree for the input graph with n vertices.

Note that $L(n)$ is a non-decreasing function and $L(0) = 1$.

Consider G that gives the maximum number of leaves.

Let $x_1, \ldots, x_d$ be the neighbors of v chosen by the algorithm.

$$L(n) \leq L(n - |N_G[v]|) + L(n - |N_G[x_1]|) + \ldots + L(n - |N_G[x_d]|).$$

Recall that

$$d = d_G(v) \leq d_G(x_1), \ldots, d_G(x_d).$$

Running time

Denote by $L(n)$ the maximum number of leaves of a search tree for the input graph with n vertices.

Note that $L(n)$ is a non-decreasing function and $L(0) = 1$.

Consider G that gives the maximum number of leaves.

Let $x_1, \ldots, x_d$ be the neighbors of v chosen by the algorithm.

$$L(n) \leq L(n - |N_G[v]|) + L(n - |N_G[x_1]|) + \ldots + L(n - |N_G[x_d]|).$$

Recall that

$$d = d_G(v) \leq d_G(x_1), \ldots, d_G(x_d).$$

Therefore,

$$L(n) \leq (d+1)L(n - (d+1)).$$

Running time

The recurrences:

$$\begin{aligned} L(0) &= 1, \\ L(n) &\leq (d+1)L(n-(d+1)). \end{aligned}$$

Running time

The recurrences:

$$\begin{aligned} L(0) &= 1, \\ L(n) &\leq (d+1)L(n-(d+1)). \end{aligned}$$

Claim: $L(n) \leq 3^{n/3}$.

Running time

The recurrences:

$$\begin{aligned} L(0) &= 1, \\ L(n) &\le (d+1)L(n-(d+1)). \end{aligned}$$

Claim: $L(n) \le 3^{n/3}$.

$L(0) = 1 = 3^{0/3}$.

Running time

The recurrences:

$$
\begin{aligned}
L(0) &= 1, \\
L(n) &\leq (d+1)L(n-(d+1)).
\end{aligned}
$$

Claim: $L(n) \leq 3^{n/3}$.

$L(0) = 1 = 3^{0/3}$.

Exercise: Show that

$$\max_{k \in \mathbb{N}} k \cdot 3^{-k/3} = 1$$

and the maximum is achieved for $k = 3$.

Running time

The recurrences:

$$L(0) = 1,$$
$$L(n) \leq (d+1)L(n-(d+1)).$$

Claim: $L(n) \leq 3^{n/3}$.

$L(0) = 1 = 3^{0/3}$.

Exercise: Show that

$$\max_{k \in \mathbb{N}} k \cdot 3^{-k/3} = 1$$

and the maximum is achieved for $k = 3$.

$$L(n) \leq (d+1)L(n-(d+1)) \leq (d+1)3^{(n-(d+1))/3}$$
$$= (d+1)3^{(d+1)/3} \cdot 3^{n/3} \leq 3^{n/3}.$$

Running time

We have that the search tree for an n-vertex graph G has at most $3^{n/3}$ leaves.

Running time

We have that the search tree for an n-vertex graph G has at most $3^{n/3}$ leaves.

Exercise: Show that a rooted tree (arborescence/outbranching) with ℓ leaves has at most $2\ell - 1$ nodes that are either leaves or have at least 2 children.

Running time

We have that the search tree for an n-vertex graph G has at most $3^{n/3}$ leaves.

Exercise: Show that a rooted tree (arborescence/outbranching) with ℓ leaves has at most $2\ell - 1$ nodes that are either leaves or have at least 2 children.

Note that if $d = 0$, then we do not branch in **Enum MIS**(G, S).

Running time

We have that the search tree for an n-vertex graph G has at most $3^{n/3}$ leaves.

Exercise: Show that a rooted tree (arborescence/outbranching) with ℓ leaves has at most $2\ell - 1$ nodes that are either leaves or have at least 2 children.

Note that if $d = 0$, then we do not branch in **Enum MIS**(G, S).

The algorithm performs $O(n + m)$ operations per node that is either a leaf of has at least 2 children.

Running time

We have that the search tree for an n-vertex graph G has at most $3^{n/3}$ leaves.

Exercise: Show that a rooted tree (arborescence/outbranching) with ℓ leaves has at most $2\ell - 1$ nodes that are either leaves or have at least 2 children.

Note that if $d = 0$, then we do not branch in **Enum MIS**(G, S).

The algorithm performs $O(n + m)$ operations per node that is either a leaf of has at least 2 children.

The running time is $O(3^{n/3} \cdot (n + m))$

Running time

We have that the search tree for an n-vertex graph G has at most $3^{n/3}$ leaves.

Exercise: Show that a rooted tree (arborescence/outbranching) with ℓ leaves has at most $2\ell - 1$ nodes that are either leaves or have at least 2 children.

Note that if $d = 0$, then we do not branch in **Enum MIS**(G, S).

The algorithm performs $O(n + m)$ operations per node that is either a leaf of has at least 2 children.

The running time is $O(3^{n/3} \cdot (n + m))$ or $O^*(3^{n/3})$.

Running time

We have that the search tree for an n-vertex graph G has at most $3^{n/3}$ leaves.

Exercise: Show that a rooted tree (arborescence/outbranching) with ℓ leaves has at most $2\ell - 1$ nodes that are either leaves or have at least 2 children.

Note that if $d = 0$, then we do not branch in **Enum MIS**(G, S).

The algorithm performs $O(n + m)$ operations per node that is either a leaf of has at least 2 children.

The running time is $O(3^{n/3} \cdot (n + m))$ or $O^*(3^{n/3})$.

We write $f(n) = O^*(g(n))$ to denote that there is a polynomial $p(n)$ such that $f(n) \leq g(n)p(n)$.

Enumeration of MIS

Theorem

Maximal independent sets of an n-vertex graph can be enumerated in time $O^*(3^{n/3})$ ($O(1.4423^n)$).

Enumeration of MIS

Theorem

Maximal independent sets of an n-vertex graph can be enumerated in time $O^*(3^{n/3})$ ($O(1.4423^n)$).

Note that $3^{1/3} \approx 1.44225\ldots < 1.4423$.

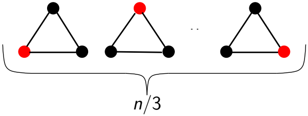
Enumeration of MIS

Theorem

Maximal independent sets of an n-vertex graph can be enumerated in time $O^*(3^{n/3})$ ($O(1.4423^n)$).

Note that $3^{1/3} \approx 1.44225\ldots < 1.4423$.

Lower bound:



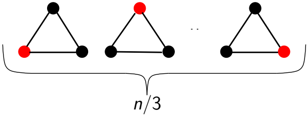
Enumeration of MIS

Theorem

Maximal independent sets of an n-vertex graph can be enumerated in time $O^*(3^{n/3})$ $(O(1.4423^n))$.

Note that $3^{1/3} \approx 1.44225\ldots < 1.4423$.

Lower bound:



This graph has $3^{n/3}$ maximal independent sets.

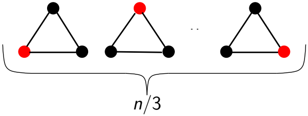
Enumeration of MIS

Theorem

Maximal independent sets of an n-vertex graph can be enumerated in time $O^*(3^{n/3})$ ($O(1.4423^n)$).

Note that $3^{1/3} \approx 1.44225\ldots < 1.4423$.

Lower bound:



This graph has $3^{n/3}$ maximal independent sets.

We can enumerate MIS in time $O^*(3^{n/3})$ but cannot do it faster than $\Omega(3^{n/3})$.

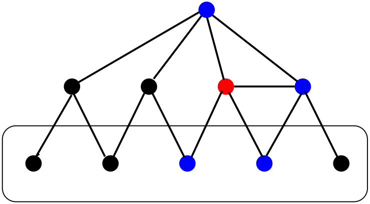
Enumeration of MIS

Since the depth of the search tree is at most n, **Enum MIS** uses polynomial space.

Enumeration of MIS

Since the depth of the search tree is at most n, **Enum MIS** uses polynomial space.

Warning: The algorithm can produce duplicates.



Enumeration of MIS

Recall that the search tree for G has at most $3^{n/3}$ leaves and the algorithm produced MIS for the leaves of the search tree.

Enumeration of MIS

Recall that the search tree for G has at most $3^{n/3}$ leaves and the algorithm produced MIS for the leaves of the search tree.

Theorem

An n-vertex graph has at most $3^{n/3}$ maximal independent sets and the bound is tight.

Enumeration of MIS

Recall that the search tree for G has at most $3^{n/3}$ leaves and the algorithm produced MIS for the leaves of the search tree.

Theorem

An n-vertex graph has at most $3^{n/3}$ maximal independent sets and the bound is tight.

Moon and Moser, 1962 proved the same by combinatorial arguments.

Enumeration of MIS

Recall that the search tree for G has at most $3^{n/3}$ leaves and the algorithm produced MIS for the leaves of the search tree.

Theorem

An n-vertex graph has at most $3^{n/3}$ maximal independent sets and the bound is tight.

Moon and Moser, 1962 proved the same by combinatorial arguments.

Let $\mathcal{A}$ be an enumeration algorithm that lists all maximal independent sets (cliques) with polynomial delay. Then $\mathcal{A}$ runs in time $O^*(3^{n/3})$.

Branching algorithms

Branching algorithms are recursively applied to (specially tailored) instances of a problem using *reduction* and *branching* rules.

Branching algorithms

Branching algorithms are recursively applied to (specially tailored) instances of a problem using *reduction* and *branching* rules.

- *Reduction rules*

Branching algorithms

Branching algorithms are recursively applied to (specially tailored) instances of a problem using *reduction* and *branching* rules.

- *Reduction rules*
  - used to simplify instances,

Branching algorithms

Branching algorithms are recursively applied to (specially tailored) instances of a problem using *reduction* and *branching* rules.

- *Reduction rules*
  - used to simplify instances,
  - typically reduce the size,

Branching algorithms

Branching algorithms are recursively applied to (specially tailored) instances of a problem using *reduction* and *branching* rules.

- *Reduction rules*
    - used to simplify instances,
    - typically reduce the size,
    - typically run in polynomial time.

Branching algorithms

Branching algorithms are recursively applied to (specially tailored) instances of a problem using *reduction* and *branching* rules.

- *Reduction rules*
  - used to simplify instances,
  - typically reduce the size,
  - typically run in polynomial time.
- *Branching rules*
  - solve the problem for an instance by recursively solving $t \geq 2$ (smaller) instances,

Branching algorithms

Branching algorithms are recursively applied to (specially tailored) instances of a problem using *reduction* and *branching* rules.

- *Reduction rules*
  - used to simplify instances,
  - typically reduce the size,
  - typically run in polynomial time.
- *Branching rules*
  - solve the problem for an instance by recursively solving $t \geq 2$ (smaller) instances,
  - typically run in polynomial time (without recursive calls).

Search trees

Search trees are used to illustrate, understand and analyze branching algorithms:

Search trees

Search trees are used to illustrate, understand and analyze branching algorithms:

- *Root*: assign the input to the root.

Search trees

Search trees are used to illustrate, understand and analyze branching algorithms:

- *Root*: assign the input to the root.
- *Node*: assign to each node a problem instance.

Search trees

Search trees are used to illustrate, understand and analyze branching algorithms:

- *Root*: assign the input to the root.
- *Node*: assign to each node a problem instance.
- *Child*: each instance produced by a branching rule is assigned to a child.

Search trees

Search trees are used to illustrate, understand and analyze branching algorithms:

- *Root*: assign the input to the root.
- *Node*: assign to each node a problem instance.
- *Child*: each instance produced by a branching rule is assigned to a child.
- *Leaf*: outputs are assigned to leaves.

Analysis of branching algorithms

- *Correctness*: show correctness of the reduction and branching rules:

Analysis of branching algorithms

- *Correctness*: show correctness of the reduction and branching rules:
  - typically, the proof is done by induction on some *measure* of an instance.
- *Running time analysis*:

Analysis of branching algorithms

- *Correctness*: show correctness of the reduction and branching rules:
    - typically, the proof is done by induction on some *measure* of an instance.
- *Running time analysis*:
    - upper bound of the maximum number of leaves $L(s)$ of a search tree for an input of given size s,

Analysis of branching algorithms

- *Correctness*: show correctness of the reduction and branching rules:
  - typically, the proof is done by induction on some *measure* of an instance.
- *Running time analysis*:
  - upper bound of the maximum number of leaves $L(s)$ of a search tree for an input of given size s,
  - if each reduction and branching rule can be done in polynomial time, then we have running time $O^*(L(s))$,

Analysis of branching algorithms

- *Correctness*: show correctness of the reduction and branching rules:
  - typically, the proof is done by induction on some *measure* of an instance.
- *Running time analysis*:
  - upper bound of the maximum number of leaves $L(s)$ of a search tree for an input of given size s,
  - if each reduction and branching rule can be done in polynomial time, then we have running time $O^*(L(s))$,
  - $L(s)$ gives a combinatorial upper bound for the number of enumerating objects.

Analysis of branching algorithms

- *Correctness*: show correctness of the reduction and branching rules:
  - typically, the proof is done by induction on some *measure* of an instance.
- *Running time analysis*:
  - upper bound of the maximum number of leaves $L(s)$ of a search tree for an input of given size s,
  - if each reduction and branching rule can be done in polynomial time, then we have running time $O^*(L(s))$,
  - $L(s)$ gives a combinatorial upper bound for the number of enumerating objects.
- *Constructing lower bounds*: family of instances with a specific lower bound on the number of enumerating objects.

Bounding the number of leaves

Let $\mathcal{A}$ be a recursive branching algorithm that uses a single branching rule that generates $r \geq 2$ instances of the problem.

Bounding the number of leaves

Let $\mathcal{A}$ be a recursive branching algorithm that uses a single branching rule that generates $r \geq 2$ instances of the problem.

We associate a **measure** $\mu(I)$ with each instance I of the problem.

Bounding the number of leaves

Let $\mathcal{A}$ be a recursive branching algorithm that uses a single branching rule that generates $r \geq 2$ instances of the problem.

We associate a **measure** $\mu(I)$ with each instance I of the problem.

Typically, for simple branching algorithms, $\mu(I)$ is integer, and often $\mu(I)$ is the number of vertices for graph problems.

Bounding the number of leaves

Let $\mathcal{A}$ be a recursive branching algorithm that uses a single branching rule that generates $r \geq 2$ instances of the problem.

We associate a **measure** $\mu(I)$ with each instance I of the problem.

Typically, for simple branching algorithms, $\mu(I)$ is integer, and often $\mu(I)$ is the number of vertices for graph problems.

Let $I_1, \ldots, I_r$ be the instances of the problem generated by the branching rule from I.

Bounding the number of leaves

Let $\mathcal{A}$ be a recursive branching algorithm that uses a single branching rule that generates $r \geq 2$ instances of the problem.

We associate a **measure** $\mu(I)$ with each instance I of the problem.

Typically, for simple branching algorithms, $\mu(I)$ is integer, and often $\mu(I)$ is the number of vertices for graph problems.

Let $I_1, \ldots, I_r$ be the instances of the problem generated by the branching rule from I.

Assume that there are positive (integers) $t_1, \ldots, t_r$ such that

$$\mu(I_i) \leq \mu(I) - t_i \text{ for } i \in \{1, \ldots, r\}.$$

Bounding the number of leaves

Let $\mathcal{A}$ be a recursive branching algorithm that uses a single branching rule that generates $r \geq 2$ instances of the problem.

We associate a **measure** $\mu(I)$ with each instance I of the problem.

Typically, for simple branching algorithms, $\mu(I)$ is integer, and often $\mu(I)$ is the number of vertices for graph problems.

Let $I_1, \ldots, I_r$ be the instances of the problem generated by the branching rule from I.

Assume that there are positive (integers) $t_1, \ldots, t_r$ such that

$$\mu(I_i) \leq \mu(I) - t_i \text{ for } i \in \{1, \ldots, r\}.$$

It is said that

$$b = (t_1, \ldots, t_r)$$

is the *branching vector* for the rule.

Bounding the number of leaves

Let $L(s)$ be the maximum number of leaves of a search tree for the instances I with $\mu(I) = s$.

Bounding the number of leaves

Let $L(s)$ be the maximum number of leaves of a search tree for the instances I with $\mu(I) = s$.

We have that

$$L(s) \leq L(s - t_1) + \ldots + L(s - t_r).$$

Bounding the number of leaves

Let $L(s)$ be the maximum number of leaves of a search tree for the instances I with $\mu(I) = s$.

We have that

$$L(s) \leq L(s - t_1) + \ldots + L(s - t_r).$$

Let $t = \max\{t_1, \ldots, t_r\}$.

Bounding the number of leaves

Let $L(s)$ be the maximum number of leaves of a search tree for the instances I with $\mu(I) = s$.

We have that

$$L(s) \leq L(s - t_1) + \ldots + L(s - t_r).$$

Let $t = \max\{t_1, \ldots, t_r\}$. We associate with $b = (t_1, \ldots, t_r)$ the *characteristic polynomial*:

$$p(x) = x^t - x^{t-t_1} - \ldots - x^{t-t_r}.$$

Bounding the number of leaves

Let $L(s)$ be the maximum number of leaves of a search tree for the instances I with $\mu(I) = s$.

We have that

$$L(s) \leq L(s - t_1) + \ldots + L(s - t_r).$$

Let $t = \max\{t_1, \ldots, t_r\}$. We associate with $b = (t_1, \ldots, t_r)$ the *characteristic polynomial*:

$$p(x) = x^t - x^{t-t_1} - \ldots - x^{t-t_r}.$$

Claim: $p(x)$ has a unique positive real root λ and $L(s) = O^*(\lambda^s)$.

Bounding the number of leaves

Let $L(s)$ be the maximum number of leaves of a search tree for the instances I with $\mu(I) = s$.

We have that

$$L(s) \leq L(s - t_1) + \ldots + L(s - t_r).$$

Let $t = \max\{t_1, \ldots, t_r\}$. We associate with $b = (t_1, \ldots, t_r)$ the *characteristic polynomial*:

$$p(x) = x^t - x^{t-t_1} - \ldots - x^{t-t_r}.$$

Claim: $p(x)$ has a unique positive real root λ and $L(s) = O^*(\lambda^s)$.

It is said that $\lambda = \lambda(t_1, \ldots, t_r)$ is the *branching number* of b.

Bounding the number of leaves

Given a *branching vector*

$$b = (t_1, \ldots, t_r),$$

we solve the equation

$$x^t - x^{t-t_1} - \ldots - x^{t-t_r} = 0$$

for $t = \max\{t_1, \ldots, t_r\}$ and find the *branching number*

$$\lambda = \lambda(t_1, \ldots, t_r).$$

Then we obtain the bound

$$L(s) = O^*(\lambda^s).$$

Enumeration of vertex subsets

Observe that to obtain the bound $L(s) = O^*(\lambda^s)$, we have not made any assumption about the considered enumeration problem or the measure $\mu(I)$.

Enumeration of vertex subsets

Observe that to obtain the bound $L(s) = O^*(\lambda^s)$, we have not made any assumption about the considered enumeration problem or the measure $\mu(I)$.

Assume that we consider an enumeration problem for graphs where the aim is to list all vertex subsets that satisfy a property P.

Enumeration of vertex subsets

Observe that to obtain the bound $L(s) = O^*(\lambda^s)$, we have not made any assumption about the considered enumeration problem or the measure $\mu(I)$.

Assume that we consider an enumeration problem for graphs where the aim is to list all vertex subsets that satisfy a property P.

Claim: If $\mu(I) \leq n$ for the inputs containing n-vertex graphs, then

$$L(n) = O^*(\lambda^n).$$

Obtaining better bounds

By general results we can obtain the bound of form $O^*(\lambda^s)$.

Obtaining better bounds

By general results we can obtain the bound of form $O^*(\lambda^s)$.

To obtain a better bound, we can

Obtaining better bounds

By general results we can obtain the bound of form $O^*(\lambda^s)$.

To obtain a better bound, we can

- round $\lambda < \hat{\lambda}$ and get the bound $O(\hat{\lambda}^s)$,

Obtaining better bounds

By general results we can obtain the bound of form $O^*(\lambda^s)$.

To obtain a better bound, we can

- round $\lambda < \hat{\lambda}$ and get the bound $O(\hat{\lambda}^s)$,
- analyze the first $t = \max\{t_1, \ldots, t_r\}$ values of $L(s)$ get the bound λ^s.

Obtaining better bounds

By general results we can obtain the bound of form $O^*(\lambda^s)$.

To obtain a better bound, we can

- round $\lambda < \hat{\lambda}$ and get the bound $O(\hat{\lambda}^s)$,
- analyze the first $t = \max\{t_1, \ldots, t_r\}$ values of $L(s)$ get the bound λ^s.

Let $b = (t_1, \ldots, t_r)$ and $\lambda = \lambda(t_1, \ldots, t_r)$.

Obtaining better bounds

By general results we can obtain the bound of form $O^*(\lambda^s)$.

To obtain a better bound, we can

- round $\lambda < \hat{\lambda}$ and get the bound $O(\hat{\lambda}^s)$,
- analyze the first $t = \max\{t_1, \ldots, t_r\}$ values of $L(s)$ get the bound λ^s.

Let $b = (t_1, \ldots, t_r)$ and $\lambda = \lambda(t_1, \ldots, t_r)$.

Suppose that $L \colon \mathbb{Z} \to \mathbb{Z}_{\geq 0}$ such that

Obtaining better bounds

By general results we can obtain the bound of form $O^*(\lambda^s)$.

To obtain a better bound, we can

- round $\lambda < \hat{\lambda}$ and get the bound $O(\hat{\lambda}^s)$,
- analyze the first $t = \max\{t_1, \dots, t_r\}$ values of $L(s)$ get the bound λ^s.

Let $b = (t_1, \dots, t_r)$ and $\lambda = \lambda(t_1, \dots, t_r)$.

Suppose that $L\colon \mathbb{Z} \to \mathbb{Z}_{\geq 0}$ such that

(i) $L(s) = 0$ for $s < 0$,

Obtaining better bounds

By general results we can obtain the bound of form $O^*(\lambda^s)$.

To obtain a better bound, we can

- round $\lambda < \hat{\lambda}$ and get the bound $O(\hat{\lambda}^s)$,
- analyze the first $t = \max\{t_1, \ldots, t_r\}$ values of $L(s)$ get the bound λ^s.

Let $b = (t_1, \ldots, t_r)$ and $\lambda = \lambda(t_1, \ldots, t_r)$.

Suppose that $L\colon \mathbb{Z} \to \mathbb{Z}_{\geq 0}$ such that

(i) $L(s) = 0$ for $s < 0$,

(ii) $L(s) \leq \max\{1, L(s - t_1) + \ldots + L(s - t_r)\}$ for $s \geq 0$.

Obtaining better bounds

By general results we can obtain the bound of form $O^*(\lambda^s)$.

To obtain a better bound, we can

- round $\lambda < \hat{\lambda}$ and get the bound $O(\hat{\lambda}^s)$,
- analyze the first $t = \max\{t_1, \dots, t_r\}$ values of $L(s)$ get the bound λ^s.

Let $b = (t_1, \dots, t_r)$ and $\lambda = \lambda(t_1, \dots, t_r)$.

Suppose that $L\colon \mathbb{Z} \to \mathbb{Z}_{\geq 0}$ such that

(i) $L(s) = 0$ for $s < 0$,

(ii) $L(s) \leq \max\{1, L(s - t_1) + \dots + L(s - t_r)\}$ for $s \geq 0$.

Exercise: Show that $L(s) \leq \lambda^s$.

Analyzing collections of recurrences

A branching algorithm can have several branching rules or/and the number of instances produced by a rule can depend on the considered instance.

Analyzing collections of recurrences

A branching algorithm can have several branching rules or/and the number of instances produced by a rule can depend on the considered instance.

For example, **Enum MIS**(G, S) generates $d + 1$ branches where $d = d_G(v)$.

Analyzing collections of recurrences

A branching algorithm can have several branching rules or/and the number of instances produced by a rule can depend on the considered instance.

For example, **Enum MIS**(G, S) generates $d+1$ branches where $d = d_G(v)$.

We consider all branching vectors and find the worst branching number.

Analyzing collections of recurrences

A branching algorithm can have several branching rules or/and the number of instances produced by a rule can depend on the considered instance.

For example, **Enum MIS**(G, S) generates $d+1$ branches where $d = d_G(v)$.

We consider all branching vectors and find the worst branching number.

For **Enum MIS**(G, S), we have

$$b_{d+1} = (\underbrace{d+1, \ldots, d+1}_{d+1})$$

Analyzing collections of recurrences

A branching algorithm can have several branching rules or/and the number of instances produced by a rule can depend on the considered instance.

For example, **Enum MIS**(G, S) generates $d + 1$ branches where $d = d_G(v)$.

We consider all branching vectors and find the worst branching number.

For **Enum MIS**(G, S), we have

$$b_{d+1} = (\underbrace{d + 1, \ldots, d + 1}_{d+1}) \text{ and } \lambda_{d+1} = (d + 1)^{1/(d+1)}.$$

Then

$$\lambda = \max_{d \geq 0} \lambda_{d+1} = 3^{1/3}.$$

Analyzing collections of recurrences

We consider all branching rules and construct the family of branching vectors

$$\mathcal{B} = \{b^{(i)} \mid i \in \mathbb{N}\}$$

Analyzing collections of recurrences

We consider all branching rules and construct the family of branching vectors

$$\mathcal{B} = \{b^{(i)} \mid i \in \mathbb{N}\}$$

$$b^{(i)} = (t_1^{(i)}, \ldots, t_{r(i)}^{(i)}).$$

Analyzing collections of recurrences

We consider all branching rules and construct the family of branching vectors

$$\mathcal{B} = \{b^{(i)} \mid i \in \mathbb{N}\}$$

$$b^{(i)} = (t_1^{(i)}, \ldots, t_{r(i)}^{(i)}).$$

We compute

$$\lambda = \sup\{\lambda(b^{(i)}) \mid b^{(i)} \in \mathcal{B}\}.$$

Analyzing collections of recurrences

We consider all branching rules and construct the family of branching vectors

$$\mathcal{B} = \{b^{(i)} \mid i \in \mathbb{N}\}$$

$$b^{(i)} = (t_1^{(i)}, \ldots, t_{r(i)}^{(i)}).$$

We compute

$$\lambda = \sup\{\lambda(b^{(i)}) \mid b^{(i)} \in \mathcal{B}\}.$$

Claim:

$$L(s) = O^*(\lambda^s).$$

Analyzing collections of recurrences

We consider all branching rules and construct the family of branching vectors

$$\mathcal{B} = \{b^{(i)} \mid i \in \mathbb{N}\}$$

$$b^{(i)} = (t_1^{(i)}, \ldots, t_{r(i)}^{(i)}).$$

We compute

$$\lambda = \sup\{\lambda(b^{(i)}) \mid b^{(i)} \in \mathcal{B}\}.$$

Claim:

$$L(s) = O^*(\lambda^s).$$

Often it could be shown that $L(s) \leq \lambda^s$.

Analyzing collections of recurrences

The family of branching vectors $\mathcal{B}$ could be infinite.

Analyzing collections of recurrences

The family of branching vectors $\mathcal{B}$ could be infinite.

Nevertheless, it is usually sufficient to consider few "worst" branching vectors and find

$$\lambda = \max\{\lambda(b^{(i)}) \mid b^{(i)} \in \mathcal{B}\}.$$

Properties of branching vectors

Let $b = (t_1, \ldots, t_r)$, $r \geq 2$ and $t_i > 0$ for $i \in \{1, \ldots, r\}$.

Properties of branching vectors

Let $b = (t_1, \ldots, t_r)$, $r \geq 2$ and $t_i > 0$ for $i \in \{1, \ldots, r\}$.

- $\lambda(t_1, \ldots, t_r) > 1$.

Properties of branching vectors

Let $b = (t_1, \ldots, t_r)$, $r \geq 2$ and $t_i > 0$ for $i \in \{1, \ldots, r\}$.

- $\lambda(t_1, \ldots, t_r) > 1$.
- For every permutation π of $\langle 1, \ldots, r \rangle$, $\lambda(t_1, \ldots, t_r) = \lambda(t_{\pi(1)}, \ldots, t_{\pi(r)})$.

Properties of branching vectors

Let $b = (t_1, \ldots, t_r)$, $r \geq 2$ and $t_i > 0$ for $i \in \{1, \ldots, r\}$.

- $\lambda(t_1, \ldots, t_r) > 1$.
- For every permutation π of $\langle 1, \ldots, r \rangle$, $\lambda(t_1, \ldots, t_r) = \lambda(t_{\pi(1)}, \ldots, t_{\pi(r)})$.
- If $b' = (t'_1, \ldots, t'_r)$ and $t_i \leq t'_i$ for $i \in \{1, \ldots, r\}$, then $\lambda(t_1, \ldots, t_r) \geq \lambda(t'_1, \ldots, t'_r)$.

Properties of branching vectors

Let $b = (t_1, \ldots, t_r)$, $r \geq 2$ and $t_i > 0$ for $i \in \{1, \ldots, r\}$.

- $\lambda(t_1, \ldots, t_r) > 1$.
- For every permutation π of $\langle 1, \ldots, r \rangle$, $\lambda(t_1, \ldots, t_r) = \lambda(t_{\pi(1)}, \ldots, t_{\pi(r)})$.
- If $b' = (t'_1, \ldots, t'_r)$ and $t_i \leq t'_i$ for $i \in \{1, \ldots, r\}$, then $\lambda(t_1, \ldots, t_r) \geq \lambda(t'_1, \ldots, t'_r)$.

Let $a, b, c > 0$. Then

Properties of branching vectors

Let $b = (t_1, \ldots, t_r)$, $r \geq 2$ and $t_i > 0$ for $i \in \{1, \ldots, r\}$.

- $\lambda(t_1, \ldots, t_r) > 1$.
- For every permutation π of $\langle 1, \ldots, r \rangle$, $\lambda(t_1, \ldots, t_r) = \lambda(t_{\pi(1)}, \ldots, t_{\pi(r)})$.
- If $b' = (t'_1, \ldots, t'_r)$ and $t_i \leq t'_i$ for $i \in \{1, \ldots, r\}$, then $\lambda(t_1, \ldots, t_r) \geq \lambda(t'_1, \ldots, t'_r)$.

Let $a, b, c > 0$. Then

- $\lambda(c, c) \leq \lambda(a, b)$ if $c = (a + b)/2$,

Properties of branching vectors

Let $b = (t_1, \ldots, t_r)$, $r \geq 2$ and $t_i > 0$ for $i \in \{1, \ldots, r\}$.

- $\lambda(t_1, \ldots, t_r) > 1$.
- For every permutation π of $\langle 1, \ldots, r \rangle$, $\lambda(t_1, \ldots, t_r) = \lambda(t_{\pi(1)}, \ldots, t_{\pi(r)})$.
- If $b' = (t'_1, \ldots, t'_r)$ and $t_i \leq t'_i$ for $i \in \{1, \ldots, r\}$, then $\lambda(t_1, \ldots, t_r) \geq \lambda(t'_1, \ldots, t'_r)$.

Let $a, b, c > 0$. Then

- $\lambda(c, c) \leq \lambda(a, b)$ if $c = (a + b)/2$,
- $\lambda(a + \varepsilon, b - \varepsilon) < \lambda(a, b)$ for $0 < a < b$ and $0 < \varepsilon < (b - a)/2$.

Some branching numbers

Branching numbers for $b = (t_1, t_2)$:

	1	2	3	4	5	6
1	2.0000	1.6181	1.4656	1.3803	1.3248	1.2852
2	1.6181	1.4143	1.3248	1.2721	1.2366	1.2107
3	1.4656	1.3248	1.2560	1.2208	1.1939	1.1740
4	1.3803	1.2721	1.2208	1.1893	1.1674	1.1510
5	1.3248	1.2366	1.1939	1.1674	1.1487	1.1348
6	1.2852	1.2107	1.1740	1.1510	1.1348	1.1225

Improving branching

Typical difficulty with the analysis of branching algorithms is that we get a “bad” branching rule that cannot be avoided.

Improving branching

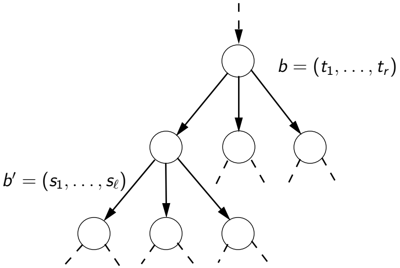
Typical difficulty with the analysis of branching algorithms is that we get a "bad" branching rule that cannot be avoided.

Sometimes it is possible to combine a "bad" branching with a consecutive "good" one.

Improving branching

Typical difficulty with the analysis of branching algorithms is that we get a “bad” branching rule that cannot be avoided.

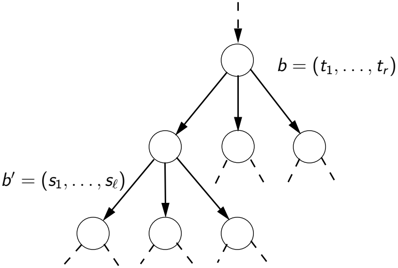
Sometimes it is possible to combine a “bad” branching with a consecutive “good” one.



Improving branching

Typical difficulty with the analysis of branching algorithms is that we get a "bad" branching rule that cannot be avoided.

Sometimes it is possible to combine a "bad" branching with a consecutive "good" one.



New branching vector: $c = (t_1 + s_1, \ldots, t_1 + s_\ell, t_2, \ldots, t_r)$.

Enumeration of minimal hitting sets

Let $\mathcal{S}$ be a family of subsets over an universe $\mathcal{U}$.

Enumeration of minimal hitting sets

Let $\mathcal{S}$ be a family of subsets over an universe $\mathcal{U}$.

$X \subseteq \mathcal{U}$ is a hitting set for $\mathcal{S}$ if for every $S \in \mathcal{S}$, $X \cap S \neq \emptyset$.

Enumeration of minimal hitting sets

Let $\mathcal{S}$ be a family of subsets over an universe $\mathcal{U}$.

$X \subseteq \mathcal{U}$ is a hitting set for $\mathcal{S}$ if for every $S \in \mathcal{S}$, $X \cap S \neq \emptyset$.

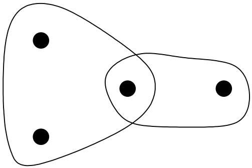
A hitting set X is (inclusion) minimal if every $Y \subset X$ is not a hitting set.

Enumeration of minimal hitting sets

Let $\mathcal{S}$ be a family of subsets over an universe $\mathcal{U}$.

$X \subseteq \mathcal{U}$ is a hitting set for $\mathcal{S}$ if for every $S \in \mathcal{S}$, $X \cap S \neq \emptyset$.

A hitting set X is (inclusion) minimal if every $Y \subset X$ is not a hitting set.

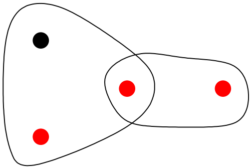


Enumeration of minimal hitting sets

Let $\mathcal{S}$ be a family of subsets over an universe $\mathcal{U}$.

$X \subseteq \mathcal{U}$ is a hitting set for $\mathcal{S}$ if for every $S \in \mathcal{S}$, $X \cap S \neq \emptyset$.

A hitting set X is (inclusion) minimal if every $Y \subset X$ is not a hitting set.

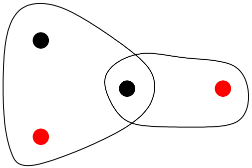


Enumeration of minimal hitting sets

Let $\mathcal{S}$ be a family of subsets over an universe $\mathcal{U}$.

$X \subseteq \mathcal{U}$ is a hitting set for $\mathcal{S}$ if for every $S \in \mathcal{S}$, $X \cap S \neq \emptyset$.

A hitting set X is (inclusion) minimal if every $Y \subset X$ is not a hitting set.



Enumeration of minimal hitting sets

Problem (Minimal Hitting Set Enumeration)

Input: *A family $\mathcal{S}$ of subsets over an universe $\mathcal{U}$.*

Task: *Enumerate all maximal independent sets of G.*

Enumeration of minimal hitting sets

Problem (Minimal Hitting Set Enumeration)

Input: *A family $\mathcal{S}$ of subsets over an universe $\mathcal{U}$.*

Task: *Enumerate all maximal independent sets of G.*

We consider the problem for $\mathcal{S}$ containing subsets of size at most 3.

Enumeration of minimal hitting sets

We construct the algorithm **Emum Hitting Sets**$(\mathcal{F}, X)$.

Enumeration of minimal hitting sets

We construct the algorithm **Emum Hitting Sets**$(\mathcal{F}, X)$.

Input: A subfamily $\mathcal{F} \subseteq \mathcal{S}$ and $X \subseteq \mathcal{U}$ such that $X \cap F = \emptyset$ for $F \in \mathcal{F}$.

Enumeration of minimal hitting sets

We construct the algorithm **Emum Hitting Sets**$(\mathcal{F}, X)$.

Input: A subfamily $\mathcal{F} \subseteq \mathcal{S}$ and $X \subseteq \mathcal{U}$ such that $X \cap F = \emptyset$ for $F \in \mathcal{F}$.

Output: Minimal hitting sets of $\mathcal{S}$ of form $X \cup Y$ where Y is a minimal hitting set for $\mathcal{F}$.

Enumeration of minimal hitting sets

We construct the algorithm **Emum Hitting Sets**$(\mathcal{F}, X)$.

Input: A subfamily $\mathcal{F} \subseteq \mathcal{S}$ and $X \subseteq \mathcal{U}$ such that $X \cap F = \emptyset$ for $F \in \mathcal{F}$.

Output: Minimal hitting sets of $\mathcal{S}$ of form $X \cup Y$ where Y is a minimal hitting set for $\mathcal{F}$.

We generate all minimal hitting sets Y of $\mathcal{F}$, and for each Y, we test whether $X \cup Y$ is minimal hitting set for $\mathcal{S}$.

Enumeration of minimal hitting sets

We construct the algorithm **Emum Hitting Sets**$(\mathcal{F}, X)$.

Input: A subfamily $\mathcal{F} \subseteq \mathcal{S}$ and $X \subseteq \mathcal{U}$ such that $X \cap F = \emptyset$ for $F \in \mathcal{F}$.

Output: Minimal hitting sets of $\mathcal{S}$ of form $X \cup Y$ where Y is a minimal hitting set for $\mathcal{F}$.

We generate all minimal hitting sets Y of $\mathcal{F}$, and for each Y, we test whether $X \cup Y$ is minimal hitting set for $\mathcal{S}$.

To enumerate minimal hitting sets of $\mathcal{S}$, we call **Emum Hitting Sets**$(\mathcal{S}, \emptyset)$.

Enumeration of minimal hitting sets

We construct the algorithm **Emum Hitting Sets**$(\mathcal{F}, X)$.

Input: A subfamily $\mathcal{F} \subseteq \mathcal{S}$ and $X \subseteq \mathcal{U}$ such that $X \cap F = \emptyset$ for $F \in \mathcal{F}$.

Output: Minimal hitting sets of $\mathcal{S}$ of form $X \cup Y$ where Y is a minimal hitting set for $\mathcal{F}$.

We generate all minimal hitting sets Y of $\mathcal{F}$, and for each Y, we test whether $X \cup Y$ is minimal hitting set for $\mathcal{S}$.

To enumerate minimal hitting sets of $\mathcal{S}$, we call **Emum Hitting Sets**$(\mathcal{S}, \emptyset)$.

The measure of the instance is the size of

$$\mathcal{W} = \cup\mathcal{F} = \cup_{S \in \mathcal{F}} S.$$

Reduction rules

- If $\mathcal{F} = \emptyset$, then check whether X is a minimal hitting set for $\mathcal{S}$ and output X if it holds.

Reduction rules

- If $\mathcal{F} = \emptyset$, then check whether X is a minimal hitting set for $\mathcal{S}$ and output X if it holds.
- If $\emptyset \in \mathcal{F}$, then **Stop**.

Reduction rules

- If $\mathcal{F} = \emptyset$, then check whether X is a minimal hitting set for $\mathcal{S}$ and output X if it holds.
- If $\emptyset \in \mathcal{F}$, then **Stop**.
- If there is $S \in \mathcal{F}$ such that $|S| = 1$, then for $\{x\} = S$ do the following:

Reduction rules

- If $\mathcal{F} = \emptyset$, then check whether X is a minimal hitting set for $\mathcal{S}$ and output X if it holds.
- If $\emptyset \in \mathcal{F}$, then **Stop**.
- If there is $S \in \mathcal{F}$ such that $|S| = 1$, then for $\{x\} = S$ do the following:
  - Set $\mathcal{F}' = \{S \in \mathcal{F} \mid x \notin S\}$,
  - Call **Emum Hitting Sets**$(\mathcal{F}', X \cup \{x\})$.

Branching rule

Select $S \in \mathcal{F}$.

Branching rule

Select $S \in \mathcal{F}$.

- If $|S| = 2$, then for $\{x, y\} = S$ do the following:

Branching rule

Select $S \in \mathcal{F}$.

- If $|S| = 2$, then for $\{x, y\} = S$ do the following:
  (i) Set $\mathcal{F}' = \{S \in \mathcal{F} \mid x \notin S\}$ and call **Emum Hitting Sets**$(\mathcal{F}', X \cup \{x\})$.
  (ii) Set $\mathcal{F}'' = \{S \setminus \{x\} \mid S \in \mathcal{F} \text{ s.t. } y \notin S\}$ and call **Emum Hitting Sets**$(\mathcal{F}'', X \cup \{y\})$.

Branching rule

Select $S \in \mathcal{F}$.

- If $|S| = 2$, then for $\{x, y\} = S$ do the following:
  (i) Set $\mathcal{F}' = \{S \in \mathcal{F} \mid x \notin S\}$ and call **Emum Hitting Sets**$(\mathcal{F}', X \cup \{x\})$.
  (ii) Set $\mathcal{F}'' = \{S \setminus \{x\} \mid S \in \mathcal{F} \text{ s.t. } y \notin S\}$ and call **Emum Hitting Sets**$(\mathcal{F}'', X \cup \{y\})$.
- If $|S| = 3$, then for $\{x, y, z\} = S$ do the following:

Branching rule

Select $S \in \mathcal{F}$.

- If $|S| = 2$, then for $\{x, y\} = S$ do the following:
  - **(i)** Set $\mathcal{F}' = \{S \in \mathcal{F} \mid x \notin S\}$ and call **Emum Hitting Sets**$(\mathcal{F}', X \cup \{x\})$.
  - **(ii)** Set $\mathcal{F}'' = \{S \setminus \{x\} \mid S \in \mathcal{F} \text{ s.t. } y \notin S\}$ and call **Emum Hitting Sets**$(\mathcal{F}'', X \cup \{y\})$.
- If $|S| = 3$, then for $\{x, y, z\} = S$ do the following:
  - **(i)** Set $\mathcal{F}' = \{S \in \mathcal{F} \mid x \notin S\}$ and call **Emum Hitting Sets**$(\mathcal{F}', X \cup \{x\})$.
  - **(ii)** Set $\mathcal{F}'' = \{S \setminus \{x\} \mid S \in \mathcal{F} \text{ s.t. } y \notin S\}$ and call **Emum Hitting Sets**$(\mathcal{F}'', X \cup \{y\})$.
  - **(iii)** Set $\mathcal{F}''' = \{S \setminus \{x, y\} \mid S \in \mathcal{F} \text{ s.t. } z \notin S\}$ and call **Emum Hitting Sets**$(\mathcal{F}''', X \cup \{z\})$.

Branching vectors and numbers

- If $|S| = 2$, then

Branching vectors and numbers

- If $|S| = 2$, then
  - the branching vector is $(1, 2)$,

Branching vectors and numbers

- If $|S| = 2$, then
  - the branching vector is $(1, 2)$,
  - the characteristic polynomial is $x^2 - x - 1 = 0$,

Branching vectors and numbers

- If $|S| = 2$, then
  - the branching vector is $(1, 2)$,
  - the characteristic polynomial is $x^2 - x - 1 = 0$,
  - $\lambda(2, 1) \approx 1.6181$.
- If $|S| = 3$, then

Branching vectors and numbers

- If $|S| = 2$, then
  - the branching vector is $(1, 2)$,
  - the characteristic polynomial is $x^2 - x - 1 = 0$,
  - $\lambda(2, 1) \approx 1.6181$.
- If $|S| = 3$, then
  - the branching vector is $(1, 2, 3)$,

Branching vectors and numbers

- If $|S| = 2$, then
    - the branching vector is $(1, 2)$,
    - the characteristic polynomial is $x^2 - x - 1 = 0$,
    - $\lambda(2, 1) \approx 1.6181$.
- If $|S| = 3$, then
    - the branching vector is $(1, 2, 3)$,
    - the characteristic polynomial is $x^3 - x^2 - x - 1 = 0$,

Branching vectors and numbers

- If $|S| = 2$, then
  - the branching vector is $(1, 2)$,
  - the characteristic polynomial is $x^2 - x - 1 = 0$,
  - $\lambda(2, 1) \approx 1.6181$.
- If $|S| = 3$, then
  - the branching vector is $(1, 2, 3)$,
  - the characteristic polynomial is $x^3 - x^2 - x - 1 = 0$,
  - $\lambda(2, 1) \approx 1.8394$.

Branching vectors and numbers

- If $|S| = 2$, then
  - the branching vector is $(1, 2)$,
  - the characteristic polynomial is $x^2 - x - 1 = 0$,
  - $\lambda(2, 1) \approx 1.6181$.
- If $|S| = 3$, then
  - the branching vector is $(1, 2, 3)$,
  - the characteristic polynomial is $x^3 - x^2 - x - 1 = 0$,
  - $\lambda(2, 1) \approx 1.8394$.

- **Emum Hitting Sets** enumerates minimal hitting sets in time $O(1.8394^n)$ where $n = |\mathcal{U}|$.

Branching vectors and numbers

- If $|S| = 2$, then
  - the branching vector is $(1, 2)$,
  - the characteristic polynomial is $x^2 - x - 1 = 0$,
  - $\lambda(2, 1) \approx 1.6181$.
- If $|S| = 3$, then
  - the branching vector is $(1, 2, 3)$,
  - the characteristic polynomial is $x^3 - x^2 - x - 1 = 0$,
  - $\lambda(2, 1) \approx 1.8394$.

- **Emum Hitting Sets** enumerates minimal hitting sets in time $O(1.8394^n)$ where $n = |\mathcal{U}|$.
- there are at most 1.8394^n minimal hitting sets if $\mathcal{S}$ contains sets of size at most 3.

Exercises

1. Enumerate all maximal matchings in a graph.
   - Construct an algorithm that runs in time $O^*(c^m)$ for $c < 2$ where m is the number of edges.
   - Is it possible to enumerate the maximal matchings in time $O^*(c^n)$ where n is the number of vertices?

Exercises

1. Enumerate all maximal matchings in a graph.
   - Construct an algorithm that runs in time $O^*(c^m)$ for $c < 2$ where m is the number of edges.
   - Is it possible to enumerate the maximal matchings in time $O^*(c^n)$ where n is the number of vertices?
2. Let G be a graph, $s, t \in V(G)$. Enumerate all induced (s, t)-paths.
   - Construct an algorithm.
   - Give a lower bound.

Exercises

1. Enumerate all maximal matchings in a graph.
   - Construct an algorithm that runs in time $O^*(c^m)$ for $c < 2$ where m is the number of edges.
   - Is it possible to enumerate the maximal matchings in time $O^*(c^n)$ where n is the number of vertices?
2. Let G be a graph, $s, t \in V(G)$. Enumerate all induced (s, t)-paths.
   - Construct an algorithm.
   - Give a lower bound.
3. Enumerate all induced cycles in a graph.
   - Construct an algorithm.
   - Give a lower bound.

Exercises

1. Enumerate all maximal matchings in a graph.
   - Construct an algorithm that runs in time $O^*(c^m)$ for $c < 2$ where m is the number of edges.
   - Is it possible to enumerate the maximal matchings in time $O^*(c^n)$ where n is the number of vertices?
2. Let G be a graph, $s, t \in V(G)$. Enumerate all induced (s, t)-paths.
   - Construct an algorithm.
   - Give a lower bound.
3. Enumerate all induced cycles in a graph.
   - Construct an algorithm.
   - Give a lower bound.

($*$) Improve the running time $O(1.8394^n)$ for the enumeration of minimal hitting sets.