Computational complexity of partitioning a graph into connected subgraphs

J. Bensmail
LaBRI, Bordeaux University, Talence, France

AGH University of Science and Technology
October 2nd, 2012

Part 1: Catching how hard is a decision problem

Part 2: Partitioning a graph following a sequence

Part 3: On some generalized graph partition problems

Part 4: A recursive version of the graph partition problem

Decision problems

A *decision problem* A is a *yes-no question* regarding the values of some *input parameters*.

An *instance* I *of* A is an application of A to some specific input parameters. If the answer to I is *yes*, then I is a *yes-instance of* A. Otherwise, we say that I is a *no-instance of* A.

Decision problems

A *decision problem A* is a *yes-no question* regarding the values of some *input parameters*.

An *instance I of A* is an application of *A* to some specific input parameters. If the answer to *I* is *yes*, then *I* is a *yes-instance of A*. Otherwise, we say that *I* is a *no-instance of A*.

> Ex. Prime Number
> Instance: A positive integer n.
> Question: Is n a prime number?

For instance:

- (3) is a yes-instance of Prime Number;
- (4) is a no-instance of Prime Number.

Complexity of a decision problem

The *complexity of* A represents *how hard* is it to deal with A. It is measured as the amount of computational resources needed by any algorithm to decide whether an instance of A is a yes- or no-instance.

The efficiency of such an algorithm is thus usually evaluated regarding how much *time* and *space* it needs to give an answer for a specific instance of A. An amount of one of these two kinds of resources is "good" if it is *polynomial in the instance size*. Conversely, it is "bad" if it is *exponential in the instance size*.

In computational complexity theory, problems are organized into *complexity classes*. At the moment, there exist plenty of complexity classes gathering problems with similar complexity together.

Deterministic poly-time problems - P problems

P is the class of decision problems for which there exists a polynomial-time solving algorithm.

Deterministic poly-time problems - P problems

P is the class of decision problems for which there exists a polynomial-time solving algorithm.

Some examples of problems known to be in P:

- is n prime?
- is an array sorted?
- does G admit a perfect matching?
- etc...

Non-deterministic poly-time problems - NP and co-NP problems

NP is the class of decision problems for which there exists an algorithm deciding whether one of their instances is a yes-instance using an algorithm dealing with a problem in P.

Clearly, any P problem is also in NP.

If A is in $NP - P$, then it means that there is no polynomial-time algorithm going straight to the answer for a given instance I of A. But there exists a polynomial-time *checking algorithm* deciding whether I is a yes-instance of A regarding some input parameters.

Non-deterministic poly-time problems - NP and co-NP problems

NP is the class of decision problems for which there exists an algorithm deciding whether one of their instances is a yes-instance using an algorithm dealing with a problem in P.

Clearly, any P problem is also in NP.

If A is in $NP - P$, then it means that there is no polynomial-time algorithm going straight to the answer for a given instance I of A. But there exists a polynomial-time *checking algorithm* deciding whether I is a yes-instance of A regarding some input parameters.

Some examples of NP problems:

- does there exist a vertex cover of G with size at most k?
- given a set of integers, does it have a subset that sums up to 0?
- is it possible to colour G with three colors?
- etc...

Non-deterministic poly-time problems - NP and co-NP problems

NP is the class of decision problems for which there exists an algorithm deciding whether one of their instances is a yes-instance using an algorithm dealing with a problem in P.

Clearly, any P problem is also in NP.

If A is in $NP - P$, then it means that there is no polynomial-time algorithm going straight to the answer for a given instance I of A. But there exists a polynomial-time *checking algorithm* deciding whether I is a yes-instance of A regarding some input parameters.

Some examples of NP problems:

- does there exist a vertex cover of G with size at most k?
- given a set of integers, does it have a subset that sums up to 0?
- is it possible to colour G with three colors?
- etc...

It seems hard to believe that a problem A in $NP - P$ is also in P since there are generally an exponential number of *things* to check before being able to decide whether a single instance I of A is a yes- or no-instance.

Non-deterministic poly-time problems - *NP* and co-*NP* problems

The co-*NP* class is almost the same as the *NP* one, except that the checking polynomial-time algorithm concerns no-instances of a decision problem.

Non-deterministic poly-time problems - *NP* and co-*NP* problems

The co-*NP* class is almost the same as the *NP* one, except that the checking polynomial-time algorithm concerns no-instances of a decision problem.

In particular, the complement of any *NP* problem is a co-*NP* problem:

- is it true that every vertex cover of G has size at least k?
- given a set of integers, does no subset sum up to 0?
- cannot G be coloured with three colors?
- etc...

Non-deterministic poly-time problems - NP and co-NP problems

The co-NP class is almost the same as the NP one, except that the checking polynomial-time algorithm concerns no-instances of a decision problem.

In particular, the complement of any NP problem is a co-NP problem:

- is it true that every vertex cover of G has size at least k?
- given a set of integers, does no subset sum up to 0?
- cannot G be coloured with three colors?
- etc...

Observe that if A is in P, then A is in $NP \cap$ co-NP.

Reducible problems

Given two problems A and B, we say that A is *reducible to* B if there exists a function that maps instances of A to instances of B in such a way that if we manage to solve B, then we also solve A. The existence of a polynomial-time reduction from A to B is denoted by $A \leq_p B$.

Reducible problems

Given two problems A and B, we say that A is *reducible to* B if there exists a function that maps instances of A to instances of B in such a way that if we manage to solve B, then we also solve A. The existence of a polynomial-time reduction from A to B is denoted by $A \leq_p B$.

Let us now consider the following well-known decision problem.

> BOOLEAN SATISFIABILITY - SAT
> Instance: A CNF formula F over variables $x_1, ..., x_n$.
> Question: Is F satisfiable?

A quick reminder:

- a *boolean variable* x_i can be either assigned to *true* or *false*;
- a *literal* l_i is either a boolean variable x_j or its negation $\overline{x_j}$;
- the *conjunction* $l_i \wedge l_j$ of two literals is evaluated true iff l_i and l_j are both true;
- the *disjunction* $l_i \vee l_j$ of two literals is evaluated true iff at least one of l_i and l_j is true;
- a *clause* C_i is a disjunction of several literals;
- a *CNF formula* is a conjunction of several clauses;
- a CNF formula is *satisfiable* iff there exists a truth assignment of its variables that makes it evaluated true.

Complete problems in *NP* - *NP*-complete problems

Boolean Satisfiability - SAT
Instance: A CNF formula F over variables $x_1, ..., x_n$.
Question: Is F satisfiable?

For example, $F = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_2 \vee x_3)$ is a satisfiable CNF formula.

Boolean Satisfiability - SAT
Instance: A CNF formula F over variables $x_1, ..., x_n$.
Question: Is F satisfiable?

For example, $F = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_2 \vee x_3)$ is a satisfiable CNF formula.

Clearly, SAT is in *NP*. Moreover, Cook-Levin theorem states that every *NP* problem A is reducible to SAT in polynomial time. Thus, SAT can be viewed as one of the *hardest problems* of *NP* since solving it under polynomial time would imply that any *NP* problem is solvable in polynomial time too.

The complexity class of the hardest problems in *NP* is the *NP*-complete class. As explained, we consider, as a starting point, that SAT is complete in *NP*. Then, a problem A is a *NP*-complete problem iff:

- A is in *NP*,
- there exists a *NP*-complete problem B such that $B \leq_p A$.

If you manage to solve one particular *NP*-complete problem in polynomial time, then you can solve every *NP*-complete problem in polynomial time by transitivity, and also every *NP* problem...

Part 1: Catching how hard is a decision problem

Part 2: Partitioning a graph following a sequence

Part 3: On some generalized graph partition problems

Part 4: A recursive version of the graph partition problem

Partitioning graphs into connected subgraphs

Let us now consider the following definition...

Def. *Realizable sequence - Realization*

Let G be a graph. A sequence $\tau = (n_1, ..., n_k)$ of positive integers summing up to $|V(G)|$ is *realizable in* G if there exists a partition $(V_1, ..., V_k)$ of $V(G)$ such that every V_i has size n_i and induces a connected subgraph of G. The partition $(V_1, ..., V_k)$ of $V(G)$ is a *realization of* τ *in* G.

... and the associated decision problem.

Realizable Sequence - RealSeq

Instance: A graph G and a sequence τ.
Question: Is τ realizable in G?

On the complexity of RealSeq

It is already known that RealSeq is a *NP*-complete problem even when:

- $\tau = (k, ..., k)$, where k is a divisor of $|V(G)|$ [DF85];
- G is a tree with maximum degree 3 [BF06].

These results have been proved by reduction from the Planar 3-Dimensional Matching and Exact Cover By 3-Sets problems, respectively.

On the complexity of RealSeq

It is already known that RealSeq is a *NP*-complete problem even when:

- $\tau = (k, ..., k)$, where k is a divisor of $|V(G)|$ [DF85];
- G is a tree with maximum degree 3 [BF06].

These results have been proved by reduction from the Planar 3-Dimensional Matching and Exact Cover By 3-Sets problems, respectively.

We give another proof of this result showing that RealSeq remains a *NP*-complete problem even if τ has only two elements. Our reduction is from the following 1-in-3 SAT problem, where a *3CNF formula* is a CNF formula whose clauses have exactly three literals.

> 1-in-3 SAT
> Instance: A 3CNF formula F over variables $x_1, ..., x_n$.
> Question: Is F satisfiable in such a way that each of its clauses has exactly one true literal?

RealSeq is complete in *NP*

> Thm. *B. - 2012*
> RealSeq is complete in *NP* even if τ has exactly two elements.

Proof.
First notice that RealSeq is in *NP*. One can provide a satisfying partition of G to an algorithm that makes sure that each part has the correct size and induces a connected subgraph of G. This can be done in polynomial time.

RealSeq is complete in *NP*

Thm. *B. - 2012*
RealSeq is complete in *NP* even if τ has exactly two elements.

Proof.
First notice that RealSeq is in *NP*. One can provide a satisfying partition of G to an algorithm that makes sure that each part has the correct size and induces a connected subgraph of G. This can be done in polynomial time.

We now show that 1-in-3 SAT $\leq_p$ RealSeq. For a given 3CNF formula F over variables $x_1, ..., x_n$ and clauses $C_1, ..., C_m$ we construct a graph G and a sequence $\tau = (n_1, n_2)$ such that

F is satisfiable in such a way that each of its clauses has only one true literal

$$\Leftrightarrow$$

τ is realizable in G.

Reduction from 1-IN-3 SAT to REALSEQ

Thm. *B. - 2012*
REALSEQ is complete in *NP* even if τ has exactly two elements.

We may suppose that every literal appears in F - if x_i does not appear in F, then

$$F' = F \wedge (x_i \vee \overline{x_i} \vee x_{n+1}) \wedge (x_{n+1} \vee \overline{x_{n+1}} \vee x_{n+1})$$

is satisfiable iff F is satisfiable.

Reduction from 1-IN-3 SAT to REALSEQ

Thm. *B. - 2012*
REALSEQ is complete in *NP* even if τ has exactly two elements.

We may suppose that every literal appears in F - if x_i does not appear in F, then

$$F' = F \wedge (x_i \vee \overline{x_i} \vee x_{n+1}) \wedge (x_{n+1} \vee \overline{x_{n+1}} \vee x_{n+1})$$

is satisfiable iff F is satisfiable.

We first construct the *clause subgraph* of G:

- to each literal l_i in F is associated a *literal vertex* v_{l_i} in G;
- a pair of literal vertices $\{v_{l_i}, v_{l_j}\}$ is linked to the root of a star S_{n+1} if l_i and l_j
  - are a variable and its negation, or
  - appear in a same clause of F;
- a *control vertex* o in G is adjacent to all literal vertices and to n pendant vertices.

Reduction from 1-IN-3 SAT to REALSEQ

Thm. *B. - 2012*
REALSEQ is complete in *NP* even if τ has exactly two elements.

Let n_2 be the number of vertices of the clause subgraph of G. Then

$$n_2 \leq 2n + n(n+1) + 3m(n+1) + n + 1$$
$$n_2 \leq n(n + 3m(1 + 1/n) + 4) + 1.$$

Reduction from 1-IN-3 SAT to REALSEQ

Thm. *B. - 2012*
REALSEQ is complete in *NP* even if τ has exactly two elements.

Let n_2 be the number of vertices of the clause subgraph of G. Then

$$n_2 \leq 2n + n(n+1) + 3m(n+1) + n + 1$$
$$n_2 \leq n(n + 3m(1 + 1/n) + 4) + 1.$$

G is finally augmented with a *base subgraph* as follows:

- for each clause C_i of F, we add a new *clause vertex* v_{C_i} to G;
- each vertex v_{C_i} is linked to $n_2 - n$ pendant vertices;
- for each $i \in [1, m-1]$, we add $v_{C_i}v_{C_{i+1}}$ to $E(G)$;
- if $C_i = (l_{i_1} \vee l_{i_2} \vee l_{i_3})$, then we add $v_{C_i}v_{l_{i_1}}$, $v_{C_i}v_{l_{i_2}}$ and $v_{C_i}v_{l_{i_3}}$ to $E(G)$.

We added $n_1 = m(n_2 - n + 1)$ vertices to G.

Reduction from 1-IN-3 SAT to REALSEQ

Thm. *B. - 2012*
REALSEQ is complete in *NP* even if τ has exactly two elements.

Let n_2 be the number of vertices of the clause subgraph of G. Then

$$n_2 \leq 2n + n(n+1) + 3m(n+1) + n + 1$$
$$n_2 \leq n(n + 3m(1 + 1/n) + 4) + 1.$$

G is finally augmented with a *base subgraph* as follows:

- for each clause C_i of F, we add a new *clause vertex* v_{C_i} to G;
- each vertex v_{C_i} is linked to $n_2 - n$ pendant vertices;
- for each $i \in [1, m-1]$, we add $v_{C_i}v_{C_{i+1}}$ to $E(G)$;
- if $C_i = (l_{i_1} \vee l_{i_2} \vee l_{i_3})$, then we add $v_{C_i}v_{l_{i_1}}$, $v_{C_i}v_{l_{i_2}}$ and $v_{C_i}v_{l_{i_3}}$ to $E(G)$.

We added $n_1 = m(n_2 - n + 1)$ vertices to G.

Finally, $|V(G)| = n_1 + n_2$. Let us consider $\tau = (n_1 + n, n_2 - n)$.

Reduction from 1-IN-3 SAT to REALSEQ

Thm. *B. - 2012*
REALSEQ is complete in *NP* even if τ has exactly two elements.

Observe that if a part of the realization contains the root of an induced star, then it also has to cover all of its pending vertices. Hence, in a realization of τ in G, the base subgraph has to be covered by the part V_1 of size $n_1 + n$.

Once the base subgraph is covered by V_1, this part is missing n additional vertices from the clause subgraph of G. Because of the structure of the clause subgraph, we may only pick up some literal vertices. It has to be done in such a way that the clause subgraph remains connected so that it can eventually be considered as the part V_2 with size $n_2 - n$.

Reduction from 1-IN-3 SAT to REALSEQ

Thm. *B. - 2012*
REALSEQ is complete in *NP* even if τ has exactly two elements.

Observe that if a part of the realization contains the root of an induced star, then it also has to cover all of its pending vertices. Hence, in a realization of τ in G, the base subgraph has to be covered by the part V_1 of size $n_1 + n$.

Once the base subgraph is covered by V_1, this part is missing n additional vertices from the clause subgraph of G. Because of the structure of the clause subgraph, we may only pick up some literal vertices. It has to be done in such a way that the clause subgraph remains connected so that it can eventually be considered as the part V_2 with size $n_2 - n$.

Choosing a literal vertex v_{l_i} to belong to V_1 is like setting l_i to true. In particular:

- two covered literal vertices cannot be both linked to a same clause vertex;
- two covered literal vertices cannot be a variable and its negation.

Finally, a realization of τ in G exists iff F is satisfiable in such a way that each of its clauses has only one true literal. Moreover, G has a polynomial number of vertices regarding the size of F. Thus, the reduction is performed in polynomial time.

■

Another graph partition problem

Let us now consider the following stronger definition...

Def. *Prescription - Realization under prescription*

A k-tuple $(v_1, ..., v_k)$ of pairwise distinct vertices of G is called a *k-prescription of G*. If $p \geq k$ and there exists a realization $(V_1, ..., V_p)$ of τ in G such that for every $i \in [1, k]$ we have $v_i \in V_i$, then τ is *realizable in G under* $(v_1, ..., v_k)$.

... and the associated decision problem.

Prescriptible Sequence - PrescSeq

Instance: A graph G, a sequence τ and a prescription P of G.
Question: Is τ realizable in G under P?

PrescSeq is complete in *NP*

Prescriptible Sequence - PrescSeq
Instance: A graph G, a sequence τ and a prescription P of G.
Question: Is τ realizable in G under P?

Surprisingly, this problem is equivalent to RealSeq without any regard to the prescription size.

Thm. *B. - 2012*
PrescSeq is complete in *NP* no matter how long is the prescription.

Proof.
It is possible to modify the checking algorithm for RealSeq in such a way that it also makes sure that the vertices of the prescription belong to the associated parts. This modification does not alter the complexity of the algorithm. Therefore, PrescSeq is in *NP*.

Reduction from RealSeq to PrescSeq

> Thm. *B. - 2012*
> PrescSeq is complete in *NP* no matter how long is the prescription.

We now show that PrescSeq is complete in *NP* by reduction from RealSeq. Given a graph G and a sequence τ, we have to construct a graph G', a sequence τ' and a prescription P of G' such that

$$\tau \text{ is realizable in } G$$
$$\Leftrightarrow$$
$$\tau' \text{ is realizable in } G' \text{ under } P.$$

Reduction from RealSeq to PrescSeq

Thm. *B. - 2012*
PrescSeq is complete in *NP* no matter how long is the prescription.

We now show that PrescSeq is complete in *NP* by reduction from RealSeq. Given a graph G and a sequence τ, we have to construct a graph G', a sequence τ' and a prescription P of G' such that

$$\tau \text{ is realizable in } G$$
$$\Leftrightarrow$$
$$\tau' \text{ is realizable in } G' \text{ under } P.$$

Consider v an arbitrary vertex of G, and link v to one extremity of a path on n_p vertices. Let us denote by v_p the other endvertex of this path, and by G' the resulting graph.

Then observe that if $\tau = (n_1, ..., n_k)$, then $\tau' = (n_p, n_1, ..., n_k)$ is realizable in G' under (v_p) iff τ is realizable in G since there is only one connected subgraph of G' with order n_p that contains v_p.

Reduction from RealSeq to PrescSeq

Thm. *B. - 2012*
PrescSeq is complete in *NP* no matter how long is the prescription.

We now show that PrescSeq is complete in *NP* by reduction from RealSeq. Given a graph G and a sequence τ, we have to construct a graph G', a sequence τ' and a prescription P of G' such that

$$\tau \text{ is realizable in } G$$
$$\Leftrightarrow$$
$$\tau' \text{ is realizable in } G' \text{ under } P.$$

Consider v an arbitrary vertex of G, and link v to one extremity of a path on n_p vertices. Let us denote by v_p the other endvertex of this path, and by G' the resulting graph.

Then observe that if $\tau = (n_1, ..., n_k)$, then $\tau' = (n_p, n_1, ..., n_k)$ is realizable in G' under (v_p) iff τ is realizable in G since there is only one connected subgraph of G' with order n_p that contains v_p.

Of course, this graph and sequence transformations can be repeated at will so that the prescription size grows as wanted. ■

Part 1: Catching how hard is a decision problem

Part 2: Partitioning a graph following a sequence

Part 3: On some generalized graph partition problems

Part 4: A recursive version of the graph partition problem

What about some generalized problems?

Once again, we consider a definition...

> Def. *AP graph*
> A graph G is *arbitrarily partitionable* if it can be partitioned following every sequence that sums up to $|V(G)|$.

... and the decision problem related to it.

> AP Graph
> Instance: A graph G.
> Question: Is G an AP graph?

Because the number of sequences we have to consider is exponential in $|V(G)|$ and RealSeq is a *NP*-complete problem, this problem does not seem to belong to either *NP* or co-*NP*.

Beyond the NP and co-NP classes

Sometimes, the complexity inherent to a decision problem A cannot be caught by the definitions of the NP and co-NP classes. It means that we cannot design a polynomial-time checking algorithm for A based on a subalgorithm dealing with a problem in P. But in some situations we feel that we could design such an algorithm thanks to a "stronger" subalgorithm - one dealing with a problem that lies in either NP or co-NP typically.

Beyond the NP and co-NP classes

Sometimes, the complexity inherent to a decision problem A cannot be caught by the definitions of the NP and co-NP classes. It means that we cannot design a polynomial-time checking algorithm for A based on a subalgorithm dealing with a problem in P. But in some situations we feel that we could design such an algorithm thanks to a "stronger" subalgorithm - one dealing with a problem that lies in either NP or co-NP typically.

Consequently, the NP and co-NP classes were generalized in the following way:

- $\Delta_0^p = \Sigma_0^p = \Pi_0^p = P$;
- $\Delta_{i+1}^p = P^{\Sigma_i^p}$;
- $\Sigma_{i+1}^p = NP^{\Sigma_i^p}$;
- $\Pi_{i+1}^p = \text{co-}NP^{\Sigma_i^p}$;

where

- P^C is the class of problems that have a polynomial-time solving algorithm that can use an efficient algorithm for a problem in C;
- NP^C (co-NP^C, resp.) is the class of problems that have a polynomial-time checking algorithm for their yes-instances (for their no-instances, resp.) that can use an efficient algorithm for a problem in C.

All these classes form the so-called *polynomial hierarchy*.

On the complexity of AP Graph

Notice that AP Graph belongs to the second level of the polynomial hierarchy.

Obs. AP Graph is in Π_2^p.

Indeed, one can design an algorithm that, provided the graph G and some sequence τ, consults an efficient algorithm dealing with RealSeq to make sure that τ is not realizable in G. We thus have a polynomial-time checking algorithm for no-instances of AP Graph based on a *NP* problem.

On the complexity of AP Graph

Notice that AP Graph belongs to the second level of the polynomial hierarchy.

Obs. AP Graph is in Π_2^p.

Indeed, one can design an algorithm that, provided the graph G and some sequence τ, consults an efficient algorithm dealing with RealSeq to make sure that τ is not realizable in G. We thus have a polynomial-time checking algorithm for no-instances of AP Graph based on a *NP* problem.

Inspired by the relationship between the *NP* and *NP*-complete complexity classes, the notion of hardest problems for any class of the polynomial hierarchy was also introduced. The main complete problem in Π_2^p is the following one.

$\forall\exists$SAT
Instance: A CNF formula F over variables $X \cup Y$.
Question: For every truth assignation to the variables in X, does there exist a truth assignation to the variables in Y such that F is satisfied?

Although the structure of the AP Graph problem fits the Π_2^p class well, we did not manage to design a reduction from $\forall\exists$SAT to AP Graph so far.

Qst. Is AP Graph complete in Π_2^p?

An example of Π_2^p-complete graph partition problem

What seems difficult is that, to represent the structure of a given formula as a graph, the latter has to have a strong structure. But then we are sure that some sequences will never be realizable in the resulting graph - just like in the previous reduction from 1-IN-3 SAT to RealSeq.

An example of Π_2^p-complete graph partition problem

What seems difficult is that, to represent the structure of a given formula as a graph, the latter has to have a strong structure. But then we are sure that some sequences will never be realizable in the resulting graph - just like in the previous reduction from 1-IN-3 SAT to REALSEQ.

Maybe the formulation of the AP GRAPH problem is actually not so representative of the one of more common Π_2^p problems. For example, the following problem looks like a Π_2^p problem much more.

> DYNAMIC REALIZABLE SEQUENCE - DYNREALSEQ
> Instance: A graph G, a sequence τ and some partial realizations $P_1, ..., P_r$ of τ in G.
> Question: For every partial realization P_i of τ in G, is it possible to extend it so that we get a realization of τ in G?

One can prove that DYNREALSEQ is actually a Π_2^p-complete problem.

> Thm. *B. - 2012*
> DYNREALSEQ is complete in Π_2^p.

DynRealSeq is complete in Π_2^p

> Thm. *B. - 2012*
> DynRealSeq is complete in Π_2^p.

Sketch of the proof.

- To prove that DynRealSeq is in Π_2^p, consider a checking algorithm for no-instances based on the RealSeq *NP* problem.

DynRealSeq is complete in Π_2^p

Thm. *B. - 2012*
DynRealSeq is complete in Π_2^p.

Sketch of the proof.

- To prove that DynRealSeq is in Π_2^p, consider a checking algorithm for no-instances based on the RealSeq *NP* problem.
- To show that this problem is complete in Π_2^p, one can consider a 1-in-3 version of $\forall\exists$SAT. Consider next the same reduction as the one we designed from 1-in-3 SAT to RealSeq except that the resulting sequence τ contains some small elements instead of the big one with size $n_1 + n$. Finally, for every possible truth assignation to the variables in X, add a corresponding partial realization to the instance.

■

What about a generalized version of PrescSeq?

We previously considered the RealSeq and PrescSeq problems and then only considered a generalized version of RealSeq. But what about a generalization of PrescSeq?

One may consider the following graph property introduced during previous works.

> Def. *AP+k graph*
> A graph G is *arbitrarily partitionable under k-prescriptions* if it can be partitioned following every sequence that sums up to $|V(G)|$ and under every k-prescription.

A quite natural decision problem then arises from the definition above.

> AP+k Graph
> Instance: A graph G.
> Question: Is G an AP+k graph?

Since we proved that PrescSeq is in *NP*, it follows that AP+k Graph is in Π_2^p for every k. But once again, we do not know whether this problem is complete in Π_2^p.

> Qst. Is AP+k Graph complete in Π_2^p?

A last graph partition problem that lies in the polynomial hierarchy...

If a graph has a spanning AP tree, then it is AP. But there exist some AP graphs that cannot be obtained by adding edges to some AP trees. Thus, we formerly interested ourselves in the following property.

> Def. *min-AP graph*
> A *minimal arbitrarily partitionable graph* is an AP graph that does not admit an AP partial subgraph.

Once again, we can derive a new decision problem from this definition.

> Min-AP Graph
> Instance: An AP graph G.
> Question: Is G a min-AP graph?

For every edge e of G, we can provide a sequence that is not realizable in $G - \{e\}$ to a checking algorithm for yes-instances. Thus, Min-AP Graph is in Σ_2^p thanks to the RealSeq *NP* problem.

> Qst. Is Min-AP Graph complete in Σ_2^p?

Part 1: Catching how hard is a decision problem

Part 2: Partitioning a graph following a sequence

Part 3: On some generalized graph partition problems

Part 4: A recursive version of the graph partition problem

When time is not enough

Some problems cannot be solved or checked within a reasonable amount of time like the previous problems we introduced. Such problems are then rather classified regarding the amount of *space* needed by an algorithm to solve them.

When time is not enough

Some problems cannot be solved or checked within a reasonable amount of time like the previous problems we introduced. Such problems are then rather classified regarding the amount of *space* needed by an algorithm to solve them.

As an illustration, let us consider the following procedure working on some graph G. Some integer n_1 comes in and we are asked to pick up a part V_1 of vertices of G that induces a connected subgraph of G on n_1 vertices. Then, a second positive integer n_2 is provided and we are requested a part V_2 of vertices in $G - V_1$ inducing a connected subgraph of G on n_2 vertices. And so on until G has some remaining vertices.

Def. *OL-AP graph*

If we can achieve the procedure above for every sequence of incoming positive integers, then G is *on-line arbitrarily partitionable*.

An alternative definition is the following.

Obs. G is OL-AP iff one the following holds:

- it is isomorphic to K_1;
- it is connected and for every positive integer $\lambda \in [1, |V(G)| - 1]$ we can find a subset $V_\lambda \subset V(G)$ such that $G[V_\lambda]$ is a connected graph on λ vertices and $G - V_\lambda$ is OL-AP.

When time is not enough

The following decision problem then comes to mind naturally.

> OL-AP Graph
> Instance: A graph G.
> Question: Is G an OL-AP graph?

When time is not enough

The following decision problem then comes to mind naturally.

> OL-AP Graph
> Instance: A graph G.
> Question: Is G an OL-AP graph?

This problem does not seem to belong to the polynomial hierarchy:

- a yes-instance checker has to consider *every* possible sequences of incoming integers;
- a no-instance checker has to consider *every* possible way to take a part with size n_i.

When time is not enough

The following decision problem then comes to mind naturally.

> OL-AP Graph
> Instance: A graph G.
> Question: Is G an OL-AP graph?

This problem does not seem to belong to the polynomial hierarchy:

- a yes-instance checker has to consider *every* possible sequences of incoming integers;
- a no-instance checker has to consider *every* possible way to take a part with size n_i.

Hence, we now consider space resources to characterize this problem. A problem belongs to the *PSPACE* class if there exists a solving algorithm for it that uses a polynomial amount of space regarding an instance size.

Since a polynomial-time algorithm can only use a polynomial amount of space, it follows that the polynomial hierarchy is included in *PSPACE*.

A polynomial-space algorithm for OL-AP Graph

Algorithm 1: isOLAP(G: Graph): boolean

```
if G ≃ K_1 then
    return TRUE;
else if G is not connected then
    return FALSE;
else
    foreach λ ∈ {1, ..., n − 1} do
        foreach subset V_λ ⊂ V(G) with size λ do
            if G[V_λ] is not connected then
                Next V_λ;
            else if isOLAP(G − V_λ) then
                Next λ;
            end
        end
        return FALSE;
    end
    return TRUE;
end
```

On the complexity of OL-AP Graph

For a given value of λ, all the V_λ subsets can be generated thanks to a binary array with size $|V(G)|$. Since the tree of the recursive calls of this algorithm has depth $|V(G)|$, then we only need $|V(G)|$ such arrays to cover all the recursive calls. Thus, this algorithm only uses quadratic space.

On the complexity of OL-AP Graph

For a given value of λ, all the V_λ subsets can be generated thanks to a binary array with size $|V(G)|$. Since the tree of the recursive calls of this algorithm has depth $|V(G)|$, then we only need $|V(G)|$ such arrays to cover all the recursive calls. Thus, this algorithm only uses quadratic space.

The notion of hardest problems was also defined for the *PSPACE* class. However, for the same reasons as the ones we mentioned about the completeness of AP Graph for Π_2^p, we do not know if the following holds.

Qst. Is OL-AP Graph complete in *PSPACE*?

On the complexity of OL-AP Graph

For a given value of λ, all the V_λ subsets can be generated thanks to a binary array with size $|V(G)|$. Since the tree of the recursive calls of this algorithm has depth $|V(G)|$, then we only need $|V(G)|$ such arrays to cover all the recursive calls. Thus, this algorithm only uses quadratic space.

The notion of hardest problems was also defined for the *PSPACE* class. However, for the same reasons as the ones we mentioned about the completeness of AP Graph for Π_2^p, we do not know if the following holds.

Qst. Is OL-AP Graph complete in *PSPACE*?

But using once again the reduction we saw earlier, we can show that if the possible part sizes for each level of the procedure are considered as input parameters, then we are dealing with a *PSPACE*-complete problem...

Thank you for your attention!

References

D. Barth and H. Fournier.
A degree bound on decomposable trees.
Discret. Math., 306(5):469–477, 2006.

M.E. Dyer and A.M. Frieze.
On the complexity of partitioning graphs into connected subgraphs.
Discret. Appl. Math., 10:139–153, 1985.