Complexity of enumeration and a practical example in cheminformatics

Yann Strozecki

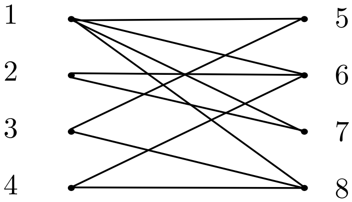
Université de Versailles St-Quentin-en-Yvelines
Laboratoire PRiSM

May 2014, Séminaire du LIMOS

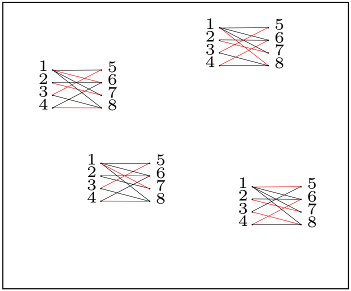
Enumeration problems

- Enumeration problems: list all solutions rather than just deciding whether there is one.

Perfect matchings:



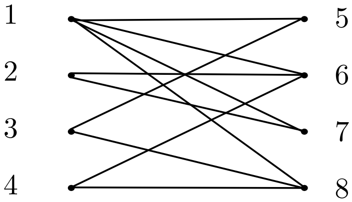
Solutions :



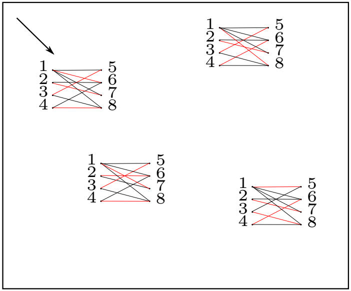
Enumeration problems

- Enumeration problems: list all solutions rather than just deciding whether there is one.

Perfect matchings:



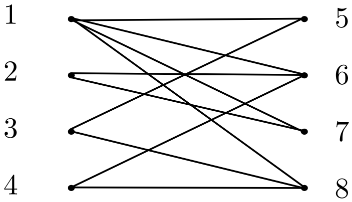
Solutions :



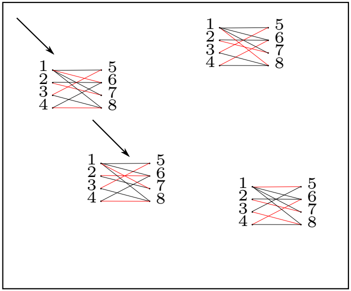
Enumeration problems

- Enumeration problems: list all solutions rather than just deciding whether there is one.

Perfect matchings:



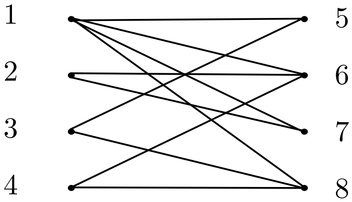
Solutions :



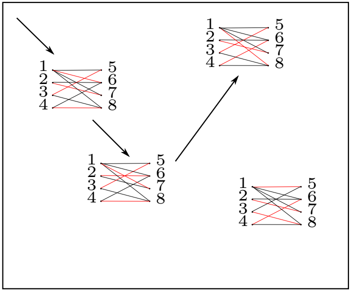
Enumeration problems

- Enumeration problems: list all solutions rather than just deciding whether there is one.

Perfect matchings:



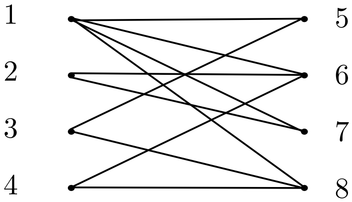
Solutions :



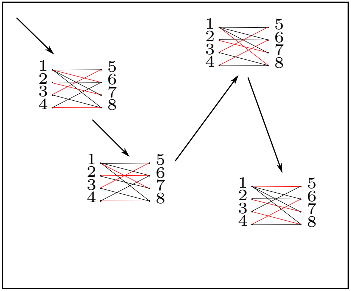
Enumeration problems

- Enumeration problems: list all solutions rather than just deciding whether there is one.
- Complexity measures: total time and delay between solutions.

Perfect matchings:



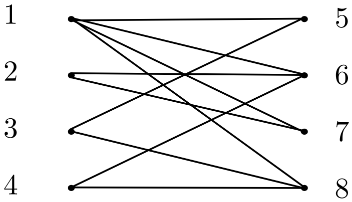
Solutions :



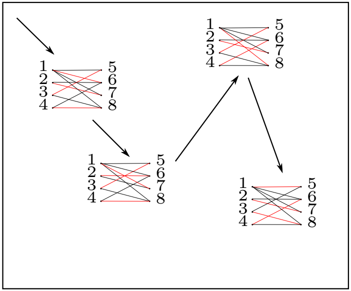
Enumeration problems

- Enumeration problems: list all solutions rather than just deciding whether there is one.
- Complexity measures: total time and delay between solutions.
- Motivations: database queries, optimization, turning an implicit representation to an explicit one.

Perfect matchings:



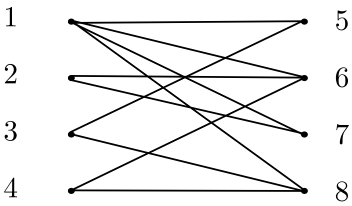
Solutions :



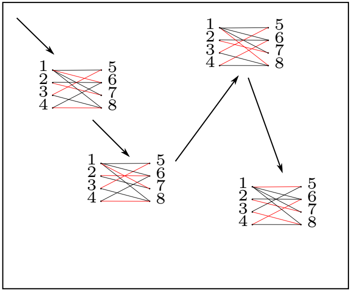
Enumeration problems

- Enumeration problems: list all solutions rather than just deciding whether there is one.
- Complexity measures: total time and delay between solutions.
- Motivations: database queries, optimization, turning an implicit representation to an explicit one.

Perfect matchings:



Solutions :



Enumeration Complexity

- Theoretical framework
- Methods for enumeration

A practical enumeration problem from cheminformatics

- Enumeration of planar maps with constraints
- Our algorithm: Kékulé

Framework

Polynomially balanced predicate $A(x, y)$, decidable in polynomial time.

- $\exists ? y A(x, y)$: decision problem (class NP)
- $\sharp\{y \mid A(x, y)\}$: counting problem (class $\sharp\mathrm{P}$)
- $\{y \mid A(x, y)\}$: enumeration problem (class $\textsc{EnumP}$)

In quite a lot of papers, an order on the enumeration is required. It often makes problems harder and is not considered here.

Framework

Polynomially balanced predicate $A(x, y)$, decidable in polynomial time.

- $\exists ? y A(x, y)$: decision problem (class NP)
- $\sharp\{y \mid A(x, y)\}$: counting problem (class $\sharp\mathrm{P}$)
- $\{y \mid A(x, y)\}$: enumeration problem (class $\textsc{EnumP}$)

In quite a lot of papers, an order on the enumeration is required. It often makes problems harder and is not considered here.

Complexity measure

1. The total time
    - polynomial total time: TOTALP (Transversal hypergraph)
    - constant amortized time: CAT (Tree enumeration)
2. The delay
    - incremental polynomial time: INCP (Circuits of a matroid)
    - polynomial delay: DELAYP (Perfect Matching)
    - Constant or linear delay
        - A two steps algorithm: preprocessing + generation
        - An ad-hoc RAM model.

Complexity measure

1. The total time
    - polynomial total time: TotalP (Transversal hypergraph)
    - constant amortized time: CAT (Tree enumeration)
2. The delay
    - incremental polynomial time: IncP (Circuits of a matroid)
    - polynomial delay: DelayP (Perfect Matching)
    - Constant or linear delay
        - A two steps algorithm: preprocessing + generation
        - An ad-hoc RAM model.
3. The space

    Polynomial in the instance or in the output ?

Complexity measure

1. The total time
    - polynomial total time: TOTALP (Transversal hypergraph)
    - constant amortized time: CAT (Tree enumeration)
2. The delay
    - incremental polynomial time: INCP (Circuits of a matroid)
    - polynomial delay: DELAYP (Perfect Matching)
    - Constant or linear delay
        - A two steps algorithm: preprocessing + generation
        - An ad-hoc RAM model.
3. The space

   Polynomial in the instance or in the output ?

Relation between classes

- $\mathrm{SDelayP}$: produce the next solution from the last one only in polynomial time
- QueryP: produce the i^{th} solution in polynomial time

Proposition

Conditional separation under $\mathsf{P} \neq \mathsf{NP}$ *hypothesis:*

$$\mathrm{QueryP} \subsetneq \mathrm{SDelayP} \subsetneq \textsc{DelayP} \subseteq \textsc{IncP} \subsetneq \textsc{TotalP} \subsetneq \textsc{EnumP}$$

Relation between classes

- $\mathrm{SDelayP}$: produce the next solution from the last one only in polynomial time
- QueryP: produce the i^{th} solution in polynomial time

Proposition

Conditional separation under P $\neq$ NP *hypothesis:*

$$\mathrm{QueryP} \subsetneq \mathrm{SDelayP} \subsetneq \textsc{DelayP} \subseteq \textsc{IncP} \subsetneq \textsc{TotalP} \subsetneq \textsc{EnumP}$$

Open question: is $\textsc{DelayP} \neq \textsc{IncP}$ modulo some complexity hypothesis ?

Relation between classes

- $\mathrm{SDelayP}$: produce the next solution from the last one only in polynomial time
- QueryP: produce the i^{th} solution in polynomial time

Proposition

Conditional separation under P $\neq$ NP *hypothesis:*

$$\mathrm{QueryP} \subsetneq \mathrm{SDelayP} \subsetneq \textsc{DelayP} \subseteq \textsc{IncP} \subsetneq \textsc{TotalP} \subsetneq \textsc{EnumP}$$

Open question: is $\textsc{DelayP} \neq \textsc{IncP}$ modulo some complexity hypothesis ?

Issue: No good notion of reduction out of parsimonious reduction and thus no complete problem except for $\textsc{EnumP}$!

Relation between classes

- $\mathrm{SDelayP}$: produce the next solution from the last one only in polynomial time
- QueryP: produce the i^{th} solution in polynomial time

Proposition

Conditional separation under $\mathsf{P} \neq \mathsf{NP}$ *hypothesis:*

$$\mathrm{QueryP} \subsetneq \mathrm{SDelayP} \subsetneq \textsc{DelayP} \subseteq \textsc{IncP} \subsetneq \textsc{TotalP} \subsetneq \textsc{EnumP}$$

Open question: is $\textsc{DelayP} \neq \textsc{IncP}$ modulo some complexity hypothesis ?

Issue: No good notion of reduction out of parsimonious reduction and thus no complete problem except for $\textsc{EnumP}$!

Randomized enumeration algorithm

Two ways of introducing randomness:

1. We can allow some randomization in our algorithms. They must satify that all solutions are enumerated correctly with probability $1 - \epsilon$.
2. We may want to solve the Uniform generation problem: generate (almost) uniformly at random a solution of a problem.

Randomized enumeration algorithm

Two ways of introducing randomness:

1. We can allow some randomization in our algorithms. They must satify that all solutions are enumerated correctly with probability $1 - \epsilon$.

2. We may want to solve the Uniform generation problem: generate (almost) uniformly at random a solution of a problem.

Uniform generation + space implies efficient enumeration

Theorem

If A has a polytime almost uniform random generator, then $\textsc{Enum}{\cdot}A \in \textsc{Inc}\mathrm{PP}$.

Idea:

- generate solutions randomly

Uniform generation + space implies efficient enumeration

Theorem

If A has a polytime almost uniform random generator, then $\textsc{Enum}\cdot A \in \textsc{Inc}\mathrm{PP}$.

Idea:

- generate solutions randomly
- put them in an efficient data structure (to avoid repetitions)

Uniform generation + space implies efficient enumeration

Theorem

If A has a polytime almost uniform random generator, then $\textsc{Enum}{\cdot}A \in \textsc{Inc}\mathrm{PP}$.

Idea:

- generate solutions randomly
- put them in an efficient data structure (to avoid repetitions)
- coupon collector problem

Uniform generation + space implies efficient enumeration

Theorem

If A has a polytime almost uniform random generator, then $\textsc{Enum}\cdot A \in \textsc{IncPP}$.

Idea:

- generate solutions randomly
- put them in an efficient data structure (to avoid repetitions)
- coupon collector problem
- amortize the output of solutions

Uniform generation + space implies efficient enumeration

Theorem

If A has a polytime almost uniform random generator, then $\textsc{Enum}{\cdot}A \in \textsc{Delay}\mathrm{PP}$.

Idea:

- generate solutions randomly
- put them in an efficient data structure (to avoid repetitions)
- coupon collector problem
- amortize the output of solutions

The space used is huge as with all enumeration algorithms using a queue to store all found solutions!

Uniform generation + space implies efficient enumeration

Theorem

If A has a polytime almost uniform random generator, then $\textsc{Enum}\cdot A \in \textsc{Delay}\mathrm{PP}$.

Idea:

- generate solutions randomly
- put them in an efficient data structure (to avoid repetitions)
- coupon collector problem
- amortize the output of solutions

The space used is huge as with all enumeration algorithms using a queue to store all found solutions!

Combinatorial algorithms

1. Gray codes to enumerate all integers of size n in constant delay.
2. Tree enumeration or linear extension of a partial order in constant amortized time by local transformations.

Combinatorial algorithms

1. Gray codes to enumerate all integers of size n in constant delay.
2. Tree enumeration or linear extension of a partial order in constant amortized time by local transformations.
3. Maximal stable sets in lexicographic order using a priority queue and local transformation ($\textsc{DelayP}$)

Combinatorial algorithms

1. Gray codes to enumerate all integers of size n in constant delay.
2. Tree enumeration or linear extension of a partial order in constant amortized time by local transformations.
3. Maximal stable sets in lexicographic order using a priority queue and local transformation ($\textsc{DelayP}$)
4. Saturation of a set of solutions ($\textsc{IncP}$)

Combinatorial algorithms

1. Gray codes to enumerate all integers of size n in constant delay.
2. Tree enumeration or linear extension of a partial order in constant amortized time by local transformations.
3. Maximal stable sets in lexicographic order using a priority queue and local transformation ($\textsc{DelayP}$)
4. Saturation of a set of solutions ($\textsc{IncP}$)
5. Method with a set of forbidden elements and necessary elements ($\textsc{DelayP}$)

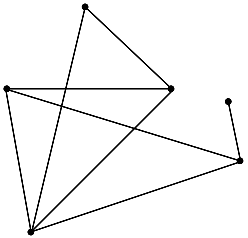
Combinatorial algorithms

1. Gray codes to enumerate all integers of size n in constant delay.
2. Tree enumeration or linear extension of a partial order in constant amortized time by local transformations.
3. Maximal stable sets in lexicographic order using a priority queue and local transformation ($\textsc{DelayP}$)
4. Saturation of a set of solutions ($\textsc{IncP}$)
5. Method with a set of forbidden elements and necessary elements ($\textsc{DelayP}$)
6. Parallel enumeration ($\textsc{DelayP}$)

Combinatorial algorithms

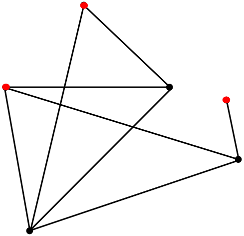
1. Gray codes to enumerate all integers of size n in constant delay.
2. Tree enumeration or linear extension of a partial order in constant amortized time by local transformations.
3. Maximal stable sets in lexicographic order using a priority queue and local transformation ($\textsc{DelayP}$)
4. Saturation of a set of solutions ($\textsc{IncP}$)
5. Method with a set of forbidden elements and necessary elements ($\textsc{DelayP}$)
6. Parallel enumeration ($\textsc{DelayP}$)

Logic algorithms



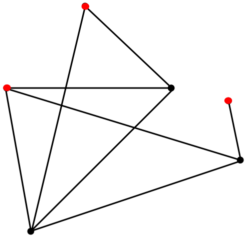
- Stable set $\equiv \forall x \forall y \; x \in I \wedge y \in I \Rightarrow \neg E(x, y)$.

Logic algorithms



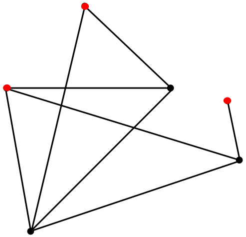
- Stable set $\equiv \forall x \forall y \; x \in I \wedge y \in I \Rightarrow \neg E(x, y)$.
- Question : generate the solutions to this kind of query.

Logic algorithms



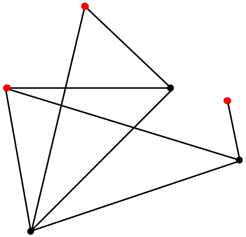
- Stable set $\equiv \forall x \forall y \; x \in I \wedge y \in I \Rightarrow \neg E(x,y)$.
- Question : generate the solutions to this kind of query.
- The delay is good when the formula has no quantifier or only existential ones.

Logic algorithms



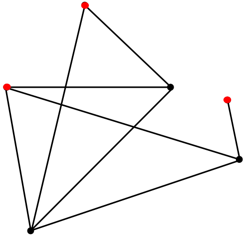
- Stable set $\equiv \forall x \forall y \; x \in I \wedge y \in I \Rightarrow \neg E(x,y)$.
- Question : generate the solutions to this kind of query.
- The delay is good when the formula has no quantifier or only existential ones.
- Solutions of unbounded size.

Logic algorithms



- Stable set $\equiv \forall x \forall y \; x \in I \wedge y \in I \Rightarrow \neg E(x,y)$.
- Question : generate the solutions to this kind of query.
- The delay is good when the formula has no quantifier or only existential ones.
- Solutions of unbounded size.
- Applications: generating maximal stable sets, vertex covers, edges covers, graph colorations . . .

Logic algorithms



- Stable set $\equiv \forall x \forall y \; x \in I \wedge y \in I \Rightarrow \neg E(x,y)$.
- Question : generate the solutions to this kind of query.
- The delay is good when the formula has no quantifier or only existential ones.
- Solutions of unbounded size.
- Applications: generating maximal stable sets, vertex covers, edges covers, graph colorations . . .

Logic algorithms (cont'd)

Variation on the formula and the structure:

- If the formula is an acyclic conjunctive query: linear delay (Simple Paths).
- If the formula is first order + structure of bounded degree: constant delay (Cliques in a Bounded Degree Graph).

Logic algorithms (cont'd)

Variation on the formula and the structure:

- If the formula is an acyclic conjunctive query: linear delay (Simple Paths).
- If the formula is first order + structure of bounded degree: constant delay (Cliques in a Bounded Degree Graph).
- Regular XPath over data trees in constant delay.

Logic algorithms (cont'd)

Variation on the formula and the structure:

- If the formula is an acyclic conjunctive query: linear delay (Simple Paths).
- If the formula is first order + structure of bounded degree: constant delay (Cliques in a Bounded Degree Graph).
- Regular XPath over data trees in constant delay.
- A linear delay algorithm when the formula is MSO and the graph is of bounded treewidth or cliquewidth. Use a combinatorial algorithm on DAGs.

Logic algorithms (cont'd)

Variation on the formula and the structure:

- If the formula is an acyclic conjunctive query: linear delay (Simple Paths).
- If the formula is first order + structure of bounded degree: constant delay (Cliques in a Bounded Degree Graph).
- Regular XPath over data trees in constant delay.
- A linear delay algorithm when the formula is MSO and the graph is of bounded treewidth or cliquewidth. Use a combinatorial algorithm on DAGs.
- Dichotomy for boolean CSP either hard or $\textsc{DelayP}$.

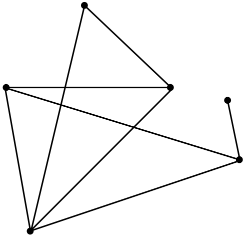
Logic algorithms (cont'd)

Variation on the formula and the structure:

- If the formula is an acyclic conjunctive query: linear delay (Simple Paths).
- If the formula is first order + structure of bounded degree: constant delay (Cliques in a Bounded Degree Graph).
- Regular XPath over data trees in constant delay.
- A linear delay algorithm when the formula is MSO and the graph is of bounded treewidth or cliquewidth. Use a combinatorial algorithm on DAGs.
- Dichotomy for boolean CSP either hard or $\textsc{DelayP}$.

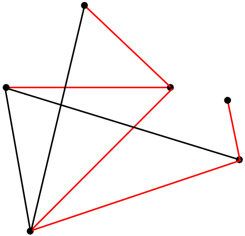
Algebraic algorithms

- Spanning tree $\equiv$ monomials of the determinant of the Kirchoff matrix.



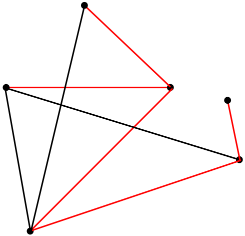
Algebraic algorithms

- Spanning tree ≡ monomials of the determinant of the Kirchoff matrix.
- Question : enumerate the monomials of an easy to evaluate polynomial.



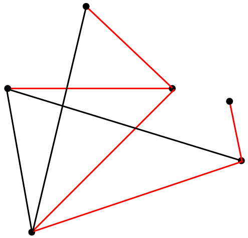
Algebraic algorithms

- Spanning tree $\equiv$ monomials of the determinant of the Kirchoff matrix.
- Question : enumerate the monomials of an easy to evaluate polynomial.
- Easy when the polynomial is multilinear, harder otherwise.



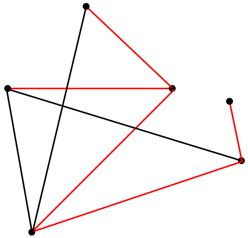
Algebraic algorithms

- Spanning tree $\equiv$ monomials of the determinant of the Kirchoff matrix.
- Question : enumerate the monomials of an easy to evaluate polynomial.
- Easy when the polynomial is multilinear, harder otherwise.
- Probabilistic enumeration.



Algebraic algorithms

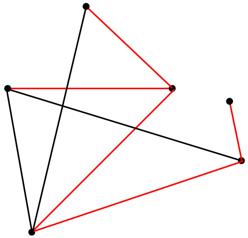
- Spanning tree ≡ monomials of the determinant of the Kirchoff matrix.
- Question : enumerate the monomials of an easy to evaluate polynomial.
- Easy when the polynomial is multilinear, harder otherwise.
- Probabilistic enumeration.



Applications :

Algebraic algorithms

- Spanning tree $\equiv$ monomials of the determinant of the Kirchoff matrix.
- Question : enumerate the monomials of an easy to evaluate polynomial.
- Easy when the polynomial is multilinear, harder otherwise.
- Probabilistic enumeration.

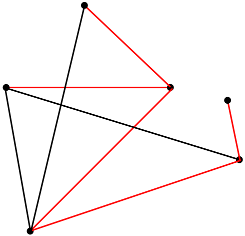


Applications :

- Use known graph polynomials for cycle cover, paths, perfect matchings . . .

Algebraic algorithms

- Spanning tree $\equiv$ monomials of the determinant of the Kirchoff matrix.
- Question : enumerate the monomials of an easy to evaluate polynomial.
- Easy when the polynomial is multilinear, harder otherwise.
- Probabilistic enumeration.

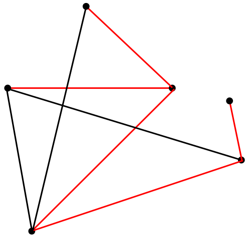


Applications :

- Use known graph polynomials for cycle cover, paths, perfect matchings . . .
- Spanning hypergraphs.

Algebraic algorithms

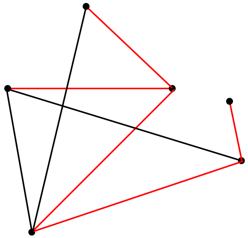
- Spanning tree $\equiv$ monomials of the determinant of the Kirchoff matrix.
- Question : enumerate the monomials of an easy to evaluate polynomial.
- Easy when the polynomial is multilinear, harder otherwise.
- Probabilistic enumeration.



Applications :

- Use known graph polynomials for cycle cover, paths, perfect matchings ...
- Spanning hypergraphs.
- Words separating probabilistic automata.

Algebraic algorithms



- Spanning tree $\equiv$ monomials of the determinant of the Kirchoff matrix.
- Question : enumerate the monomials of an easy to evaluate polynomial.
- Easy when the polynomial is multilinear, harder otherwise.
- Probabilistic enumeration.

Applications :

- Use known graph polynomials for cycle cover, paths, perfect matchings ...
- Spanning hypergraphs.
- Words separating probabilistic automata.

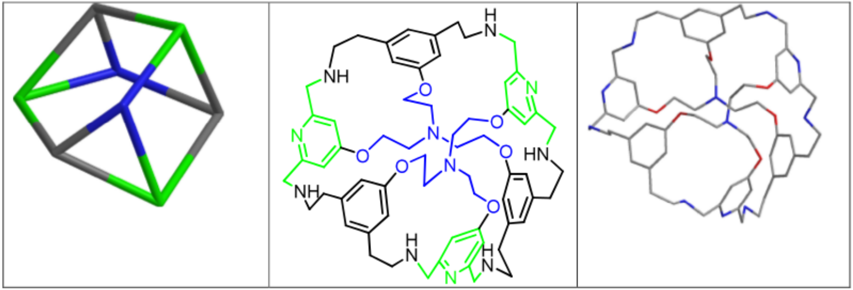
Enumeration Complexity

- Theoretical framework
- Methods for enumeration

A practical enumeration problem from cheminformatics

- Enumeration of planar maps with constraints
- Our algorithm: Kékulé

Nice pictures to impress the layman

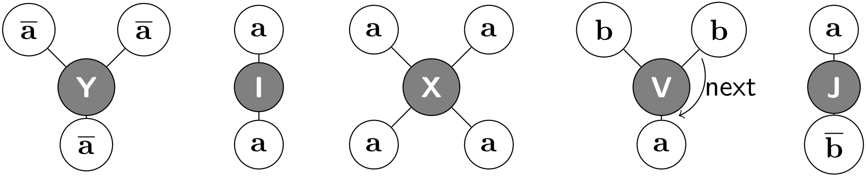


The motifs

Definition

A map $G = (V_{\mathsf{c}}, V, E, \text{next})$ is a **motif** if

1. V_{c} contains only one vertex c called the center
2. each vertex in V is colored with a color in $\mathcal{A}$ a fixed alphabet
3. $E = \{(c, u),\ u \in V\}$
4. next gives an order on the edges of c



Map of motifs

Definition

A connected planar map $G = (V_c, V, E, \text{next})$ is a **map of motifs** based on $\mathcal{M}$ if,

1. each vertex in V is connected to at most one vertex in V, which is of the complementary colour.
2. when all edges between vertices in V are removed, the remaining connected components must all be motifs of $\mathcal{M}$

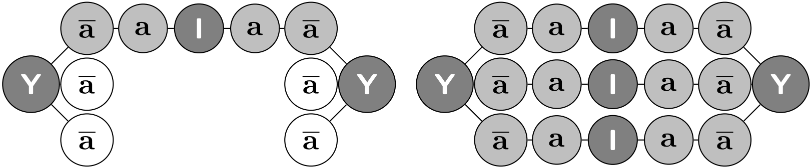


Figure : Example of two maps of motifs based on $\mathcal{M} = \{\mathbf{Y}, \mathbf{I}\}$, the first map is unsaturated while the second map is saturated.

Molecular map

Definition

Let $G = (V_{\mathsf{c}}, V, E_G, \mathsf{next}_G)$ be a saturated map of motifs based on $\mathcal{M}$, we define the **molecular map** $M = (V, E_M, \mathsf{next}_M)$:

1. $V = V_{\mathsf{c}}$
2. $(c_1, c_2) \in E_M$ if it exists a path (c_1, u, v, c_2) in G
3. $\mathsf{next}_M((c, c_1)) = (c, c_2)$ if it exists two paths (c, u_1, v_1, c_1) and (c, u_2, v_2, c_2) in G and $\mathsf{next}_G((c, u_1)) = (c, u_2)$

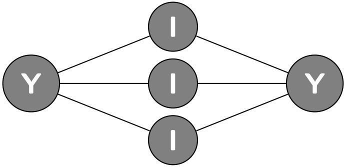


Figure : The molecular map corresponding to the saturated map of motifs in Fig. 1

The indices

Why is a molecular map a good representation of a molecula ?

1. Constraint on the edges: possible chemical connections
2. The *size* of a cut $S = (S_1, S_2)$ is the number of edges with one end in S_1 and the other in S_2.

$$sparsity(S) = \frac{size(S)}{\min(|S_1|, |S_2|)}$$

Sound molecula have high minimum sparsity.

3. Planar graphs and large automorphism groups $\equiv$ spherical shape.
4. A large face in the graph $\equiv$ an entrance in the cage

The indices

Why is a molecular map a good representation of a cage ?

1. Constraint on the edges: possible chemical connections
2. The *size* of a cut $S = (S_1, S_2)$ is the number of edges with one end in S_1 and the other in S_2.

   $$sparsity(S) = \frac{size(S)}{\min(|S_1|, |S_2|)}$$

   Sound molecula have high minimum sparsity.

3. Planar graphs and large automorphism groups $\equiv$ spherical shape.
4. A large face in the graph $\equiv$ an entrance in the cage

The problem

Enumeration problem

We want to generate, given a set of motifs $\mathcal{M}$ and a size n, all molecular maps based on $\mathcal{M}$ and of size n.

The number of maps is exponential in n. We would like an algorithm in $\textsc{DelayP}$ or at least in linear total time.

The problem

Enumeration problem

We want to generate, given a set of motifs $\mathcal{M}$ and a size n, all molecular maps based on $\mathcal{M}$ and of size n.

The number of maps is exponential in n. We would like an algorithm in $\textsc{DelayP}$ or at least in linear total time.

Is it possible to restrict the solutions generated to the ones with a large face? with a good minimum sparsity? a large automorphism group?

The problem

Enumeration problem

We want to generate, given a set of motifs $\mathcal{M}$ and a size n, all molecular maps based on $\mathcal{M}$ and of size n.

The number of maps is exponential in n. We would like an algorithm in $\textsc{DelayP}$ or at least in linear total time.

Is it possible to restrict the solutions generated to the ones with a large face? with a good minimum sparsity? a large automorphism group?

What is the meaning of my previous question?

The problem

Enumeration problem

We want to generate, given a set of motifs $\mathcal{M}$ and a size n, all molecular maps based on $\mathcal{M}$ and of size n.

The number of maps is exponential in n. We would like an algorithm in $\textsc{DelayP}$ or at least in linear total time.

Is it possible to restrict the solutions generated to the ones with a large face? with a good minimum sparsity? a large automorphism group?

What is the meaning of my previous question?

A three steps approach

1. Generate the backbones which are simple maps of motifs
2. From each backbone we compute all saturated maps of motifs we can obtain

A three steps approach

1. Generate the backbones which are simple maps of motifs
2. From each backbone we compute all saturated maps of motifs we can obtain
3. Compute the indices of the solutions generated

A three steps approach

1. Generate the backbones which are simple maps of motifs
2. From each backbone we compute all saturated maps of motifs we can obtain
3. Compute the indices of the solutions generated

Issue: a solution can be obtained several times. No guarantee on this number.

A three steps approach

1. Generate the backbones which are simple maps of motifs
2. From each backbone we compute all saturated maps of motifs we can obtain
3. Compute the indices of the solutions generated

Issue: a solution can be obtained several times. No guarantee on this number.

Our (bad) method: Store all solutions in a self balanced tree and do an isomorphism test for each new solution.

A three steps approach

1. Generate the backbones which are simple maps of motifs
2. From each backbone we compute all saturated maps of motifs we can obtain
3. Compute the indices of the solutions generated

Issue: a solution can be obtained several times. No guarantee on this number.

Our (bad) method: Store all solutions in a self balanced tree and do an isomorphism test for each new solution.

The less the steps, the better the algorithm!

A three steps approach

1. Generate the backbones which are simple maps of motifs
2. From each backbone we compute all saturated maps of motifs we can obtain
3. Compute the indices of the solutions generated

Issue: a solution can be obtained several times. No guarantee on this number.

Our (bad) method: Store all solutions in a self balanced tree and do an isomorphism test for each new solution.

The less the steps, the better the algorithm!

The backbones

We generate different families of backbones. Their free vertices (of degree 1) will be folded to get a saturated map.

First idea: generate trees. Since every connected map has a spanning tree, it will make the generation exhaustive.

To generate them we use a bruteforce method and an isomorphism test.

The backbones

We generate different families of backbones. Their free vertices (of degree 1) will be folded to get a saturated map.

First idea: generate trees. Since every connected map has a spanning tree, it will make the generation exhaustive.

To generate them we use a bruteforce method and an isomorphism test.

Open Problem: find a CAT algorithm to generate maps of motifs which are trees

The backbones

We generate different families of backbones. Their free vertices (of degree 1) will be folded to get a saturated map.

First idea: generate trees. Since every connected map has a spanning tree, it will make the generation exhaustive.

To generate them we use a bruteforce method and an isomorphism test.

Open Problem: find a CAT algorithm to generate maps of motifs which are trees

A new path

Second idea: paths are simpler than trees.

Bruteforce method: add at the end of a path any possible motif until the path is of the right size.

A new path

Second idea: paths are simpler than trees.

Bruteforce method: add at the end of a path any possible motif until the path is of the right size.

The only isomorphic paths are obtained by reversing a path. We get a CAT algorithm by discarding the non canonical paths.

A new path

Second idea: paths are simpler than trees.

Bruteforce method: add at the end of a path any possible motif until the path is of the right size.

The only isomorphic paths are obtained by reversing a path. We get a CAT algorithm by discarding the non canonical paths.

Drawback: not every planar map has an Hamiltonian circuit. But all planar cubic 3-connected graphs of size less than 38 are Hamiltonian.

A new path

Second idea: paths are simpler than trees.

Bruteforce method: add at the end of a path any possible motif until the path is of the right size.

The only isomorphic paths are obtained by reversing a path. We get a CAT algorithm by discarding the non canonical paths.

Drawback: not every planar map has an Hamiltonian circuit. But all planar cubic 3-connected graphs of size less than 38 are Hamiltonian.

The same can be done with cycles instead of paths.

A new path

Second idea: paths are simpler than trees.

Bruteforce method: add at the end of a path any possible motif until the path is of the right size.

The only isomorphic paths are obtained by reversing a path. We get a CAT algorithm by discarding the non canonical paths.

Drawback: not every planar map has an Hamiltonian circuit. But all planar cubic 3-connected graphs of size less than 38 are Hamiltonian.

The same can be done with cycles instead of paths.

Fold and outline

The fold operation on the vertices u and v is adding the edge (u, v) to G. Valid when u and v are:

1. free
2. of complementary colors
3. in the same face of G

A map is foldable when by successive fold operations we can turn it into a saturated map.

Fold and outline

The fold operation on the vertices u and v is adding the edge (u, v) to G. Valid when u and v are:

1. free
2. of complementary colors
3. in the same face of G

A map is foldable when by successive fold operations we can turn it into a saturated map.

The outline of a face is the list in order of traversal of the free vertices. When the backbone is a tree or a path there is a single outline.

Fold and outline

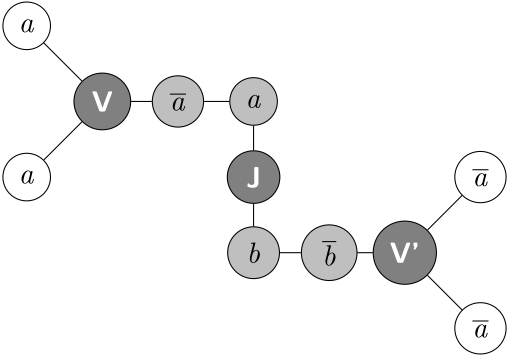
The fold operation on the vertices u and v is adding the edge (u, v) to G. Valid when u and v are:

1. free
2. of complementary colors
3. in the same face of G

A map is foldable when by successive fold operations we can turn it into a saturated map.

The outline of a face is the list in order of traversal of the free vertices. When the backbone is a tree or a path there is a single outline.

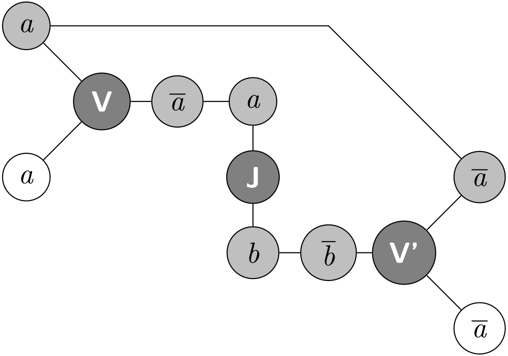
Example



outline $= \{a, \overline{a}, \overline{a}, a\}$

Figure : A map of three motifs on $\mathcal{A}_M = \{\mathbf{V}, \mathbf{V}', \mathbf{J}\}$ and its outline before a fold operation.

Example



outline $= \{\overline{a}, a\}$

Figure : A map of three motifs on $\mathcal{A}_M = \{\mathbf{V}, \mathbf{V}', \mathbf{J}\}$ and its outline after a fold operation.

When is a map foldable?

The outline is a circular sequence of vertices. The fold remove two vertices of compatible colours.

Enough to work with the sequence of colours of the vertices. In the previous example $a\bar{a}\bar{a}a$.

When is a map foldable?

The outline is a circular sequence of vertices. The fold remove two vertices of compatible colours.

Enough to work with the sequence of colours of the vertices. In the previous example $a\bar{a}\bar{a}a$.

Definition

A word is a Dyck word if we can reduce it to the empty word by removing consecutive complementary letters.

When is a map foldable?

The outline is a circular sequence of vertices. The fold remove two vertices of compatible colours.

Enough to work with the sequence of colours of the vertices. In the previous example $a\bar{a}\bar{a}a$.

Definition

A word is a Dyck word if we can reduce it to the empty word by removing consecutive complementary letters.

Lemma

A map is foldable if and only if the associated word is a Dyck word.

When is a map foldable?

The outline is a circular sequence of vertices. The fold remove two vertices of compatible colours.

Enough to work with the sequence of colours of the vertices. In the previous example $a\bar{a}\bar{a}a$.

Definition

A word is a Dyck word if we can reduce it to the empty word by removing consecutive complementary letters.

Lemma

A map is foldable if and only if the associated word is a Dyck word.

This yields a linear time algorithm to test whether a map is foldable.

When is a map foldable?

The outline is a circular sequence of vertices. The fold remove two vertices of compatible colours.

Enough to work with the sequence of colours of the vertices. In the previous example $a\bar{a}\bar{a}a$.

Definition

A word is a Dyck word if we can reduce it to the empty word by removing consecutive complementary letters.

Lemma

A map is foldable if and only if the associated word is a Dyck word.

This yields a linear time algorithm to test whether a map is foldable.

How to avoid non foldable maps?

Definition

A map is almost foldable if for every letter in $a \in \mathcal{A}$, there are as many vertices labeled with a and $\bar{a}$.

Since a foldable backbone is always almost foldabe, we would like to enumerate almost foldable backbones only.

How to avoid non foldable maps?

Definition

A map is almost foldable if for every letter in $a \in \mathcal{A}$, there are as many vertices labeled with a and $\bar{a}$.

Since a foldable backbone is always almost foldabe, we would like to enumerate almost foldable backbones only.

Dynamic programming algorithm:
k is the number of positive letters in $\mathcal{A}$.
We generate a k-dimensional array which allows to decide whether a path can be extended by l motifs and be almost foldable.

Takes $O(n^{k+1})$ but there are about C^n paths and it reduces their number by a large factor.

How to avoid non foldable maps?

Definition

A map is almost foldable if for every letter in $a \in \mathcal{A}$, there are as many vertices labeled with a and $\bar{a}$.

Since a foldable backbone is always almost foldabe, we would like to enumerate almost foldable backbones only.

Dynamic programming algorithm:
k is the number of positive letters in $\mathcal{A}$.
We generate a k-dimensional array which allows to decide whether a path can be extended by l motifs and be almost foldable.

Takes $O(n^{k+1})$ but there are about C^n paths and it reduces their number by a large factor.

How to fold a map?

We call result of a sequence of reductions the set of pairs (i, j) such that the sequence has paired i and j.

Problem: given a word, we want to generate all different results of sequences of reduction which yields an empty word.

How to fold a map?

We call result of a sequence of reductions the set of pairs (i, j) such that the sequence has paired i and j.

Problem: given a word, we want to generate all different results of sequences of reduction which yields an empty word.

Another dynamic programming algorithm:

How to fold a map?

We call result of a sequence of reductions the set of pairs (i, j) such that the sequence has paired i and j.

Problem: given a word, we want to generate all different results of sequences of reduction which yields an empty word.

Another dynamic programming algorithm:

- Build the matrix M such that $M_{i,j}$ is true if and only if the subword $w_i \dots w_j$ is foldable.

How to fold a map?

We call result of a sequence of reductions the set of pairs (i, j) such that the sequence has paired i and j.

Problem: given a word, we want to generate all different results of sequences of reduction which yields an empty word.

Another dynamic programming algorithm:

- Build the matrix M such that $M_{i,j}$ is true if and only if the subword $w_i \dots w_j$ is foldable.
- In the enumeration algorithm a partially folded word is a set of subwords.

How to fold a map?

We call result of a sequence of reductions the set of pairs (i, j) such that the sequence has paired i and j.

Problem: given a word, we want to generate all different results of sequences of reduction which yields an empty word.

Another dynamic programming algorithm:

- Build the matrix M such that $M_{i,j}$ is true if and only if the subword $w_i \dots w_j$ is foldable.
- In the enumeration algorithm a partially folded word is a set of subwords.
- At each step reduce the first non folded letter with all possible letter given by M.

How to fold a map?

We call result of a sequence of reductions the set of pairs (i, j) such that the sequence has paired i and j.

Problem: given a word, we want to generate all different results of sequences of reduction which yields an empty word.

Another dynamic programming algorithm:

- Build the matrix M such that $M_{i,j}$ is true if and only if the subword $w_i \dots w_j$ is foldable.
- In the enumeration algorithm a partially folded word is a set of subwords.
- At each step reduce the first non folded letter with all possible letter given by M.
- The preprocessing is in $O(n^3)$ and the delay is linear.

How to fold a map?

We call result of a sequence of reductions the set of pairs (i, j) such that the sequence has paired i and j.

Problem: given a word, we want to generate all different results of sequences of reduction which yields an empty word.

Another dynamic programming algorithm:

- Build the matrix M such that $M_{i,j}$ is true if and only if the subword $w_i \dots w_j$ is foldable.
- In the enumeration algorithm a partially folded word is a set of subwords.
- At each step reduce the first non folded letter with all possible letter given by M.
- The preprocessing is in $O(n^3)$ and the delay is linear.

Additional computations

Some hidden costs:

1. Isomorphism test for each produced map: $O(n^2)$. A signature is the trace of a DFS. We keep the smallest when trying every starting point.

For each non isomorphic map we must compute indices.

Additional computations

Some hidden costs:

1. Isomorphism test for each produced map: $O(n^2)$. A signature is the trace of a DFS. We keep the smallest when trying every starting point.

   For each non isomorphic map we must compute indices.
2. Computing all faces and their sizes: $O(n)$

Additional computations

Some hidden costs:

1. Isomorphism test for each produced map: $O(n^2)$. A signature is the trace of a DFS. We keep the smallest when trying every starting point.

   For each non isomorphic map we must compute indices.
2. Computing all faces and their sizes: $O(n)$
3. The equivalence class of each vertex: $O(n^3)$

Additional computations

Some hidden costs:

1. Isomorphism test for each produced map: $O(n^2)$. A signature is the trace of a DFS. We keep the smallest when trying every starting point.

   For each non isomorphic map we must compute indices.
2. Computing all faces and their sizes: $O(n)$
3. The equivalence class of each vertex: $O(n^3)$
4. Computing the minimum sparsity of a map. Currently Gray Code to generate all partitions: $O(2^n)$.
   Problem NP-hard in general but cubic algorithm for planar graphs.

Additional computations

Some hidden costs:

1. Isomorphism test for each produced map: $O(n^2)$. A signature is the trace of a DFS. We keep the smallest when trying every starting point.

   For each non isomorphic map we must compute indices.
2. Computing all faces and their sizes: $O(n)$
3. The equivalence class of each vertex: $O(n^3)$
4. Computing the minimum sparsity of a map. Currently Gray Code to generate all partitions: $O(2^n)$.
   Problem NP-hard in general but cubic algorithm for planar graphs.

Thanks!

Thanks!

Thanks,

Thanks!

Thanks, thanks,

Thanks!

Thanks, thanks, thanks,

Thanks!

Thanks, thanks, thanks, thanks,

Thanks!

Thanks, thanks, thanks, thanks, thanks,

Thanks!

Thanks, thanks, thanks, thanks, thanks, thanks,

Thanks!

Thanks, thanks, thanks, thanks, thanks, thanks, thanks,

Thanks!

Thanks, thanks, thanks, thanks, thanks, thanks, thanks, thanks,

Thanks!

Thanks, thanks, thanks, thanks, thanks, thanks, thanks, thanks, thanks,

Thanks!

Thanks, thanks, thanks, thanks, thanks, thanks, thanks, thanks, thanks, thanks

Thanks!

Thanks, thanks, thanks, thanks, thanks, thanks, thanks, thanks, thanks, thanks
Let's all do enumeration

Open Questions

Enumeration:

1. Separate DelayP from IncP modulo ETH
2. Design a reduction compatible with low enumeration classes.

Cheminformatics:

1. A CAT algorithm to generate maps of motifs which are trees
2. A smaller family of backbones which still make the generation exhaustive.
3. A better fold algorithm (constant delay, linear precomputation).
4. A way to avoid some foldings.