Algorithm Design XIV

NP Problem II

Guoqiang Li
School of Software

Shanghai Jiao Tong University

NP-Completeness

Hard Problems, Easy Problems

Hard problems (NP-complete)	Easy problems (in P)
3SAT	2SAT, HORN SAT
TRAVELING SALESMAN PROBLEM	MINIMUM SPANNING TREE
LONGEST PATH	SHORTEST PATH
3D MATCHING	BIPARTITE MATCHING
KNAPSACK	UNARY KNAPSACK
INDEPENDENT SET	INDEPENDENT SET ON TREES
INTEGER LINEAR PROGRAMMING	LINEAR PROGRAMMING
RUDRATA PATH	EULER PATH
BALANCED CUT	MINIMUM CUT

Recall a search problem is defined by:

NP

Recall a search problem is defined by:

1. An efficient checking algorithm C, taking as input the given `instance` I, a `solution` S, and outputs `true` iff S is a solution I.
2. The running time of $C(I, S)$ is bounded by a polynomial in $|I|$.

NP

Recall a search problem is defined by:

1. An efficient checking algorithm C, taking as input the given `instance` I, a `solution` S, and outputs `true` iff S is a solution I.
2. The running time of $C(I, S)$ is bounded by a polynomial in $|I|$.

We denote the class of all search problems by NP.

P

An algorithm that takes as input an `instance` I and has a running time polynomial in $|I|$.

An algorithm that takes as input an `instance` I and has a running time polynomial in $|I|$.

- I has a solution, the algorithm returns such a solution;
- I has no solution, the algorithm correctly reports so.

P

An algorithm that takes as input an `instance` I and has a running time polynomial in $|I|$.

- I has a solution, the algorithm returns such a solution;
- I has no solution, the algorithm correctly reports so.

The class of all search problems that can be solved in polynomial time is denoted P.

Why P and NP

P: polynomial time

NP: nondeterministic polynomial time

$\mathbf{P} \neq \mathbf{NP}$

Shanghai Jiao Tong University

Theorem Proving

- Input: A mathematical statement φ and n.
- Problem: Find a proof of φ of length $\leq n$ if there is one.

Theorem Proving

- Input: A mathematical statement φ and n.
- Problem: Find a proof of φ of length $\leq n$ if there is one.

A formal proof of a mathematical assertion is written out in excruciating detail, it can be checked mechanically, by an efficient algorithm and is therefore in NP.

Theorem Proving

- Input: A mathematical statement φ and n.
- Problem: Find a proof of φ of length $\leq n$ if there is one.

A formal proof of a mathematical assertion is written out in excruciating detail, it can be checked mechanically, by an efficient algorithm and is therefore in NP.

So if P $=$ NP, there would be an efficient method to prove any theorem, thus eliminating the need for mathematicians!

Solve One and All Solved

Even if we believe $\mathsf{P} \neq \mathsf{NP}$, can we find an evidence that these particular problems have no efficient algorithm?

Solve One and All Solved

Even if we believe $\mathsf{P} \neq \mathsf{NP}$, can we find an evidence that these particular problems have no efficient algorithm?

Such evidence is provided by reductions, which translate one search problem into another.

Solve One and All Solved

Even if we believe P $\neq$ NP, can we find an evidence that these particular problems have no efficient algorithm?

Such evidence is provided by reductions, which translate one search problem into another.

We will show that the hard problems in previous lecture exactly the same problem, the hardest search problems in NP.

Solve One and All Solved

Even if we believe $\mathsf{P} \neq \mathsf{NP}$, can we find an evidence that these particular problems have no efficient algorithm?

Such evidence is provided by reductions, which translate one search problem into another.

We will show that the hard problems in previous lecture exactly the same problem, the hardest search problems in NP.

If one of them has a polynomial time algorithm, then every problem in NP has a polynomial time algorithm.

Reduction Between Search Problems

A reduction from A to B is a polynomial time algorithm f that transforms any instance I of A into an instance $f(I)$ of B

Reduction Between Search Problems

A reduction from A to B is a polynomial time algorithm f that transforms any instance I of A into an instance $f(I)$ of B

Together with another polynomial time algorithm h that maps any solution S of $f(I)$ back into a solution $h(S)$ of I.

Reduction Between Search Problems

A reduction from A to B is a polynomial time algorithm f that transforms any instance I of A into an instance $f(I)$ of B

Together with another polynomial time algorithm h that maps any solution S of $f(I)$ back into a solution $h(S)$ of I.

If $f(I)$ has no solution, then neither does I.

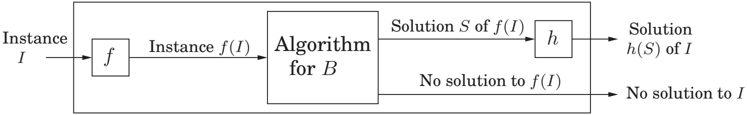
Reduction Between Search Problems

A reduction from A to B is a polynomial time algorithm f that transforms any instance I of A into an instance $f(I)$ of B

Together with another polynomial time algorithm h that maps any solution S of $f(I)$ back into a solution $h(S)$ of I.

If $f(I)$ has no solution, then neither does I.

These two translation procedures f and h imply that any algorithm for B can be converted into an algorithm for A.



The Two Ways to Use Reductions

Assume there is a reduction from a problem A to a problem B.

The Two Ways to Use Reductions

Assume there is a reduction from a problem A to a problem B.

$$A \rightarrow B$$

The Two Ways to Use Reductions

Assume there is a reduction from a problem A to a problem B.

$$A \to B$$

- If we can solve B efficiently, then we can also solve A efficiently.

The Two Ways to Use Reductions

Assume there is a reduction from a problem A to a problem B.

$$A \rightarrow B$$

- If we can solve B efficiently, then we can also solve A efficiently.
- If we know A is hard, then B must be hard too.

The Two Ways to Use Reductions

Assume there is a reduction from a problem A to a problem B.

$$A \to B$$

- If we can solve B efficiently, then we can also solve A efficiently.
- If we know A is hard, then B must be hard too.

If $A \to B$ and $B \to C$, then $A \to C$.

NP-Completeness

Definition

A NP problem is NP-complete if all other NP problems reduce to it.

Reductions to NP-Complete

NP-complete problems are hard: all other search problems reduce to them.

Reductions to NP-Complete

NP-complete problems are hard: all other search problems reduce to them.

For a problem to be NP-complete, it can solve every NP problem in the world.

Reductions to NP-Complete

NP-complete problems are hard: all other search problems reduce to them.

For a problem to be NP-complete, it can solve every NP problem in the world.

If even one NP-complete problem is in P, then P = NP.

Reductions to NP-Complete

NP-complete problems are hard: all other search problems reduce to them.

For a problem to be NP-complete, it can solve every NP problem in the world.

If even one NP-complete problem is in P, then P = NP.

If a problem A is NP-complete, a new NP problem B is proved to be NP-complete, by reducing A to B.

Reduction

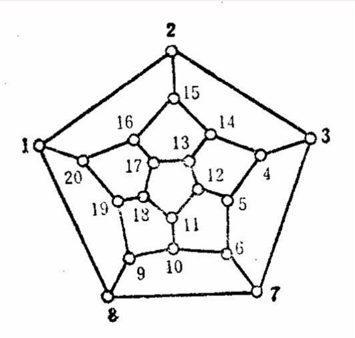
RUDRATA PATH $\rightarrow$ RUDRATA CYCLE

Rudrata Cycle

Shanghai Jiao Tong University

Rudrata Cycle

Given a graph, find a cycle that visits each vertex exactly once.



RUDRATA (s,t)-PATH $\rightarrow$ RUDRATA CYCLE

A RUDRATA (s,t)-PATH problem specifies two vertices s and t and wants a path starting at s and ending at t that goes through each vertex exactly once.

RUDRATA (s,t)-PATH $\to$ RUDRATA CYCLE

A RUDRATA (s,t)-PATH problem specifies two vertices s and t and wants a path starting at s and ending at t that goes through each vertex exactly once.

Q: Is it possible that RUDRATA CYCLE is easier than RUDRATA (s,t)-PATH?

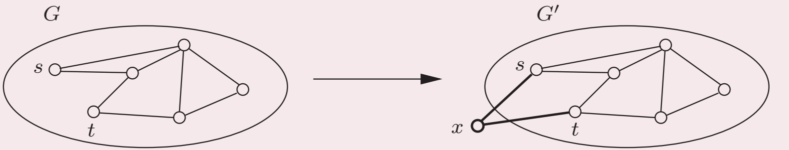
RUDRATA (s, t)-PATH $\rightarrow$ RUDRATA CYCLE

A RUDRATA (s, t)-PATH problem specifies two vertices s and t and wants a path starting at s and ending at t that goes through each vertex exactly once.

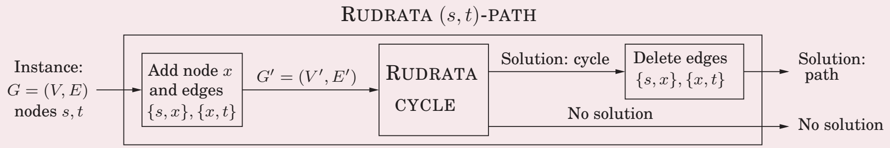
Q: Is it possible that RUDRATA CYCLE is easier than RUDRATA (s, t)-PATH?

The reduction maps an `instance` G of RUDRATA (s, t)-PATH into an `instance` G' of RUDRATA CYCLE as follows: G' is G with an additional vertex x and two new edges $\{s, x\}$ and $\{x, t\}$.

RUDRATA (s,t)-PATH $\rightarrow$ RUDRATA CYCLE



RUDRATA (s, t)-PATH $\to$ RUDRATA CYCLE



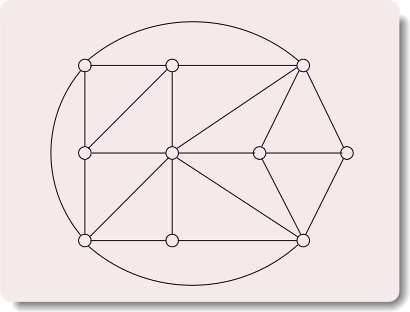
3SAT $\rightarrow$ INDEPENDENT SET

3 SAT

The instances of 3SAT, is set of clauses, each with three or fewer literals.

$$(x \vee y \vee z)(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{x})(\overline{x} \vee \overline{y} \vee \overline{z})$$

Independent Set



INDEPENDENT SET: Given a graph G and an integer g, find g vertices, no two of which have an edge between them.

True Assignment

To form a satisfying truth assignment we must pick one literal from each clause and give it the value `true`.

True Assignment

To form a satisfying truth assignment we must pick one literal from each clause and give it the value `true`.

The choices must be consistent, if we choose $\overline{x}$ in one clause, we cannot choose x in another.

True Assignment

To form a satisfying truth assignment we must pick one literal from each clause and give it the value `true`.

The choices must be consistent, if we choose $\overline{x}$ in one clause, we cannot choose x in another.

Solution: put an edge between any two vertices that correspond to opposite literals.

Clause

Clause

Represent a clause, say $(x \vee \overline{y} \vee z)$, by a triangle, with vertices labeled $x, \overline{y}, z$.

Clause

Represent a clause, say $(x \vee \overline{y} \vee z)$, by a triangle, with vertices labeled $x, \overline{y}, z$.

Because a triangle has its three vertices maximally connected, and thus forces to pick only one of them for the independent set.

Given an `instance` I of 3SAT, create an `instance` (G, g) of INDEPENDENT SET as follows,

3SAT $\rightarrow$ INDEPENDENT SET

Given an `instance` I of 3SAT, create an `instance` (G, g) of INDEPENDENT SET as follows,

- A triangle for each clause, with vertices labeled by the clause's literals.

3SAT $\to$ Independent Set

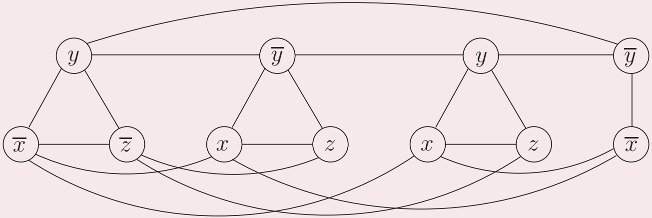
Given an `instance` I of 3SAT, create an `instance` (G, g) of Independent Set as follows,

- A triangle for each clause, with vertices labeled by the clause's literals.
- Additional edges between any two vertices that represent opposite literals.

Given an `instance` I of 3SAT, create an `instance` (G, g) of INDEPENDENT SET as follows,

- A triangle for each clause, with vertices labeled by the clause's literals.
- Additional edges between any two vertices that represent opposite literals.
- The goal g is set to the number of clauses.

3SAT $\to$ INDEPENDENT SET



$$(\overline{x} \vee y \vee \overline{z})(x \vee \overline{y} \vee z)(x \vee y \vee z)(\overline{x} \vee \overline{y})$$

SAT $\rightarrow$ 3SAT

SAT $\rightarrow$ 3SAT

This is an interesting and common kind of reduction, from a problem to a special case of itself.

SAT → 3SAT

This is an interesting and common kind of reduction, from a problem to a special case of itself.

Given an `instance` I of SAT, use exactly the same `instance` for 3SAT,

SAT $\rightarrow$ 3SAT

This is an interesting and common kind of reduction, from a problem to a special case of itself.

Given an `instance` I of SAT, use exactly the same `instance` for 3SAT, except that any clause with more than three literals,

$$(a_1 \vee a_2 \vee \ldots \vee a_k)$$

is replaced by a set of clauses,

$$(a_1 \vee a_2 \vee y_1)(\overline{y_1} \vee a_3 \vee y_2)(\overline{y_2} \vee a_4 \vee y_3) \ldots (\overline{y_{k-3}} \vee a_{k-1} \vee a_k)$$

where the y_i's are new variables.

SAT → 3SAT

This is an interesting and common kind of reduction, from a problem to a special case of itself.

Given an `instance` I of SAT, use exactly the same `instance` for 3SAT, except that any clause with more than three literals,

$$(a_1 \vee a_2 \vee \ldots \vee a_k)$$

is replaced by a set of clauses,

$$(a_1 \vee a_2 \vee y_1)(\overline{y_1} \vee a_3 \vee y_2)(\overline{y_2} \vee a_4 \vee y_3)\ldots(\overline{y_{k-3}} \vee a_{k-1} \vee a_k)$$

where the y_i's are new variables.

The reduction is in polynomial and I' is equivalent to I in terms of satisfiability.

SAT $\to$ 3SAT

$$\left\{ \begin{array}{c} (a_1 \vee a_2 \vee \cdots \vee a_k) \\ \text{is satisfied} \end{array} \right\} \iff \left\{ \begin{array}{c} \text{there is a setting of the } y_i\text{'s for which} \\ (a_1 \vee a_2 \vee y_1)\ (\overline{y}_1 \vee a_3 \vee y_2)\ \cdots\ (\overline{y}_{k-3} \vee a_{k-1} \vee a_k) \\ \text{are all satisfied} \end{array} \right\}$$

SAT $\rightarrow$ 3SAT

$$\left\{\begin{array}{c}(a_1 \vee a_2 \vee \cdots \vee a_k) \\ \text{is satisfied}\end{array}\right\} \Longleftrightarrow \left\{\begin{array}{c}\text{there is a setting of the } y_i\text{'s for which} \\ (a_1 \vee a_2 \vee y_1)\ (\overline{y}_1 \vee a_3 \vee y_2)\ \cdots\ (\overline{y}_{k-3} \vee a_{k-1} \vee a_k) \\ \text{are all satisfied}\end{array}\right\}$$

Suppose that the clauses on the right are all satisfied. Then at least one of the literals $a_1, \ldots, a_k$ must be `true`.

SAT $\rightarrow$ 3SAT

$$\left\{ \begin{array}{c} (a_1 \vee a_2 \vee \cdots \vee a_k) \\ \text{is satisfied} \end{array} \right\} \Longleftrightarrow \left\{ \begin{array}{c} \text{there is a setting of the } y_i\text{'s for which} \\ (a_1 \vee a_2 \vee y_1)\ (\overline{y}_1 \vee a_3 \vee y_2)\ \cdots\ (\overline{y}_{k-3} \vee a_{k-1} \vee a_k) \\ \text{are all satisfied} \end{array} \right\}$$

Suppose that the clauses on the right are all satisfied. Then at least one of the literals $a_1, \ldots, a_k$ must be `true`. Otherwise y_1 would have to be `true`, which would in turn force y_2 to be `true`, and so on.

SAT $\rightarrow$ 3SAT

$$\left\{ \begin{array}{c} (a_1 \vee a_2 \vee \cdots \vee a_k) \\ \text{is satisfied} \end{array} \right\} \Longleftrightarrow \left\{ \begin{array}{c} \text{there is a setting of the } y_i\text{'s for which} \\ (a_1 \vee a_2 \vee y_1)\ (\overline{y}_1 \vee a_3 \vee y_2)\ \cdots\ (\overline{y}_{k-3} \vee a_{k-1} \vee a_k) \\ \text{are all satisfied} \end{array} \right\}$$

Suppose that the clauses on the right are all satisfied. Then at least one of the literals $a_1, \ldots, a_k$ must be `true`. Otherwise y_1 would have to be `true`, which would in turn force y_2 to be `true`, and so on.

Conversely, if $(a_1 \vee a_2 \vee \ldots \vee a_k)$ is satisfied, then some a_i must be `true`. Set $y_1, \ldots, y_{i-2}$ to `true` and the rest to `false`.

SAT $\rightarrow$ 3SAT

3SAT remains hard even under the further restriction that no variable appears in more than three clauses.

SAT → 3SAT

3SAT remains hard even under the further restriction that no variable appears in more than three clauses.

Suppose that in the 3SAT `instance`, variable x appears in $k > 3$ clauses. Then replace its first appearance by x_1, its second by x_2, and so on, replacing each of its k appearances by a different new variable.

SAT $\to$ 3SAT

3SAT remains hard even under the further restriction that no variable appears in more than three clauses.

Suppose that in the 3SAT `instance`, variable x appears in $k > 3$ clauses. Then replace its first appearance by x_1, its second by x_2, and so on, replacing each of its k appearances by a different new variable.

Finally, add the clauses

$$(\overline{x_1} \vee x_2)(\overline{x_2} \vee x_3) \ldots (\overline{x_k} \vee x_1)$$

SAT $\rightarrow$ 3SAT

3SAT remains hard even under the further restriction that no variable appears in more than three clauses.

Suppose that in the 3SAT `instance`, variable x appears in $k > 3$ clauses. Then replace its first appearance by x_1, its second by x_2, and so on, replacing each of its k appearances by a different new variable.

Finally, add the clauses

$$(\overline{x_1} \vee x_2)(\overline{x_2} \vee x_3)\ldots(\overline{x_k} \vee x_1)$$

In the new formula no variable appears more than three times (and in fact, no literal appears more than twice).

RUDRATA CYCLE $\rightarrow$ TSP

Rudrata cycle $\rightarrow$ TSP

Given a graph $G = (V, E)$, construct the `instance` of the TSP:

RUDRATA CYCLE $\rightarrow$ TSP

Given a graph $G = (V, E)$, construct the `instance` of the TSP:

- The set of nodes is the same as V.
- The distance between cities u and v is 1 if $\{u, v\}$ is an edge of G and $1 + \alpha$ otherwise, for some $\alpha > 1$ to be determined.
- The budget of the TSP `instance` is $|V|$.

RUDRATA CYCLE $\rightarrow$ TSP

Given a graph $G = (V, E)$, construct the `instance` of the TSP:

- The set of nodes is the same as V.
- The distance between cities u and v is 1 if $\{u, v\}$ is an edge of G and $1 + \alpha$ otherwise, for some $\alpha > 1$ to be determined.
- The budget of the TSP `instance` is $|V|$.

If G has a RUDRATA CYCLE, then the same cycle is also a tour within the budget of the TSP `instance`.

RUDRATA CYCLE $\rightarrow$ TSP

Given a graph $G = (V, E)$, construct the `instance` of the TSP:

- The set of nodes is the same as V.
- The distance between cities u and v is 1 if $\{u, v\}$ is an edge of G and $1 + \alpha$ otherwise, for some $\alpha > 1$ to be determined.
- The budget of the TSP `instance` is $|V|$.

If G has a RUDRATA CYCLE, then the same cycle is also a tour within the budget of the TSP `instance`.

If G has no RUDRATA CYCLE, then there is no solution: the cheapest possible TSP tour has cost at least $n + \alpha$.

If $\alpha = 1$, then all distances are either 1 or 2, and so this `instance` of the TSP satisfies the triangle inequality: if i, j, k are cities, then

$$d_{ij} + d_{jk} \geq d_{ik}$$

Rudrata cycle $\rightarrow$ TSP

If $\alpha = 1$, then all distances are either 1 or 2, and so this `instance` of the TSP satisfies the triangle inequality: if i, j, k are cities, then

$$d_{ij} + d_{jk} \geq d_{ik}$$

This is a special case of the TSP which is in a certain sense easier, since it can be efficiently approximated.

RUDRATA CYCLE $\rightarrow$ TSP

If α is large, then the resulting `instance` of the TSP may not satisfy the triangle inequality, and has another important property.

Rudrata cycle $\rightarrow$ TSP

If α is large, then the resulting instance of the TSP may not satisfy the triangle inequality, and has another important property.

This important gap property implies that, unless P = NP, no approximation algorithm is possible.

ANY PROBLEM $\rightarrow$ SAT

home reading!

Homework

Homework

Assignment 6(1 week). Exercises 8.3, 8.9, 8.14 and 8.19.