Beyond NP-Completeness

Stepan Kuznetsov

Discrete Math Bridging Course, HSE University

Euler and Hamiltonian Paths

- An **Euler path** in a (multi)graph is a path which traverses each **edge** exactly once.

Euler and Hamiltonian Paths

- An **Euler path** in a (multi)graph is a path which traverses each **edge** exactly once.
- A **Hamiltonian path** should traverse each **vertex** exactly once.

Euler and Hamiltonian Paths

- An **Euler path** in a (multi)graph is a path which traverses each **edge** exactly once.
- A **Hamiltonian path** should traverse each **vertex** exactly once.
- The two notions look similar, but there is a complexity gap: finding an Euler path is polynomial, while existence of a Hamiltonian one is NP-complete.

Euler and Hamiltonian Paths

- Finding a Hamiltonian cycle, in general, is hard.

Euler and Hamiltonian Paths

- Finding a Hamiltonian cycle, in general, is hard.
- Finding a Euler cycle is easy (can be done in polynomial time).

Euler and Hamiltonian Paths

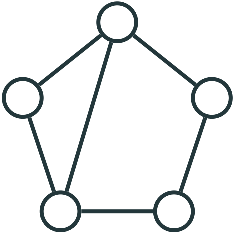
- Finding a Hamiltonian cycle, in general, is hard.
- Finding a Euler cycle is easy (can be done in polynomial time).
- What if, for a specific class of graphs, the problem of finding a Hamiltonian cycle could be *reduced* to the problem of finding a Euler cycle?

Line Graphs

- For a graph G, we define its line graph $L(G)$, as follows:

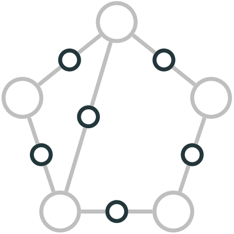
Line Graphs

- For a graph G, we define its **line graph** $L(G)$, as follows:
  - vertices of $L(G)$ are *edges* of G;



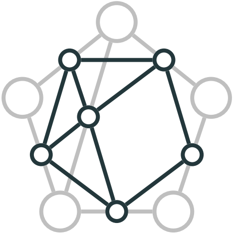
Line Graphs

- For a graph G, we define its **line graph** $L(G)$, as follows:
  - vertices of $L(G)$ are *edges* of G;
  - two vertices are connected in $L(G)$, if the corresponding edges of G have a common end.



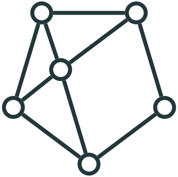
Line Graphs

- For a graph G, we define its **line graph** $L(G)$, as follows:
  - vertices of $L(G)$ are *edges* of G;
  - two vertices are connected in $L(G)$, if the corresponding edges of G have a common end.



Line Graphs

- For a graph G, we define its **line graph** $L(G)$, as follows:
  - vertices of $L(G)$ are *edges* of G;
  - two vertices are connected in $L(G)$, if the corresponding edges of G have a common end.



Line Graphs and Paths

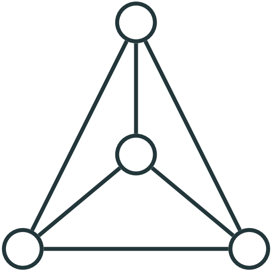
- A Euler path in G induces a Hamiltonian path in $L(G)$.

Line Graphs and Paths

- A Euler path in G induces a Hamiltonian path in $L(G)$.
- The converse, however, does not hold: in $L(G)$, there could be a Hamiltonian cycle, which is not induced by a Euler cycle in G.

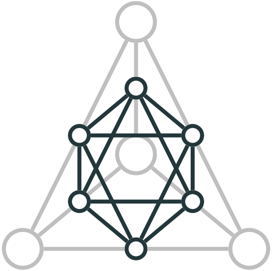
Line Graphs and Paths

- A Euler path in G induces a Hamiltonian path in $L(G)$.
- The converse, however, does not hold: in $L(G)$, there could be a Hamiltonian cycle, which is not induced by a Euler cycle in G. Example:



Line Graphs and Paths

- A Euler path in G induces a Hamiltonian path in $L(G)$.
- The converse, however, does not hold: in $L(G)$, there could be a Hamiltonian cycle, which is not induced by a Euler cycle in G. Example:



Line Graphs and Paths

- And, of course, not every Hamiltonian graph is a line graph of some other graph G.

Line Graphs and Paths

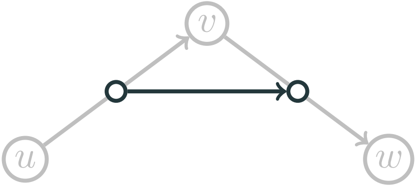
- And, of course, not every Hamiltonian graph is a line graph of some other graph G.
- Nevertheless, in some practically important cases representation of a given graph as $L(G)$ allows efficient construction of Hamiltonian cycles.

Directed Line Graphs

- The line graph $L(G)$ can be also defined for the case of directed G.

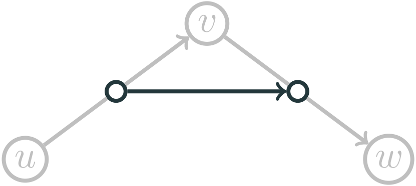
Directed Line Graphs

- The line graph $L(G)$ can be also defined for the case of directed G.
- Vertices of $L(G)$ are directed edges of G, and we connect $\langle u, v\rangle$ with $\langle v, w\rangle$, in the given direction:



Directed Line Graphs

- The line graph $L(G)$ can be also defined for the case of directed G.
- Vertices of $L(G)$ are directed edges of G, and we connect $\langle u, v\rangle$ with $\langle v, w\rangle$, in the given direction:



- Again, a directed Euler path in G induces a directed Hamiltonian path in $L(G)$.

Application: Genome and Its Fragments

- The *genome* is, roughly a string of letters A, C, G, T (they encode *nucleotides:* adenine, cytosine, guanine, thymine).

ACTAGCTGCC

Application: Genome and Its Fragments

- The *genome* is, roughly a string of letters A, C, G, T (they encode *nucleotides:* adenine, cytosine, guanine, thymine).

  ACTAGCTGCC

- Consider the following model situation. Experiment does not give us the complete genome, but rather all its fragments of length 3, in a random order:

  TGC, CTA, GCT, AGC, ACT, GCC, TAG, CTG

Application: Genome and Its Fragments

- The *genome* is, roughly a string of letters A, C, G, T (they encode *nucleotides:* adenine, cytosine, guanine, thymine).

  ACTAGCTGCC

- Consider the following model situation. Experiment does not give us the complete genome, but rather all its fragments of length 3, in a random order:

  TGC, CTA, GCT, AGC, ACT, GCC, TAG, CTG

- Our goal is to reassemble the genome.

Reassembly as Hamiltonian Path

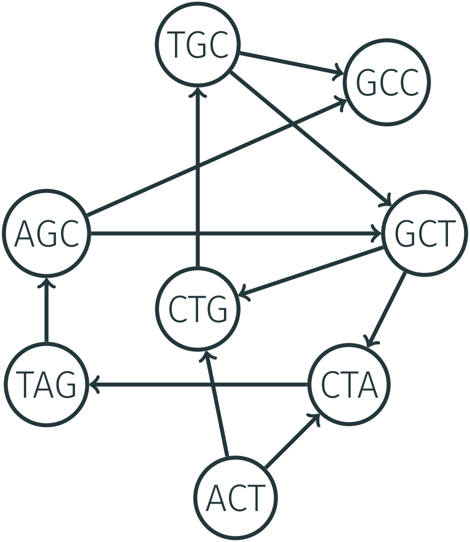
- It is easy to see that the reassembly corresponds to a Hamiltonian path in the *overlap graph*.

Reassembly as Hamiltonian Path

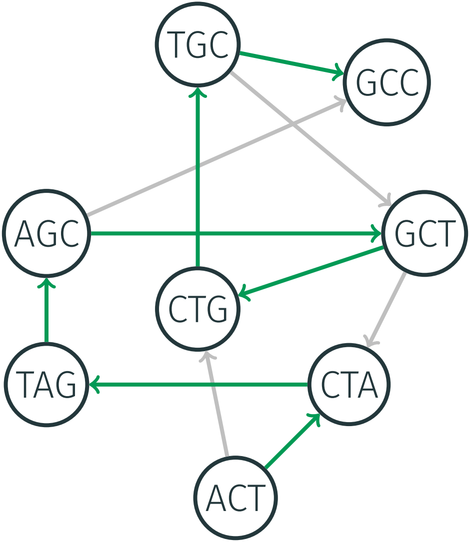
- It is easy to see that the reassembly corresponds to a Hamiltonian path in the *overlap graph*.
- In this graph, triplet u is connected to triplet v, if the last two letters of u are the first two letters of v:

$$\text{CTA} \rightarrow \text{TAG}.$$

Reassembly as Hamiltonian Path



Reassembly as Hamiltonian Path



Reassembly as Hamiltonian Path

- We reduced the reassembly problem to the Hamiltonian path problem.

Reassembly as Hamiltonian Path

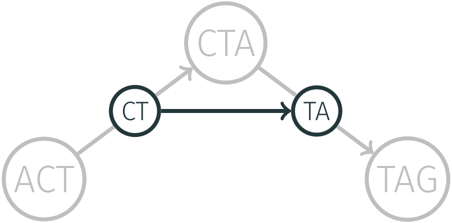
- We reduced the reassembly problem to the Hamiltonian path problem.
- Unfortunately, the latter is hard.

Reassembly as Hamiltonian Path

- We reduced the reassembly problem to the Hamiltonian path problem.
- Unfortunately, the latter is hard.
- It would be much better if we could use Euler path instead.

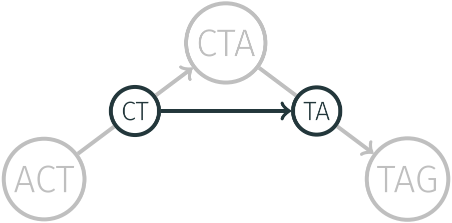
De Bruijn Graph

- Fortunately, the overlap graph is $L(G)$ for some other graph G:



De Bruijn Graph

- Fortunately, the overlap graph is $L(G)$ for some other graph G:

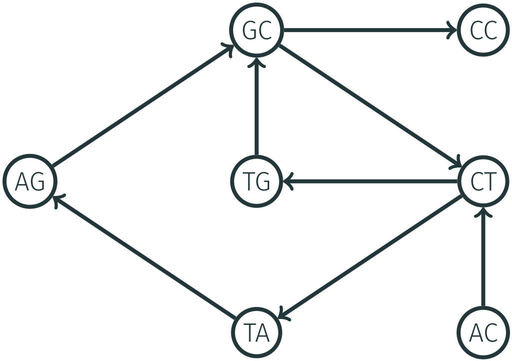


- After identifying vertices with the same annotation in G and adding AC and CC (start and end), we get **de Bruijn graph.**

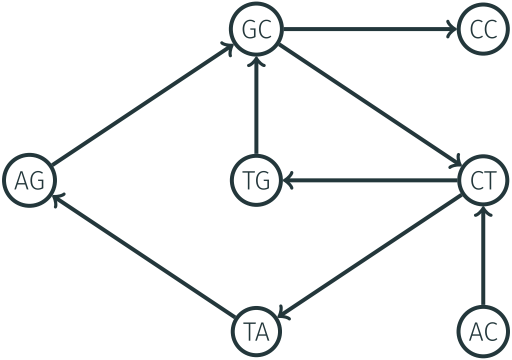
De Bruijn Graph

A Euler path in de Bruijn graph induces a Hamiltonian path in overlap graph.

Reassembly as Euler Path



Reassembly as Euler Path



Two possible ways to reassemble: ACTAGCTGCC and ACTGCTAGCC.

Overlap vs. de Bruijn

- This is an example how discovering the inner structure of a graph helps making problems algorithmically simpler.

Overlap vs. de Bruijn

- This is an example how discovering the inner structure of a graph helps making problems algorithmically simpler.
- De Bruijn graph is used in real-world genome assemblers.

The NP Class

- Let us recall the NP class.

The NP Class

- Let us recall the NP class.
- Today we shall use Definition 2, with *hints*.

The NP Class

- Let us recall the NP class.
- Today we shall use Definition 2, with *hints*.
- Denote the decision problem by $A(x)$.

$$A(x) = 1 \iff \exists y \, (|y| < q(|x|) \,\&\, R(x, y) = 1),$$

where $R \in P$.

The NP Class

- Let us recall the NP class.
- Today we shall use Definition 2, with *hints*.
- Denote the decision problem by $A(x)$.

  $$A(x) = 1 \iff \exists y\,(|y| < q(|x|)\,\&\,R(x, y) = 1),$$

  where $R \in P$.
- Let us check $|y| < q(|x|)$ inside R.

The NP Class

- Let us recall the NP class.
- Today we shall use Definition 2, with *hints*.
- Denote the decision problem by $A(x)$.

  $$A(x) = 1 \iff \exists y \, (|y| < q(|x|) \,\&\, R(x, y) = 1),$$

  where $R \in P$.
- Let us check $|y| < q(|x|)$ inside R.
- y is a *hint*, given by someone to help us solve the problem.

The NP Class

- Let us recall the NP class.
- Today we shall use Definition 2, with *hints*.
- Denote the decision problem by $A(x)$.

$$A(x) = 1 \iff \exists y\,(|y| < q(|x|)\,\&\,R(x,y) = 1),$$

  where $R \in P$.
- Let us check $|y| < q(|x|)$ inside R.
- y is a *hint*, given by someone to help us solve the problem.
- Examples of y: the satisfying assignment; the Hamiltonian cycle; …

Beyond Decision Problems

- An NP **decision** problem is the question **whether there exists** a *witness* y such that $R(x, y) = 1$.

Beyond Decision Problems

- An NP **decision** problem is the question **whether there exists** a *witness* y such that $R(x, y) = 1$.
    - E.g., a satisfying assignment for φ.

Beyond Decision Problems

- An NP **decision** problem is the question **whether there exists** a *witness* y such that $R(x, y) = 1$.
  - E.g., a satisfying assignment for φ.
- We could ask for **all** witnesses, and the algorithm can yield them with **polynomial delay.**

Beyond Decision Problems

- An NP **decision** problem is the question **whether there exists** a *witness* y such that $R(x, y) = 1$.
    - E.g., a satisfying assignment for φ.
- We could ask for **all** witnesses, and the algorithm can yield them with **polynomial delay.**
- **Search problem:** yield a witness or say "no."

Beyond Decision Problems

- An NP **decision** problem is the question **whether there exists** a *witness* y such that $R(x, y) = 1$.
  - E.g., a satisfying assignment for φ.
- We could ask for **all** witnesses, and the algorithm can yield them with **polynomial delay.**
- **Search problem:** yield a witness or say "no."
- **Counting problem** (the #P class): yield the **number** of witnesses.

Beyond Decision Problems

- *A priori*, the decision problem is the easiest one.

Beyond Decision Problems

- *A priori*, the decision problem is the easiest one.
- Indeed, if we can solve the search problem or the counting problem, then we automatically get a solution for the decision problem (with the same R).

Beyond Decision Problems

- *A priori*, the decision problem is the easiest one.
- Indeed, if we can solve the search problem or the counting problem, then we automatically get a solution for the decision problem (with the same R).
- However, search problems are also not harder than decision ones.

Beyond Decision Problems

- *A priori*, the decision problem is the easiest one.
- Indeed, if we can solve the search problem or the counting problem, then we automatically get a solution for the decision problem (with the same R).
- However, search problems are also not harder than decision ones.
- Namely, if $\mathrm{P} = \mathrm{NP}$, then any search problem is also solvable in polynomial time.

Beyond Decision Problems

- *A priori*, the decision problem is the easiest one.
- Indeed, if we can solve the search problem or the counting problem, then we automatically get a solution for the decision problem (with the same R).
- However, search problems are also not harder than decision ones.
- Namely, if $\mathrm{P} = \mathrm{NP}$, then any search problem is also solvable in polynomial time.
  - E.g., searching for SAT can be done via dichotomy using decision for SAT.

Search Problems

- However, it is not true that the search problem is always reduced to **the same** decision problem.

Search Problems

- However, it is not true that the search problem is always reduced to **the same** decision problem.
- For example, let $R(\varphi, y)$ mean “$y = (a, b)$, where a is a satisfying assignment for φ or b is a satisfying assignment for $\neg\varphi$.“

Search Problems

- However, it is not true that the search problem is always reduced to **the same** decision problem.
- For example, let $R(\varphi, y)$ mean "$y = (a, b)$, where a is a satisfying assignment for φ or b is a satisfying assignment for $\neg\varphi$."
- Here the decision problem is trivial (always "yes"), but the search problem is equivalent to the one for SAT.

Counting Problems

- #P is the class of counting problems corresponding to NP decision problems.

Counting Problems

- #P is the class of counting problems corresponding to NP decision problems.
- Counting problems can be harder than the corresponding decision ones!

Counting Problems

- #P is the class of counting problems corresponding to NP decision problems.
- Counting problems can be harder than the corresponding decision ones!

Theorem

$\#2$-SAT is not solvable in polynomial time, unless P $=$ NP (while 2-SAT as a decision problem belongs to P).

Counting Problems

- #P is the class of counting problems corresponding to NP decision problems.
- Counting problems can be harder than the corresponding decision ones!

Theorem

#2-SAT is not solvable in polynomial time, unless $P = NP$ (while 2-SAT as a decision problem belongs to P).

- In order to prove theorems like this one, one has to develop the theory of #P-completeness.

Counting Reductions

- As the theory of NP-completeness is based on polynomial m-reductions (denoted by $A \leq_m^P B$), the theory of #P-completeness is based on counting reductions: $\#A \leq_c^P \#B$.

Counting Reductions

- As the theory of NP-completeness is based on polynomial m-reductions (denoted by $A \leq_m^P B$), the theory of #P-completeness is based on counting reductions: $\#A \leq_c^P \#B$.
- A counting reduction consists of **two** functions, $f\colon \Sigma^* \to \Sigma^*$ on input data and $g\colon \mathbb{N} \to \mathbb{N}$ on counts (results).

Counting Reductions

- As the theory of NP-completeness is based on polynomial m-reductions (denoted by $A \leq_m^P B$), the theory of #P-completeness is based on counting reductions: $\#A \leq_c^P \#B$.
- A counting reduction consists of **two** functions, $f\colon \Sigma^* \to \Sigma^*$ on input data and $g\colon \mathbb{N} \to \mathbb{N}$ on counts (results).
- Recall that $\#A$ and $\#B$ are counting problems, that is,

$$\#A(x) = |\{y \mid R(x,y) = 1\}| \in \mathbb{N},$$

and the same for $\#B$.

Counting Reductions

- We say that $\#A \leq_c^P \#B$, if there exists a pair of polynomially computable reducing functions f and g such that for any input x we have

$$\#A(x) = g(\#B(f(x))).$$

- This indeed allows to **reduce** $\#A$ to $\#B$. Suppose we know how to solve $\#B$. Then, in order to solve $\#A$, we take x, apply f, then solve $\#B$ (yielding a natural number) and apply g.

#P-Completeness

- A counting problem $\#B$ is #P-complete, if for any other $\#A \in \#$P we have $\#A \leq_c^P \#B$...
- ... just as for NP-completeness.
- Now we can develop a theory of #P-complete problems, which is parallel to the theory of NP-completeness.

Parsimonious Reductions

- A counting reduction (f, g), where g is identity, $g(n) = n$, is called a **parsimonious** reduction.
- A parsimonious reduction is also a specific kind of m-reduction, since, in particular, $g(0) = 0$, thus, it conveys the answer to the decision problem.

Parsimonious Reductions

- The reductions in Cook–Levin theorem are parsimonious.

Parsimonious Reductions

- The reductions in Cook–Levin theorem are parsimonious.
    - Indeed, each trajectory of the non-deterministic run (that is, each value of hint y) is represented by exactly one satisfying assignment.

Parsimonious Reductions

- The reductions in Cook–Levin theorem are parsimonious.
  - Indeed, each trajectory of the non-deterministic run (that is, each value of hint y) is represented by exactly one satisfying assignment.
- This yields the counting version of Cook–Levin:

Parsimonious Reductions

- The reductions in Cook–Levin theorem are parsimonious.
  - Indeed, each trajectory of the non-deterministic run (that is, each value of hint y) is represented by exactly one satisfying assignment.
- This yields the counting version of Cook–Levin:

Theorem

#SAT is #P-complete.

Cook – Levin Theorem

- The sequence of configurations (protocol) of A on input x is encoded by a binary matrix (b_{ij}) of size $(m \cdot p(|x|)) \times p(|x|)$.

Cook – Levin Theorem

- The sequence of configurations (protocol) of A on input x is encoded by a binary matrix (b_{ij}) of size $(m \cdot p(|x|)) \times p(|x|)$.
- Next, we construct a formula φ_x with variables $b_{00}, b_{01}, \ldots$ which expresses the fact that this matrix represents a correct protocol of a successful execution.

Cook – Levin Theorem

φ_x is a conjunction of the following claims:

1. the first row represents the configuration with x on the tape, the machine observing its first letter;
2. each next row is obtained from the previous one by one of the rules of the machine;
3. the last row includes state q_F and the answer “yes” (1).

Cook – Levin Theorem

φ_x is a conjunction of the following claims:

1. the first row represents the configuration with x on the tape, the machine observing its first letter;
2. each next row is obtained from the previous one by one of the rules of the machine;
3. the last row includes state q_F and the answer “yes” (1).

This is all expressible as Boolean formulae.

Parsimonious Reductions

- Tseitin’s transformations are also parsimonious.

Parsimonious Reductions

- Tseitin's transformations are also parsimonious.
- That is, any Boolean formula φ can be translated into a 3-CNF ψ, such satisfying assignments of ψ are in one-to-one correspondence with those for φ.

Parsimonious Reductions

- Tseitin's transformations are also parsimonious.
- That is, any Boolean formula φ can be translated into a 3-CNF ψ, such satisfying assignments of ψ are in one-to-one correspondence with those for φ.
    - Values for new variables t_i are restored uniquely.

Parsimonious Reductions

- Tseitin's transformations are also parsimonious.
- That is, any Boolean formula φ can be translated into a 3-CNF ψ, such satisfying assignments of ψ are in one-to-one correspondence with those for φ.
    - Values for new variables t_i are restored uniquely.
- Thus, #3-SAT is also #P-complete.

Tseitin’s Transformations

Theorem

For any Boolean formula φ, there exists an equisatisfiable 3-CNF ψ of polynomial size.

- Equisatisfiability means that ψ is satisfiable iff so is φ.

Tseitin's Transformations

Theorem

For any Boolean formula φ, there exists an equisatisfiable 3-CNF ψ of polynomial size.

- Equisatisfiability means that ψ is satisfiable iff so is φ.
- Constructing an *equivalent* 3-CNF of polynomial size is not always possible: even translation to CNF can lead to exponential blowup.

Tseitin's Transformations

- Tseitin's transformations look like translation into 3-address (Assembler-like) code:
  $(a+b)*(c+d)$ is translated to
  "add a b t_1; add c d t_2; mul t_1 t_2 r"

Tseitin's Transformations

- Tseitin's transformations look like translation into 3-address (Assembler-like) code:
  $(a + b) * (c + d)$ is translated to
  "add $a\ b\ t_1$; add $c\ d\ t_2$; mul $t_1\ t_2\ r$"
- For each subformula we introduce a new variable and write the corresponding equivalences.

Tseitin's Transformations

Example: $(p \to q) \vee (q \to (p \to r))$

Tseitin's Transformations

Example: $(p \rightarrow q) \vee (q \rightarrow (p \rightarrow r))$

$$\begin{aligned}
&(t_1 \leftrightarrow (p \rightarrow q)) \wedge \\
&(t_2 \leftrightarrow (p \rightarrow r)) \wedge \\
&(t_3 \leftrightarrow (q \rightarrow t_2)) \wedge \\
&(t_4 \leftrightarrow (t_1 \vee t_3)) \wedge \\
&t_4
\end{aligned}$$

Tseitin's Transformations

Transform into 3-CNF by the following table:

$$
\begin{array}{l|l}
t_k \leftrightarrow (t_i \wedge t_j) & (\neg t_i \vee \neg t_j \vee t_k) \wedge (t_i \vee \neg t_k) \wedge (t_j \vee \neg t_k) \\
t_k \leftrightarrow (t_i \vee t_j) & (t_i \vee t_j \vee \neg t_k) \wedge (\neg t_i \vee t_k) \wedge (\neg t_j \vee t_k) \\
t_k \leftrightarrow (t_i \rightarrow t_j) & (\neg t_i \vee t_j \vee \neg t_k) \wedge (t_i \vee t_k) \wedge (\neg t_j \vee t_k) \\
t_k \leftrightarrow \neg t_i & (t_i \vee t_k) \wedge (\neg t_i \vee \neg t_k)
\end{array}
$$

Tseitin's Transformations

Transform into 3-CNF by the following table:

$t_k \leftrightarrow (t_i \wedge t_j)$	$(\neg t_i \vee \neg t_j \vee t_k) \wedge (t_i \vee \neg t_k) \wedge (t_j \vee \neg t_k)$
$t_k \leftrightarrow (t_i \vee t_j)$	$(t_i \vee t_j \vee \neg t_k) \wedge (\neg t_i \vee t_k) \wedge (\neg t_j \vee t_k)$
$t_k \leftrightarrow (t_i \rightarrow t_j)$	$(\neg t_i \vee t_j \vee \neg t_k) \wedge (t_i \vee t_k) \wedge (\neg t_j \vee t_k)$
$t_k \leftrightarrow \neg t_i$	$(t_i \vee t_k) \wedge (\neg t_i \vee \neg t_k)$

For our example, we get:

$$(\neg p \vee q \vee \neg t_1) \wedge (p \vee t_1) \wedge (\neg q \vee t_1) \wedge$$
$$(\neg p \vee r \vee \neg t_2) \wedge (p \vee t_2) \wedge (\neg r \vee t_2) \wedge$$
$$(\neg q \vee t_2 \vee \neg t_3) \wedge (q \vee t_3) \wedge (\neg t_2 \vee t_3) \wedge$$
$$(t_1 \vee t_3 \vee \neg t_4) \wedge (\neg t_1 \vee t_4) \wedge (\neg t_3 \vee t_4) \wedge t_4$$

Beyond Parsimonious Reductions

- Using **only** parsimonious reductions for establishing #P-completeness is meaningless.

Beyond Parsimonious Reductions

- Using **only** parsimonious reductions for establishing #P-completeness is meaningless.
- Indeed, if a counting problem $\#A$ is proven #P-complete by parsimonious reductions, then its decision variant A is NP-complete.

Beyond Parsimonious Reductions

- Using **only** parsimonious reductions for establishing #P-completeness is meaningless.
- Indeed, if a counting problem $\#A$ is proven #P-complete by parsimonious reductions, then its decision variant A is NP-complete.
- In this case, if P $\neq$ NP, we know that even A is not polynomially solvable, nothing to say about $\#A$.

Beyond Parsimonious Reductions

- Using **only** parsimonious reductions for establishing #P-completeness is meaningless.
- Indeed, if a counting problem $\#A$ is proven #P-complete by parsimonious reductions, then its decision variant A is NP-complete.
- In this case, if P $\neq$ NP, we know that even A is not polynomially solvable, nothing to say about $\#A$.
- Using more general counting reductions, however, could give interesting results.

$A \in$ P, $\#A$ #P-complete

- Interesting cases include situations when the decision problem is polynomially decidable, while the counting problem is hard.

$A \in$ P, $\#A$ #P-complete

- Interesting cases include situations when the decision problem is polynomially decidable, while the counting problem is hard.
- The famous example is 2-SAT.

$A \in$ P, $\#A$ #P-complete

- Interesting cases include situations when the decision problem is polynomially decidable, while the counting problem is hard.
- The famous example is 2-SAT.
    - We know that 2-SAT $\in$ P.

$A \in$ P, $\#A$ #P-complete

- Interesting cases include situations when the decision problem is polynomially decidable, while the counting problem is hard.
- The famous example is 2-SAT.
    - We know that 2-SAT $\in$ P.
    - We shall not give the proof of #P-completeness for #2-SAT, since it is technically hard.

$A \in$ P, $\#A$ #P-complete

- Interesting cases include situations when the decision problem is polynomially decidable, while the counting problem is hard.
- The famous example is 2-SAT.
    - We know that 2-SAT $\in$ P.
    - We shall not give the proof of #P-completeness for #2-SAT, since it is technically hard.
    - See A. Ben-Dor, S. Halevi (1993), “Zero-one permanent is #P-complete, a simple proof” and L.G. Valiant (1979), “The complexity of enumeration and reliability problems”.

$A \in$ P, $\#A$ #P-complete

- We shall consider an easier example: DNF-SAT vs. #DNF-SAT.

$A \in$ P, $\#A$ #P-complete

- We shall consider an easier example: DNF-SAT vs. #DNF-SAT.
- Easily, DNF-SAT $\in$ P (as a decision problem).

$A \in$ P, $\#A$ #P-complete

- We shall consider an easier example: DNF-SAT vs. #DNF-SAT.
- Easily, DNF-SAT $\in$ P (as a decision problem).
- However, in the counting case we can reduce from CNF-SAT by duality:

$$f(\varphi) = \mathsf{DNF}(\neg\varphi)$$
$$g(n) = 2^k - n$$

(where k is the number of variables).

$A \in$ P, $\#A$ #P-complete

- Indeed, the set of satisfying assingments for φ is the complement of that for $\neg\varphi$.

$A \in$ P, $\#A$ #P-complete

- Indeed, the set of satisfying assingments for φ is the complement of that for $\neg\varphi$.
- If φ is in CNF, then DNF($\neg\varphi$) is polynomially computable.

$A \in$ P, $\#A$ #P-complete

- Indeed, the set of satisfying assingments for φ is the complement of that for $\neg\varphi$.
- If φ is in CNF, then DNF($\neg\varphi$) is polynomially computable.
- Thus, #CNF-SAT $\leq_c^P$ #DNF-SAT, and therefore #DNF-SAT is #P-complete.

$A \in$ P, $\#A$ #P-complete

- Indeed, the set of satisfying assingments for φ is the complement of that for $\neg\varphi$.
- If φ is in CNF, then DNF($\neg\varphi$) is polynomially computable.
- Thus, #CNF-SAT $\leq_c^P$ #DNF-SAT, and therefore #DNF-SAT is #P-complete.
- **Corollary:** if P $\neq$ NP, then #DNF-SAT is not polynomially solvable.

$A \in$ P, $\#A$ #P-complete

- Indeed, the set of satisfying assingments for φ is the complement of that for $\neg\varphi$.
- If φ is in CNF, then DNF($\neg\varphi$) is polynomially computable.
- Thus, #CNF-SAT $\leq_c^P$ #DNF-SAT, and therefore #DNF-SAT is #P-complete.
- **Corollary:** if P $\neq$ NP, then #DNF-SAT is not polynomially solvable.
- Otherwise so would be #CNF-SAT, and therefore CNF-SAT, which implies P $=$ NP.

Permanent

- And now let us see how #P-completeness arises in a completely different area.

Permanent

- And now let us see how #P-completeness arises in a completely different area.
- The **determinant** of a matrix is a well-known notion in linear algebra:

$$\det(a_{i,j}) = \sum_{\sigma \in \mathbf{S}_n} (-1)^{\operatorname{sign}(\sigma)} a_{1,\sigma(1)} \cdot \ldots \cdot a_{n,\sigma(n)}$$

Permanent

- And now let us see how #P-completeness arises in a completely different area.
- The **determinant** of a matrix is a well-known notion in linear algebra:

$$\det(a_{i,j}) = \sum_{\sigma \in \mathbf{S}_n} (-1)^{\mathrm{sign}(\sigma)} a_{1,\sigma(1)} \cdot \ldots \cdot a_{n,\sigma(n)}$$

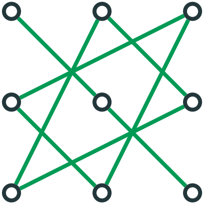
- There exist fast algorithms for computing the determinant, not by its definition (e.g., Gauss' diagonalization).

Permanent

- In the definition of determinant, products are taken with different signs.

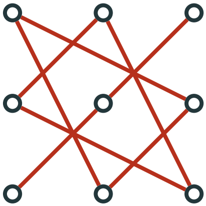
Permanent

- In the definition of determinant, products are taken with different signs.



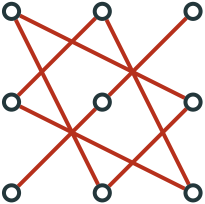
Permanent

- In the definition of determinant, products are taken with different signs.



Permanent

- In the definition of determinant, products are taken with different signs.



- The **permanent** is like the determinant, but without signs:

$$\operatorname{perm}(a_{i,j}) = \sum_{\sigma \in \mathbf{S}_n} a_{1,\sigma(1)} \cdot \ldots \cdot a_{n,\sigma(n)}$$

Permanent

- Permanent is also useful in linear algebra and its application to data analysis.

Permanent

- Permanent is also useful in linear algebra and its application to data analysis.
- One example: computing the normalization constant for Markov random fields is equivalent to computing the permanent.

Permanent

- Permanent is also useful in linear algebra and its application to data analysis.
- One example: computing the normalization constant for Markov random fields is equivalent to computing the permanent.
- However, computing the permanent by definition requires more than exponential time: namely, $n \cdot n!$.

Permanent

- … and, unlike the determinant, for the permanent there is probably no fast algorithm.

Permanent

- … and, unlike the determinant, for the permanent there is probably no fast algorithm.
- This follows from the theory of #P-hardness.

Permanent

- … and, unlike the determinant, for the permanent there is probably no fast algorithm.
- This follows from the theory of #P-hardness.
- Let $a_{i,j}$ be zeroes and ones. Then $\mathrm{perm}(a_{i,j})$ can be seen as a counting problem: how many permutations from $\mathbf{S}_n$ give all ones?

Permanent

- The decision problem here ($\mathrm{perm} > 0$) is easy (polynomial), since it reduces to finding a perfect matching in a bipartite graph.

Permanent

- The decision problem here ($\mathrm{perm} > 0$) is easy (polynomial), since it reduces to finding a perfect matching in a bipartite graph.
- The counting problem (computing perm) is #P-hard (see Valiant 1979).

Permanent

- The decision problem here ($\mathrm{perm} > 0$) is easy (polynomial), since it reduces to finding a perfect matching in a bipartite graph.
- The counting problem (computing perm) is #P-hard (see Valiant 1979).
- This problem is parsimoniously reducible to #2-SAT, so the latter is also #P-hard.