Analysis of Algorithms

NP-Completeness

Andres Mendez-Vazquez

January 12, 2021

Outline

Outline

Polynomial Time

Algorithms Until Now

All the algorithms, we have studied this far have been polynomial-time algorithms.

However

- There is a collection of algorithms that cannot being solved in polynomial time!!!

Example

- In the Turing's "Halting Problem," we cannot even say if the algorithm is going to stop!!!

Polynomial Time

Algorithms Until Now

All the algorithms, we have studied this far have been polynomial-time algorithms.

However

- There is a collection of algorithms that cannot being solved in polynomial time!!!

Example

- In the Turing's "Halting Problem," we cannot even say if the algorithm is going to stop!!!

Polynomial Time

Algorithms Until Now

All the algorithms, we have studied this far have been polynomial-time algorithms.

However

- There is a collection of algorithms that cannot being solved in polynomial time!!!

Example

- In the Turing's "Halting Problem," we cannot even say if the algorithm is going to stop!!!

Outline

The Intuition

Class P

They are algorithms that on inputs of size *n* have a worst case running time of $O\left(n^k\right)$ for some constant k.

Class NP

Informally, the Non-Polynomial (NP) time algorithms are the ones that cannot be solved in $O\left(n^k\right)$ for any constant k.

The Intuition

Class P

They are algorithms that on inputs of size *n* have a worst case running time of $O\left(n^k\right)$ for some constant k.

Class NP

Informally, the Non-Polynomial (NP) time algorithms are the ones that cannot be solved in $O\left(n^k\right)$ for any constant k.

There are still many thing to say about NP problems

But the one that is making everybody crazy

There is a theorem that hints to a possibility of $NP = P$

Thus

We have the following vision of the world of problems in computer science.

There are still many thing to say about NP problems

But the one that is making everybody crazy

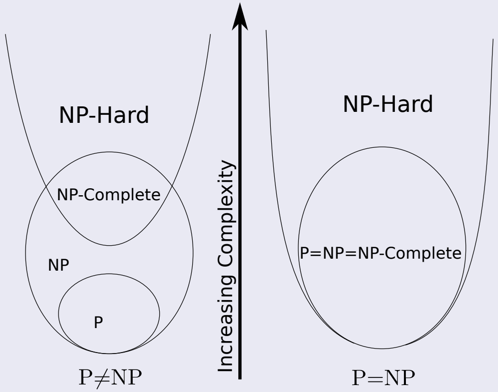
There is a theorem that hints to a possibility of $NP = P$

Thus

We have the following vision of the world of problems in computer science.

The Two Views of The World

The Paradox



However, There are differences pointing to $P \neq NP$

Shortest Path is in P

Even with negative edge weights, we can find a shortest path for a single source in a directed graph $G = (V, E)$ in $O(VE)$ time.

Longest Path is in NP

Merely determining if a graph contains a simple path with at least a given number of edges is NP.

It is more

A simple change on a polynomial time problem can move it from P to NP.

However, There are differences pointing to $P \neq NP$

Shortest Path is in P

Even with negative edge weights, we can find a shortest path for a single source in a directed graph $G = (V, E)$ in $O(VE)$ time.

Longest Path is in NP

Merely determining if a graph contains a simple path with at least a given number of edges is NP.

It is more

A simple change on a polynomial time problem can move it from P to NP.

However, There are differences pointing to $P \neq NP$

Shortest Path is in P

Even with negative edge weights, we can find a shortest path for a single source in a directed graph $G = (V, E)$ in $O(VE)$ time.

Longest Path is in NP

Merely determining if a graph contains a simple path with at least a given number of edges is NP.

It is more

A simple change on a polynomial time problem can move it from P to NP.

cinvestav

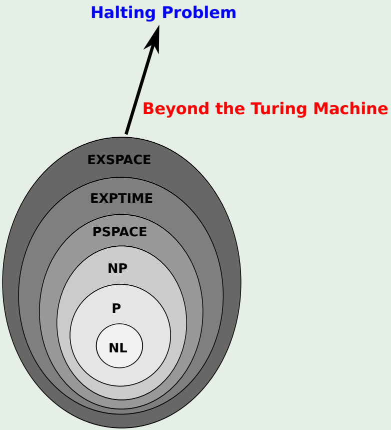
And here, a simplified classification of problems

Different Complexity Classes

Complexity Class	Model of Computation	Resource Constraint
P	Deterministic Turing Machine	Solvable using poly(n) time
NP	Non-deterministic Turing Machine	Verifiable in poly(n) time
PSPACE	Deterministic Turing Machine	Solvable using poly(n) Space
EXPTIME	Deterministic Turing Machine	Solvable using $2^{\text{poly}(n)}$ time
EXPSPACE	Deterministic Turing Machine	Space $2^{\text{poly}(n)}$
NL	Non-deterministic Turing Machine	Space $O(\log n)$

Graphically

What is contained into what



Outline

We start by formalizing the notion of polynomial time

Polynomial Time Problems

We generally regard these problems as tractable, but for philosophical, not mathematical, reasons.

First

It is possible to regard a problem with complexity $O(n^{100})$ as intractable, really few practical problems require time complexities with such high degree polynomial.

It is more

Experience has shown that once the first polynomial-time algorithm for a problem has been discovered, more efficient algorithms often follow.

We start by formalizing the notion of polynomial time

Polynomial Time Problems

We generally regard these problems as tractable, but for philosophical, not mathematical, reasons.

First

It is possible to regard a problem with complexity $O\left(n^{100}\right)$ as intractable, really few practical problems require time complexities with such high degree polynomial.

It is more

Experience has shown that once the first polynomial-time algorithm for a problem has been discovered, more efficient algorithms often follow.

We start by formalizing the notion of polynomial time

Polynomial Time Problems

We generally regard these problems as tractable, but for philosophical, not mathematical, reasons.

First

It is possible to regard a problem with complexity $O\left(n^{100}\right)$ as intractable, really few practical problems require time complexities with such high degree polynomial.

It is more

Experience has shown that once the first polynomial-time algorithm for a problem has been discovered, more efficient algorithms often follow.

Reasons

Second

For many reasonable models of computation, a problem that can be solved in polynomial time in one model can be solved in polynomial time in another.

Example

Problems that can be solved in polynomial time by a serial random-access machine can be solved in a Turing Machine.

- Since polynomials are closed under addition, multiplication, and composition.

Reasons

Second

For many reasonable models of computation, a problem that can be solved in polynomial time in one model can be solved in polynomial time in another.

Example

Problems that can be solved in polynomial time by a serial random-access machine can be solved in a Turing Machine.

Third

- The class of polynomial-time solvable problems has nice closure properties.
- Since polynomials are closed under addition, multiplication, and composition.

Reasons

Second

For many reasonable models of computation, a problem that can be solved in polynomial time in one model can be solved in polynomial time in another.

Example

Problems that can be solved in polynomial time by a serial random-access machine can be solved in a Turing Machine.

Third

- The class of polynomial-time solvable problems has nice closure properties.
- Since polynomials are **closed** under addition, multiplication, and composition.

Reasons

Why?

For example, if the output of one polynomial time algorithm is fed into the input of another, the composite algorithm is polynomial.

Polynomial time

To understand a polynomial time we need to define:

- What is the meaning of an abstract problem?
- How to encode problems.
- A formal language framework.

Polynomial time

To understand a polynomial time we need to define:

- What is the meaning of an abstract problem?
- How to encode problems.
- A formal language framework.

Polynomial time

To understand a polynomial time we need to define:

- What is the meaning of an abstract problem?
- How to encode problems.
- A formal language framework.

Polynomial time

To understand a polynomial time we need to define:

- What is the meaning of an abstract problem?
- How to encode problems.
- A formal language framework.

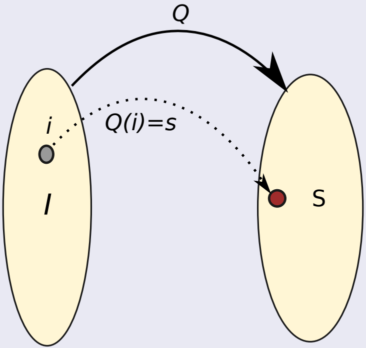
Outline

What do we need to understand the polynomial time?

What is an abstract problem?

We define an **abstract** problem Q to be a binary relation on a set I of problem **instances** and a set S of problem solutions:

$$Q : I \rightarrow S,\ Q(i) = s \tag{1}$$



Abstract problem as decision problems

Something Notable

The formulation is too general to our purpose!!!

Abstract problem as decision problems

Something Notable

The formulation is too general to our purpose!!!

Thus, we do a restriction

The theory of NP-completeness restricts attention to **decision problems:**

- Those having a **YES/NO** solution.

Abstract problem as decision problems

Something Notable

The formulation is too general to our purpose!!!

Thus, we do a restriction

The theory of NP-completeness restricts attention to **decision problems:**

- Those having a **YES/NO** solution.

Then

We can view an abstract decision problem as a function that maps the instance set I to the solution set $\{0,1\}$:

$$Q : I \rightarrow \{0,1\}$$

Abstract problem as decision problems

Something Notable

The formulation is too general to our purpose!!!

Thus, we do a restriction

The theory of NP-completeness restricts attention to **decision problems:**

- Those having a **YES/NO** solution.

Then

We can view an abstract decision problem as a function that maps the instance set I to the solution set $\{0,1\}$:

$$Q : I \rightarrow \{0,1\}$$

Example

Example of optimization problem: SHORTEST-PATH

The problem SHORTEST-PATH is the one that associates each graph G and two vertices with the shortest path between them.

Problem, this is a optimization problem

We need a decision problem!!!

What do we do?

We can cast a given optimization problem as a related decision problem by imposing a bound on the value to be optimized.

Example

Example of optimization problem: SHORTEST-PATH

The problem SHORTEST-PATH is the one that associates each graph G and two vertices with the shortest path between them.

Problem, this is a optimization problem

We need a decision problem!!!

What do we do?

We can cast a given optimization problem as a related decision problem by imposing a bound on the value to be optimized.

Example

Example of optimization problem: SHORTEST-PATH

The problem SHORTEST-PATH is the one that associates each graph G and two vertices with the shortest path between them.

Problem, this is a optimization problem

We need a decision problem!!!

What do we do?

We can cast a given optimization problem as a related decision problem by imposing a bound on the value to be optimized.

Thus

Example PATH Problem

Given a undirected graph G, vertices u and v, and an integer k, we need to answer the following question:

- **Does a path exist from u to v consisting of at most k edges?**

In a more formal way

We have the following optimization problem

$$
\begin{aligned}
\min_{t} \;\; & d\left[t\right] \\
& s.t. d[v] \leq d[u] + w(u,v) \text{ for each edge } (u,v) \in E \\
& d\left[s\right] = 0
\end{aligned}
$$

Then, we have the following decision problem

$PATH = \{\langle G, u, v, k\rangle \,|\, G = (V, E)$ is an undirected graph,
$u, v \in V, k \geq 0$ is an integer and there exist a path from
u to v in G consisting of at most k edges$\}$

In a more formal way

We have the following optimization problem

$$\begin{aligned} &\min_{t} \ d[t] \\ &\quad s.t. d[v] \leq d[u] + w(u,v) \text{ for each edge } (u,v) \in E \\ &\quad\quad d[s] = 0 \end{aligned}$$

Then, we have the following decision problem

$$\begin{aligned} PATH = \{\langle G,u,v,k\rangle \,|G = (V,E) &\text{ is an undirected graph,} \\ u,v \in V, k \geq 0 &\text{ is an integer and there exist a path from} \\ u \text{ to } v \text{ in } G &\text{ consisting of at most } k \text{ edges}\} \end{aligned}$$

Outline

Why Encoding?

We need to represent our problems in a way that the Turing machine can understand.

That is called an encoding.

Why Encoding?

We need to represent our problems in a way that the Turing machine can understand.

That is called an encoding.

Example

Given a graph $G = (V, E)$:

- We can encode each vertex $\{1, 2, ...\}$ as $\{0, 1, 10, ...\}$
- Then, each edge, for example $\{1, 2\}$ as $\{0, 1\}$
- Clearly you need to encode some kind of delimiter for each element in the description

Why Encoding?

We need to represent our problems in a way that the Turing machine can understand.

That is called an encoding.

Example

Given a graph $G = (V, E)$:

- We can encode each vertex $\{1, 2, ...\}$ as $\{0, 1, 10, ...\}$
- Then, each edge, for example $\{1, 2\}$ as $\{0, 1\}$
- Clearly you need to encode some kind of delimiter for each element in the description

Why Encoding?

We need to represent our problems in a way that the Turing machine can understand.

That is called an encoding.

Example

Given a graph $G = (V, E)$:

- We can encode each vertex $\{1, 2, ...\}$ as $\{0, 1, 10, ...\}$
- Then, each edge, for example $\{1, 2\}$ as $\{0, 1\}$
- Clearly you need to encode some kind of delimiter for each element in the description

Why Encoding?

We need to represent our problems in a way that the Turing machine can understand.

That is called an encoding.

Example

Given a graph $G = (V, E)$:

- We can encode each vertex $\{1, 2, ...\}$ as $\{0, 1, 10, ...\}$
- Then, each edge, for example $\{1, 2\}$ as $\{0, 1\}$
- Clearly you need to encode some kind of delimiter for each element in the description

Why a Turing machine or a RAM machine?

Given an encoding...

We need a computational device to solve the encoded problem

Why a Turing machine or a RAM machine?

Given an encoding...

We need a computational device to solve the encoded problem

Some facts

- This means that when the device solves a problem in reality solves the encoded version of Q.
- This encoded problem is called a ***concrete problem***.
- This tells us how important encoding is!!!

Why a Turing machine or a RAM machine?

Given an encoding...

We need a computational device to solve the encoded problem

Some facts

- This means that when the device solves a problem in reality solves the encoded version of Q.
- This encoded problem is called a ***concrete problem*.**
- This tells us how important encoding is!!!

Why a Turing machine or a RAM machine?

Given an encoding...

We need a computational device to solve the encoded problem

Some facts

- This means that when the device solves a problem in reality solves the encoded version of Q.
- This encoded problem is called a ***concrete problem*.**
- This tells us how important encoding is!!!

It is more!!!

We want time complexities of $O\left(T\left(n\right)\right)$

When it is provided with a problem instance i of length $|i| = n$.

Then

The algorithm can produce the solution in $O\left(T\left(n\right)\right)$.

It is more!!!

We want time complexities of $O\left(T\left(n\right)\right)$

When it is provided with a problem instance i of length $|i| = n$.

Then

The algorithm can produce the solution in $O\left(T\left(n\right)\right)$.

Using Encodings

Something Notable

Given an abstract decision problem Q mapping an instance set I to $\{0, 1\}$.

Then

An encoding $e : I \longrightarrow \{0, 1\}^*$ can induce a related concrete decision problem, denoted as by $e(Q)$.

- Then the solution to the concrete problem instance $e(i) \in \{0, 1\}^*$ is also $Q(i)$.

Using Encodings

Something Notable

Given an abstract decision problem Q mapping an instance set I to $\{0, 1\}$.

Then

An encoding $e : I \longrightarrow \{0, 1\}^*$ can induce a related concrete decision problem, denoted as by $e(Q)$.

IMPORTANT

- If the solution to an abstract-problem instance $i \in I$ is $Q(i) \in \{0, 1\}$.
- Then the solution to the concrete problem instance $e(i) \in \{0, 1\}^*$ is also $Q(i)$.

Using Encodings

Something Notable

Given an abstract decision problem Q mapping an instance set I to $\{0,1\}$.

Then

An encoding $e : I \longrightarrow \{0,1\}^*$ can induce a related concrete decision problem, denoted as by $e(Q)$.

IMPORTANT

- If the solution to an abstract-problem instance $i \in I$ is $Q(i) \in \{0,1\}$.
- Then the solution to the concrete-problem instance $e(i) \in \{0,1\}^*$ is also $Q(i)$.

Using Encodings

Something Notable

Given an abstract decision problem Q mapping an instance set I to $\{0, 1\}$.

Then

An encoding $e : I \longrightarrow \{0, 1\}^*$ can induce a related concrete decision problem, denoted as by $e(Q)$.

IMPORTANT

- If the solution to an abstract-problem instance $i \in I$ is $Q(i) \in \{0, 1\}$.
- Then the solution to the concrete-problem instance $e(i) \in \{0, 1\}^*$ is also $Q(i)$.

What do we want?

Something Notable

We would like to extend the definition of polynomial-time solvability from concrete problems to abstract problems by using encodings as the bridge.

IMPORTANT!!!

We want the definition to be independent of any particular encoding.

- HOWEVER, it depends quite heavily on the encoding.

What do we want?

Something Notable

We would like to extend the definition of polynomial-time solvability from concrete problems to abstract problems by using encodings as the bridge.

IMPORTANT!!!

We want the definition to be independent of any particular encoding.

In other words

The efficiency of solving a problem should not depend on how the problem is encoded.

- HOWEVER, it depends quite heavily on the encoding.

What do we want?

Something Notable

We would like to extend the definition of polynomial-time solvability from concrete problems to abstract problems by using encodings as the bridge.

IMPORTANT!!!

We want the definition to be independent of any particular encoding.

In other words

The efficiency of solving a problem should not depend on how the problem is encoded.

- **HOWEVER, it depends quite heavily on the encoding.**

An example of a really BAD situation

Imagine the following

- You could have an algorithm that takes k as the sole input with an algorithm that runs in $\Theta(k)$.

Now, if the integer is provided in an unary representation (Only ones)

Quite naive!!!

Then

- Running time of the algorithm is $O(n)$ on n-length inputs, which is polynomial.

An example of a really BAD situation

Imagine the following

- You could have an algorithm that takes k as the sole input with an algorithm that runs in $\Theta(k)$.

Now, if the integer is provided in an unary representation (Only ones)

Quite naive!!!

Then

- Running time of the algorithm is $O(n)$ on n-length inputs, which is polynomial.

An example of a really BAD situation

Imagine the following

- You could have an algorithm that takes k as the sole input with an algorithm that runs in $\Theta(k)$.

Now, if the integer is provided in an unary representation (Only ones)

Quite naive!!!

Then

- Running time of the algorithm is $O(n)$ on n-length inputs, which is polynomial.

For example

Now, use the more natural binary representation of the integer k

- Now, given a binary representation:
  - Thus the input length is $n = \lfloor \log k \rfloor + 1 \rightarrow \Theta(k) = \Theta(2^n)$

For example

Now, use the more natural binary representation of the integer k

- Now, given a binary representation:
  - Thus the input length is $n = \lfloor \log k \rfloor + 1 \to \Theta(k) = \Theta(2^n)$

Remark

Thus, depending on the encoding, the algorithm runs in either polynomial or superpolynomial time.

For example

Now, use the more natural binary representation of the integer k

- Now, given a binary representation:
  - Thus the input length is $n = \lfloor \log k \rfloor + 1 \rightarrow \Theta(k) = \Theta(2^n)$

Remark

Thus, depending on the encoding, the algorithm runs in either polynomial or superpolynomial time.

More Observations

Thus

How we encode an abstract problem matters quite a bit to how we understand it!!!

It is more

We cannot talk about solving an abstract problem without specifying the encoding!!!

Nevertheless

If we rule out expensive encodings such as unary ones, the actual encoding of a problem makes little difference to whether the problem can be solved in polynomial time.

More Observations

Thus

How we encode an abstract problem matters quite a bit to how we understand it!!!

It is more

We cannot talk about solving an abstract problem without specifying the encoding!!!

Nevertheless

If we rule out expensive encodings such as unary ones, the actual encoding of a problem makes little difference to whether the problem can be solved in polynomial time.

More Observations

Thus

How we encode an abstract problem matters quite a bit to how we understand it!!!

It is more

We cannot talk about solving an abstract problem without specifying the encoding!!!

Nevertheless

If we rule out expensive encodings such as unary ones, the actual encoding of a problem makes little difference to whether the problem can be solved in polynomial time.

Some properties of the polynomial encoding

First

We say that a function $f : \{0,1\}^* \to \{0,1\}^*$ is polynomial time computable, if there exists a polynomial time algorithm A that, given any input $x \in \{0,1\}^*$, produces as output $f(x)$.

Some properties of the polynomial encoding

First

We say that a function $f : \{0,1\}^* \to \{0,1\}^*$ is polynomial time computable, if there exists a polynomial time algorithm A that, given any input $x \in \{0,1\}^*$, produces as output $f(x)$.

Second

For some set I of problem instances, we say that two encodings e_1 and e_2 are polynomially related

- if there exist two polynomial time computable functions f_{12} and f_{21} such that for any $i \in I$, we have $f_{12}(e_1(i)) = e_2(i)$ and $f_{21}(e_2(i)) = e_1(i)$.

Some properties of the polynomial encoding

First

We say that a function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ is polynomial time computable, if there exists a polynomial time algorithm A that, given any input $x \in \{0,1\}^*$, produces as output $f(x)$.

Second

For some set I of problem instances, we say that two encodings e_1 and e_2 are polynomially related

- if there exist two polynomial time computable functions f_{12} and f_{21} such that for any $i \in I$, we have $f_{12}(e_1(i)) = e_2(i)$ and $f_{21}(e_2(i)) = e_1(i)$.

Some properties of the polynomial encoding

First

We say that a function $f : \{0,1\}^* \to \{0,1\}^*$ is polynomial time computable, if there exists a polynomial time algorithm A that, given any input $x \in \{0,1\}^*$, produces as output $f(x)$.

Second

For some set I of problem instances, we say that two encodings e_1 and e_2 are polynomially related

- if there exist two polynomial time computable functions f_{12} and f_{21} such that for any $i \in I$, we have $f_{12}(e_1(i)) = e_2(i)$ and $f_{21}(e_2(i)) = e_1(i)$.

Observation

We have that

A polynomial-time algorithm can compute the encoding $e_2(i)$ from the encoding $e_1(i)$, and vice versa.

Observation

We have that

A polynomial-time algorithm can compute the encoding $e_2(i)$ from the encoding $e_1(i)$, and vice versa.

Something Notable

If two encodings e_1 and e_2 of an abstract problem are polynomially related

- We have that if the problem is polynomial-time solvable or not is **independent of which encoding we use.**

Observation

We have that

A polynomial-time algorithm can compute the encoding $e_2(i)$ from the encoding $e_1(i)$, and vice versa.

Something Notable

If two encodings e_1 and e_2 of an abstract problem are polynomially related

- We have that if the problem is polynomial-time solvable or not is **independent of which encoding we use.**

Observation

We have that

A polynomial-time algorithm can compute the encoding $e_2(i)$ from the encoding $e_1(i)$, and vice versa.

Something Notable

If two encodings e_1 and e_2 of an abstract problem are polynomially related

- We have that if the problem is polynomial-time solvable or not is **independent of which encoding we use**.

An important lemma

Lemma 34.1

Let Q be an abstract decision problem on an instance set I, and let e_1 and e_2 be polynomially related encodings on I. Then, $e_1(Q) \in P$ if and only if $e_2(Q) \in P$.

Proof in the board

An important lemma

Lemma 34.1

Let Q be an abstract decision problem on an instance set I, and let e_1 and e_2 be polynomially related encodings on I. Then, $e_1(Q) \in P$ if and only if $e_2(Q) \in P$.

Proof in the board

Outline

Formal language framework to handle representation

Definitions

1. An alphabet Σ is a finite set of symbols.
2. A language L over Σ is any set of strings made up of symbols from Σ^*.
3. The empty language is ε.
4. The language of all strings over Σ is denoted Σ^*.

Formal language framework to handle representation

Definitions

1. An alphabet Σ is a finite set of symbols.
2. A language L over Σ is any set of strings made up of symbols from Σ^*.
3. The empty language is ε.
4. The language of all strings over Σ is denoted Σ^*.

Formal language framework to handle representation

Definitions

1. An alphabet Σ is a finite set of symbols.
2. A language L over Σ is any set of strings made up of symbols from Σ^*.
3. The empty language is ε.
4. The language of all strings over Σ is denoted Σ^*.

Formal language framework to handle representation

Definitions

1. An alphabet Σ is a finite set of symbols.
2. A language L over Σ is any set of strings made up of symbols from Σ^*.
3. The empty language is ε.
4. The language of all strings over Σ is denoted Σ^*.

Special languages and operations

Union, intersection and complement

- $L_1 \bigcap L_2 = \{x \in \Sigma^* | \ x \in L_1 \wedge x \in L_2\}$
- $L_1 \bigcup L_2 = \{x \in \Sigma^* | \ x \in L_1 \vee x \in L_2\}$
- $\bar{L} = \{x \in \Sigma^* | \ x \neq L\}$

Special languages and operations

Union, intersection and complement

- $L_1 \bigcap L_2 = \{x \in \Sigma^* |\ x \in L_1 \wedge x \in L_2\}$
- $L_1 \bigcup L_2 = \{x \in \Sigma^* |\ x \in L_1 \vee x \in L_2\}$
- $\bar{L} = \{x \in \Sigma^* |\ x \neq L\}$

Concatenation

- $L = \{x_1 x_2 \mid x_1 \in L_1$ and $x_2 \in L_2\}$

Special languages and operations

Union, intersection and complement

- $L_1 \bigcap L_2 = \{x \in \Sigma^* |\ x \in L_1 \wedge x \in L_2\}$
- $L_1 \bigcup L_2 = \{x \in \Sigma^* |\ x \in L_1 \vee x \in L_2\}$
- $\bar{L} = \{x \in \Sigma^* |\ x \neq L\}$

Concatenation

- $L = \{x_1 x_2 \mid x_1 \in L_1$ and $x_2 \in L_2\}$

Kleene closure

- $L^* = \{\varepsilon\} \bigcup L^2 \bigcup L^3 \bigcup \ldots$

Special languages and operations

Union, intersection and complement

- $L_1 \bigcap L_2 = \{x \in \Sigma^* |\ x \in L_1 \wedge x \in L_2\}$
- $L_1 \bigcup L_2 = \{x \in \Sigma^* |\ x \in L_1 \vee x \in L_2\}$
- $\bar{L} = \{x \in \Sigma^* |\ x \neq L\}$

Concatenation

- $L = \{x_1 x_2 \mid x_1 \in L_1$ and $x_2 \in L_2\}$

Kleene closure

- $L^* = \{\varepsilon\} \bigcup L^2 \bigcup L^3 \bigcup \dots$

Special languages and operations

Union, intersection and complement

- $L_1 \bigcap L_2 = \{x \in \Sigma^* |\ x \in L_1 \wedge x \in L_2\}$
- $L_1 \bigcup L_2 = \{x \in \Sigma^* |\ x \in L_1 \vee x \in L_2\}$
- $\bar{L} = \{x \in \Sigma^* |\ x \neq L\}$

Concatenation

- $L = \{x_1 x_2 \mid x_1 \in L_1$ and $x_2 \in L_2\}$

Kleene closure

- $L^* = \{\varepsilon\} \bigcup L^2 \bigcup L^3 \bigcup ...$

Outline

Observation

We have that

From the point of view of language theory,

- The set of instances for any decision problem Q is simply the set $\sum^*$ with $\sum = \{0, 1\}$.

Observation

We have that

From the point of view of language theory,

- The set of instances for any decision problem Q is simply the set $\sum^*$ with $\sum = \{0, 1\}$.

Something Notable

Q is entirely characterized by instances that produce a YES or ONE answer.

Observation

We have that

From the point of view of language theory,

- The set of instances for any decision problem Q is simply the set $\sum^*$ with $\sum = \{0, 1\}$.

Something Notable

Q is entirely characterized by instances that produce a YES or ONE answer.

This allow us to define a language that is solvable by Q

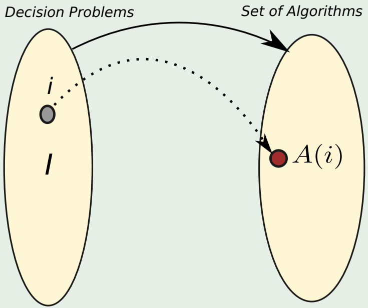
We can write it down as the language

$$L = \{x \in \Sigma^* | Q(x) = 1\} \quad (2)$$

Thus, we can express the duality Decision Problem-Algorithm

Important

The formal-language framework allows to express concisely the relation between decision problems and algorithms that solve them.



Next

Given an instance x of a problem

- An algorithm A **accepts** a string $x \in \{0,1\}^*$, if given x, the algorithm's output is $A(x) = 1$.

The language accepted by an algorithm A is the set of strings

$$L = \{x \in \{0,1\}^* \mid A(x) = 1\}. \quad (3)$$

Next

Given an instance x of a problem

- An algorithm A **accepts** a string $x \in \{0,1\}^*$, if given x, the algorithm's output is $A(x) = 1$.

The language accepted by an algorithm A is the set of strings

$$L = \{x \in \{0,1\}^* \, | A(x) = 1\}. \tag{3}$$

Nevertheless

We have a problem

- Even if language L is accepted by an algorithm A.
- The algorithm will not necessarily reject a string $x \notin L$ provided as input to it.
  - Example: The algorithm could loop forever.

Nevertheless

We have a problem

- Even if language L is accepted by an algorithm A.
- The algorithm will not necessarily reject a string $x \notin L$ provided as input to it.
  - Example: The algorithm could loop forever.

Nevertheless

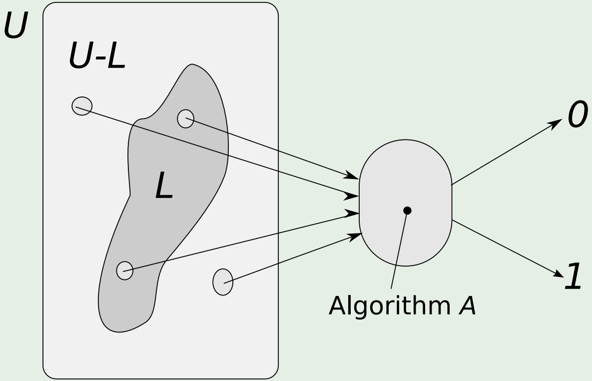
We have a problem

- Even if language L is accepted by an algorithm A.
- The algorithm will not necessarily reject a string $x \notin L$ provided as input to it.
  - Example: The algorithm could loop forever.

Nevertheless

We need to be more stringent

A language L is decided by an algorithm A if every binary string in L is accepted by A and every binary string not in L is rejected by A.



Finally

Thus, a language L is decided by A, if

Given a string $x \in \{0,1\}^*$, only one of two things can happen:

- An algorithm A **accepts**, if given $x \in L$ the algorithm outputs $A(x) = 1$.
- An algorithm A **rejects**, if if given $x \notin L$ the algorithm outputs $A(x) = 0$.

Finally

Thus, a language L is decided by A, if

Given a string $x \in \{0,1\}^*$, only one of two things can happen:

- An algorithm A **accepts**, if given $x \in L$ the algorithm outputs $A(x) = 1$.
- An algorithm A **rejects**, if if given $x \notin L$ the algorithm outputs $A(x) = 0$.

Finally

Thus, a language L is decided by A, if

Given a string $x \in \{0,1\}^*$, only one of two things can happen:

- An algorithm A **accepts**, if given $x \in L$ the algorithm outputs $A(x) = 1$.
- An algorithm A **rejects**, if if given $x \notin L$ the algorithm outputs $A(x) = 0$.

Now, it is possible to define acceptance in polynomial time

We have that

- A language L is **accepted** in polynomial time by an algorithm A, if it is accepted by A in polynomial time:
  - There exists a constant k such that for any n-length string $x \in L \Rightarrow$ algorithm A accepts x in time $O\left(n^k\right)$.

- The algorithm correctly decides whether $x \in L$ in time $O\left(n^k\right)$

Now, it is possible to define acceptance in polynomial time

We have that

- A language L is **accepted** in polynomial time by an algorithm A, if it is accepted by A in polynomial time:
  - There exists a constant k such that for any n-length string $x \in L \Rightarrow$ algorithm A accepts x in time $O\left(n^k\right)$.

Thus

- A language L is **decided in polynomial time** by an algorithm A, if there exists a constant k such that for any n-length string $x \in \{0,1\}^*$:
  - The algorithm correctly decides whether $x \in L$ in time $O\left(n^k\right)$.

Now, it is possible to define acceptance in polynomial time

We have that

- A language L is **accepted** in polynomial time by an algorithm A, if it is accepted by A in polynomial time:
  - There exists a constant k such that for any n-length string $x \in L \Rightarrow$ algorithm A accepts x in time $O\left(n^k\right)$.

Thus

- A language L is **decided in polynomial time** by an algorithm A, if there exists a constant k such that for any n-length string $x \in \{0,1\}^*$:
  - The algorithm correctly decides whether $x \in L$ in time $O\left(n^k\right)$.

Now, it is possible to define acceptance in polynomial time

We have that

- A language L is **accepted** in polynomial time by an algorithm A, if it is accepted by A in polynomial time:
  - There exists a constant k such that for any n-length string $x \in L \Rightarrow$ algorithm A accepts x in time $O\left(n^k\right)$.

Thus

- A language L is **decided in polynomial time** by an algorithm A, if there exists a constant k such that for any n-length string $x \in \{0,1\}^*$:
  - The algorithm correctly decides whether $x \in L$ in time $O\left(n^k\right)$.

Example of polynomial accepted problems

Example of polynomial accepted problem

$$PATH = \{\langle G, u, v, k\rangle \,|G = (V, E) \text{ is an undirected graph,}$$
$$u, v \in V, k \geq 0 \text{ is an integer and there exist a path from}$$
$$u \text{ to } v \text{ in } G \text{ consisting of at most } k \text{ edges}\}$$

Example of polynomial accepted problems

Example of polynomial accepted problem

$$PATH = \{\langle G, u, v, k\rangle \,|G = (V, E) \text{ is an undirected graph,} \\ u, v \in V, k \geq 0 \text{ is an integer and there exist a path from} \\ u \text{ to } v \text{ in } G \text{ consisting of at most } k \text{ edges}\}$$

What does the polynomial times accepting algorithm do?

- One polynomial-time accepting algorithm does the following
  - It verifies that G encodes an undirected graph.
  - It calculate the shortest path between vertices and compares the number of edges of that path with k.
  - If it finds such a path outputs ONE and halt.
  - If it does not, it runs forever!!! $\Longleftarrow$ **PROBLEM!!!**

Example of polynomial accepted problems

Example of polynomial accepted problem

$$PATH = \{\langle G, u, v, k\rangle \, | G = (V, E) \text{ is an undirected graph,}$$
$$u, v \in V, k \geq 0 \text{ is an integer and there exist a path from}$$
$$u \text{ to } v \text{ in } G \text{ consisting of at most } k \text{ edges}\}$$

What does the polynomial times accepting algorithm do?

- One polynomial-time accepting algorithm does the following
  - It verifies that G encodes an undirected graph.
  - It calculate the shortest path between vertices and compares the number of edges of that path with k.
  - If it finds such a path outputs ONE and halt.
  - If it does not, it runs forever!!! $\Longleftarrow$ **PROBLEM!!!**

Example of polynomial accepted problems

Example of polynomial accepted problem

$$PATH = \{\langle G, u, v, k\rangle \,|G = (V, E) \text{ is an undirected graph,}$$
$$u, v \in V, k \geq 0 \text{ is an integer and there exist a path from}$$
$$u \text{ to } v \text{ in } G \text{ consisting of at most } k \text{ edges}\}$$

What does the polynomial times accepting algorithm do?

- One polynomial-time accepting algorithm does the following
  - It verifies that G encodes an undirected graph.
  - It calculate the shortest path between vertices and compares the number of edges of that path with k.
  - If it finds such a path outputs ONE and halt.
  - If it does not, it runs forever!!! $\Longleftarrow$ **PROBLEM!!!**

Example of polynomial accepted problems

Example of polynomial accepted problem

$$PATH = \{\langle G, u, v, k \rangle \, | G = (V, E) \text{ is an undirected graph,}$$
$$u, v \in V, k \geq 0 \text{ is an integer and there exist a path from}$$
$$u \text{ to } v \text{ in } G \text{ consisting of at most } k \text{ edges}\}$$

What does the polynomial times accepting algorithm do?

- One polynomial-time accepting algorithm does the following
  - It verifies that G encodes an undirected graph.
  - It calculate the shortest path between vertices and compares the number of edges of that path with k.
  - If it finds such a path outputs ONE and halt.
  - If it does not, it runs forever!!! $\Longleftarrow$ **PROBLEM!!!**

Example of polynomial accepted problems

Example of polynomial accepted problem

$$PATH = \{\langle G, u, v, k\rangle \,|G = (V, E) \text{ is an undirected graph,}$$
$$u, v \in V, k \geq 0 \text{ is an integer and there exist a path from}$$
$$u \text{ to } v \text{ in } G \text{ consisting of at most } k \text{ edges}\}$$

What does the polynomial times accepting algorithm do?

- One polynomial-time accepting algorithm does the following
  - It verifies that G encodes an undirected graph.
  - It calculate the shortest path between vertices and compares the number of edges of that path with k.
  - If it finds such a path outputs ONE and halt.
  - If it does not, it runs forever!!! $\Longleftarrow$ **PROBLEM!!!**

What we will like to have...

A decision algorithm

Because we want to avoid the infinite loop, we do the following...

What we will like to have...

A decision algorithm

Because we want to avoid the infinite loop, we do the following...

Steps

- It verifies that G encodes an undirected graph.
- It calculate the shortest path between vertices and compares the number of edges of that path with k.
- If it finds such a path outputs ONE and halt.
- If it does not find such a path output ZERO and halt.

What we will like to have...

A decision algorithm

Because we want to avoid the infinite loop, we do the following...

Steps

- It verifies that G encodes an undirected graph.
- It calculate the shortest path between vertices and compares the number of edges of that path with k.
- If it finds such a path outputs ONE and halt.
- If it does not find such a path output ZERO and halt.

What we will like to have...

A decision algorithm

Because we want to avoid the infinite loop, we do the following...

Steps

- It verifies that G encodes an undirected graph.
- It calculate the shortest path between vertices and compares the number of edges of that path with k.
- If it finds such a path outputs ONE and halt.
- If it does not find such a path output ZERO and halt.

What we will like to have...

A decision algorithm

Because we want to avoid the infinite loop, we do the following...

Steps

- It verifies that G encodes an undirected graph.
- It calculate the shortest path between vertices and compares the number of edges of that path with k.
- If it finds such a path outputs ONE and halt.
- If it does not find such a path output ZERO and halt.

However

There are problems

As the Turing's Halting Problem where:

- There exists an accepting algorithm
- But no decision algorithm exist

However

There are problems

As the Turing's Halting Problem where:

- There exists an accepting algorithm
- But no decision algorithm exist

What!?

It turns out there are perfectly decent computational problems for which no algorithms exist at all!

However

There are problems

As the Turing's Halting Problem where:

- There exists an accepting algorithm
- But no decision algorithm exist

What!?

It turns out there are perfectly decent computational problems for which no algorithms exist at all!

For example, an arithmetical version of what will talk later, the SAT problem

Given a polynomial equation in many variables, perhaps:

$$x^3yz + 2y^4z^2 - 7xy^5z = 6$$

are there integer values of x, y, z that satisfy it?

However

There are problems

As the Turing's Halting Problem where:

- There exists an accepting algorithm
- But no decision algorithm exist

What!?

It turns out there are perfectly decent computational problems for which no algorithms exist at all!

For example, an arithmetical version of what will talk later, the SAT problem

Given a polynomial equation in many variables, perhaps:

$$x^3yz + 2y^4z^2 - 7xy^5z = 6$$

are there integer values of x, y, z that satisfy it?

However

There are problems

As the Turing's Halting Problem where:

- There exists an accepting algorithm
- But no decision algorithm exist

What!?

It turns out there are perfectly decent computational problems for which no algorithms exist at all!

For example, an arithmetical version of what will talk later, the SAT problem

Given a polynomial equation in many variables, perhaps:

$$x^3yz + 2y^4z^2 - 7xy^5z = 6$$

are there integer values of x, y, z that satisfy it?

Actually

Something Notable

- There is no algorithm that solves this problem.
- No algorithm at all, polynomial, exponential, doubly exponential, or worse!
  - Such problems are called unsolvable.

Actually

Something Notable

- There is no algorithm that solves this problem.
- No algorithm at all, polynomial, exponential, doubly exponential, or worse!
  - Such problems are called unsolvable.

This was discovered by

The first unsolvable problem was discovered in 1936 by Alan M. Turing, then a student of mathematics at Cambridge, England.

Actually

Something Notable

- There is no algorithm that solves this problem.
- No algorithm at all, polynomial, exponential, doubly exponential, or worse!
  - Such problems are called unsolvable.

This was discovered by

The first unsolvable problem was discovered in 1936 by Alan M. Turing, then a student of mathematics at Cambridge, England.

Actually

Something Notable

- There is no algorithm that solves this problem.
- No algorithm at all, polynomial, exponential, doubly exponential, or worse!
  - Such problems are called unsolvable.

This was discovered by

The first unsolvable problem was discovered in 1936 by Alan M. Turing, then a student of mathematics at Cambridge, England.

What did he do?

Basic Idea

1. Suppose that given a program p and an input x.
2. There is an algorithm, called TERMINATE, that takes p and x and tell us if p will ever terminate in x.

What did he do?

Basic Idea

1. Suppose that given a program p and an input x.
   1. There is an algorithm, called TERMINATE, that takes p and x and tell us if p will ever terminate in x.

Then

We have the following program

function PARADOX$(z : file)$

1. if TERMINATES(z, z) goto 1

Notice what paradox does

It terminates if and only if program z does not terminate when given its own code as input.

1. Case I : The PARADOX terminates $\Rightarrow$ Then TERMINATE says false!!!
2. Case II : The PARADOX never terminates $\Rightarrow$ Then TERMINATE says true!!!

Then

We have the following program

function PARADOX($z : file$)

1. if TERMINATES(z, z) goto 1

Notice what paradox does

It terminates if and only if program z does not terminate when given its own code as input.

What if run PARADOX(PARADOX)

Funny PARADOX!!!

1. Case I : The PARADOX terminates $\Rightarrow$ Then TERMINATE says false!!!
2. Case II : The PARADOX never terminates $\Rightarrow$ Then TERMINATE says true!!!

Then

We have the following program

function PARADOX($z : file$)

1. if TERMINATES(z, z) goto 1

Notice what paradox does

It terminates if and only if program z does not terminate when given its own code as input.

What if run PARADOX(PARADOX)

Funny PARADOX!!!

1. Case I : The PARADOX terminates -> Then TERMINATE says false!!!
2. Case II : The PARADOX never terminates -> Then TERMINATE says true!!!

Then

We have the following program

function PARADOX$(z : file)$

1. if TERMINATES(z, z) goto 1

Notice what paradox does

It terminates if and only if program z does not terminate when given its own code as input.

What if run PARADOX(PARADOX)

Funny PARADOX!!!

1. Case I : The PARADOX terminates -> Then TERMINATE says false!!!
2. Case II : The PARADOX never terminates -> Then TERMINATE says true!!!

Outline

Complexity Classes

Then

We can informally define a **complexity class** as a set of languages.

Now

The membership to this class is determined by a **complexity measure**, such as running time, of an algorithm that determines whether a given string x belongs to language L.

However

The actual definition of a complexity class is somewhat more technical.

Complexity Classes

Then

We can informally define a **complexity class** as a set of languages.

Now

The membership to this class is determined by a **complexity measure**, such as running time, of an algorithm that determines whether a given string x belongs to language L.

However

The actual definition of a complexity class is somewhat more technical.

Complexity Classes

Then

We can informally define a **complexity class** as a set of languages.

Now

The membership to this class is determined by a **complexity measure**, such as running time, of an algorithm that determines whether a given string x belongs to language L.

However

The actual definition of a complexity class is somewhat more technical.

Thus

We can use this framework to say the following

- $P = \{L \subseteq \{0,1\}^* \mid$ There exists an algorithm A that decides L in polynomial time$\}$
  - In fact, P is also the class of languages that can be accepted in polynomial time

Thus

We can use this framework to say the following

- $P = \{L \subseteq \{0,1\}^* \mid$ There exists an algorithm A that decides L in polynomial time$\}$
  - In fact, P is also the class of languages that can be accepted in polynomial time

Theorem 34.2

$P = \{L \mid L$ is accepted by a polynomial-time algorithm$\}$

Thus

We can use this framework to say the following

- $P = \{L \subseteq \{0,1\}^* \mid$ There exists an algorithm A that decides L in polynomial time$\}$
  - In fact, P is also the class of languages that can be accepted in polynomial time

Theorem 34.2

P=$\{L|$ L is accepted by a polynomial-time algorithm$\}$

Exercises

From Cormen's book solve

- 34.1-1
- 34.1-2
- 34.1-3
- 34.1-4
- 34.1-5
- 34.1-6

Outline

What is verification?

Intuitive definition

Given an instance of a decision problem:

- For example $\langle G, u, v, k \rangle$ of PATH.

What is verification?

Intuitive definition

Given an instance of a decision problem:

- For example $\langle G, u, v, k \rangle$ of PATH.

Then

We are given :

- A path p from A to F.

Then, check if the length of p is at most k (i.e. Belongs to PATH), then it is called a "certificate."

What is verification?

Intuitive definition

Given an instance of a decision problem:

- For example $\langle G, u, v, k \rangle$ of PATH.

Then

We are given :

- A path p from A to F.

Then, check if the length of p is at most k (i.e. Belongs to PATH), then it is called a "certificate."

What is verification?

Intuitive definition

Given an instance of a decision problem:

- For example $\langle G, u, v, k \rangle$ of PATH.

Then

We are given :

- A path p from A to F.

Then, check if the length of p is at most k (i.e. Belongs to PATH), then it is called a "certificate."

It is more

In fact

- We would like to be able to verify in polynomial time the certificate of certain types of problems.
- For example:
  - Polynomial time problems.
  - Non-Polynomial time problems.

It is more

In fact

- We would like to be able to verify in polynomial time the certificate of certain types of problems.
- For example:
  - Polynomial time problems.
  - Non-Polynomial time problems.

It is more

In fact

- We would like to be able to verify in polynomial time the certificate of certain types of problems.
- For example:
  - Polynomial time problems.
  - Non-Polynomial time problems.

It is more

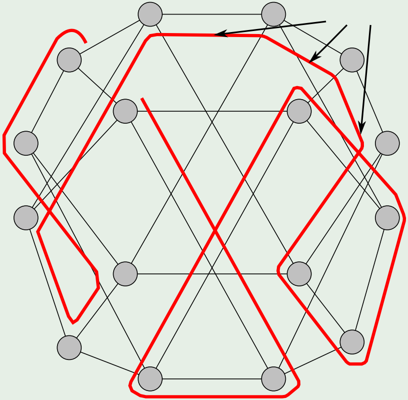
In fact

- We would like to be able to verify in polynomial time the certificate of certain types of problems.
- For example:
  - Polynomial time problems.
  - Non-Polynomial time problems.

Example of verifiable problems

Hamiltonian cycle

A Hamiltonian cycle of an undirected graph $G = (V, E)$ is a simple cycle that contains each vertex in V.



As a formal language

Does a graph G have a Hamiltonian cycle?

$$HAM-CYCLES = \{\langle G\rangle \,|G \text{ is a Hamiltonian graph}\} \quad (4)$$

- How do we solve this decision problem?
- Can we even solve it?

As a formal language

Does a graph G have a Hamiltonian cycle?

$$HAM-CYCLES=\left\{\langle G\rangle\,|G \text{ is a Hamiltonian graph}\right\} \quad (4)$$

- How do we solve this decision problem?
- Can we even solve it?

As a formal language

Does a graph G have a Hamiltonian cycle?

$$HAM-CYCLES = \{\langle G\rangle\,|G \text{ is a Hamiltonian graph}\} \quad (4)$$

- How do we solve this decision problem?
- Can we even solve it?

Decision algorithm for Hamiltonian

Given an instance $< G >$ and encode it

- If we use the "reasonable" encoding of a graph as its adjacency matrix.

$$
\begin{array}{c c}
 & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \left[\begin{array}{ccccc} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{array}\right]
\end{array}
$$

Thus

We can then say the following

If the number of vertices is $m = \Omega\left(\sqrt{n}\right)$

We have then

$$
\sqrt{n}\left\{
\begin{array}{c c}
 & \overbrace{\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array}}^{\sqrt{n}} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}
\end{array}\right.
$$

Thus

We can then say the following

If the number of vertices is $m = \Omega\left(\sqrt{n}\right)$

We have then

$$\begin{array}{cc}
 & \overbrace{\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array}}^{\sqrt{n}} \\
\sqrt{n}\left\{\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\right. & \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}
\end{array}$$

Then

The encoding size is

$$\sqrt{n} \times \sqrt{n} = n = |\langle G \rangle|$$

Then, I decide to go NAIVE!!!

The algorithm does the following

It lists the all permutations of the vertices of G and then checks each permutation to see if it is a Hamiltonian path.

Complexity

Performance analysis on the previous algorithm

- We have then $m = \Omega(\sqrt{n})$
- Then, for our naive algorithm produce $m!$ permutations.
- Then $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$ **EXPONENTIAL TIME!!!**

Complexity

Performance analysis on the previous algorithm

- We have then $m = \Omega(\sqrt{n})$
- Then, for our naive algorithm produce $m!$ permutations.
- Then $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$ **EXPONENTIAL TIME!!!**

Something Notable

Still, with no-naive algorithm the Hamiltonian is not solvable in polynomial time!

Complexity

Performance analysis on the previous algorithm

- We have then $m = \Omega(\sqrt{n})$
- Then, for our naive algorithm produce $m!$ permutations.
- Then $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$ **EXPONENTIAL TIME!!!**

Something Notable

Still, with no-naive algorithm the Hamiltonian is not solvable in polynomial time!

Complexity

Performance analysis on the previous algorithm

- We have then $m = \Omega(\sqrt{n})$
- Then, for our naive algorithm produce $m!$ permutations.
- Then $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$ **EXPONENTIAL TIME!!!**

Something Notable

Still, with no-naive algorithm the Hamiltonian is not solvable in polynomial time!

Outline

Verification Algorithms

A more formal definition in terms of formal languages

A verification algorithm is a two-argument algorithm A, where:

1. There is an input string x (Instance of the problem).
2. There is a binary string y, **certificate** (The possible solution).

Verification Algorithms

A more formal definition in terms of formal languages

A verification algorithm is a two-argument algorithm A, where:

1. There is an input string x (Instance of the problem).
2. There is a binary string y, **certificate** (The possible solution).

Then, a two-argument algorithm A verifies

- A verifies x by using a certificate y.
- Then, it verifies x by taking y and outputting ONE i.e. $A(x, y) = 1$.

Verification Algorithms

A more formal definition in terms of formal languages

A verification algorithm is a two-argument algorithm A, where:

1. There is an input string x (Instance of the problem).
2. There is a binary string y, **certificate** (The possible solution).

Then, a two-argument algorithm A verifies

- A verifies x by using a certificate y.
- Then, it verifies x by taking y and outputting ONE i.e. $A(x, y) = 1$.

Verification Algorithms

A more formal definition in terms of formal languages

A verification algorithm is a two-argument algorithm A, where:

1. There is an input string x (Instance of the problem).
2. There is a binary string y, **certificate** (The possible solution).

Then, a two-argument algorithm A verifies

- A verifies x by using a certificate y.
- Then, it verifies x by taking y and outputting ONE i.e. $A(x, y) = 1$.

Verification Algorithms

A more formal definition in terms of formal languages

A verification algorithm is a two-argument algorithm A, where:

1. There is an input string x (Instance of the problem).
2. There is a binary string y, **certificate** (The possible solution).

Then, a two-argument algorithm A verifies

- A verifies x by using a certificate y.
- Then, it verifies x by taking y and outputting ONE i.e. $A(x,y) = 1$.

Finally we have

The language verified by a verification algorithm is

$$L = \{x \in \{0,1\}^* | \exists y \in 0,1^* \text{such that } A(x,y) = 1\}$$

Important remark!

For any string $x \notin L$ there must be no certificate proving $x \in L$ (consistency is a must).

Finally we have

The language verified by a verification algorithm is

$$L = \{x \in \{0,1\}^* | \exists y \in 0,1^* \text{such that } A(x,y) = 1\}$$

Important remark!

For any string $x \notin L$ there must be no certificate proving $x \in L$ (consistency is a must).

The NP class

Definition

The complexity class NP is the class of the languages that can be verified by a polynomial time algorithm.

$$L = \{x \in \{0,1\}^* | \exists y \in \{0,1\}^* \text{ with } |y| = O(|x|^c) \text{ such that } A(x,y) = 1\}$$

The NP class

Definition

The complexity class NP is the class of the languages that can be verified by a polynomial time algorithm.

$$L = \{x \in \{0,1\}^* | \exists y \in \{0,1\}^* \text{ with } |y| = O(|x|^c) \text{ such that } A(x,y) = 1\}$$

Note

- We say that A verifies language L in polynomial time.
- Clearly, the size of the certificate must be polynomial in size!!!

The NP class

Definition

The complexity class NP is the class of the languages that can be verified by a polynomial time algorithm.

$$L = \{x \in \{0,1\}^* | \exists y \in \{0,1\}^* \text{ with } |y| = O(|x|^c) \text{ such that } A(x,y) = 1\}$$

Note

- We say that A verifies language L in polynomial time.
- Clearly, the size of the certificate must be polynomial in size!!!

Observation of the class NP and co-NP

Example

- HAM-CYCLE is NP, thus NP class is not empty.
- Observation: $L \in P \to L \in NP$ or $P \subseteq NP$.

Observation of the class NP and co-NP

Example

- HAM-CYCLE is NP, thus NP class is not empty.
- Observation: $L \in P \rightarrow L \in NP$ or $P \subseteq NP$.

Now, Do we have $P = NP$?

- There is evidence that $P \neq NP$ basically because
  - the existence of languages that are NP-Complete.

Observation of the class NP and co-NP

Example

- HAM-CYCLE is NP, thus NP class is not empty.
- Observation: $L \in P \to L \in NP$ or $P \subseteq NP$.

Now, Do we have $P = NP$?

- There is evidence that $P \neq NP$ basically because
  - the existence of languages that are NP-Complete.

And actually, it is worse

- We still cannot answer if $L \in NP \to \overline{L} \in NP$ (closure under complement).

Observation of the class NP and co-NP

Example

- HAM-CYCLE is NP, thus NP class is not empty.
- Observation: $L \in P \to L \in NP$ or $P \subseteq NP$.

Now, Do we have $P = NP$?

- There is evidence that $P \neq NP$ basically because
  - the existence of languages that are NP-Complete.

And actually, it is worse

- We still cannot answer if $L \in NP \to \overline{L} \in NP$ (closure under complement).

Observation of the class NP and co-NP

Example

- HAM-CYCLE is NP, thus NP class is not empty.
- Observation: $L \in P \to L \in NP$ or $P \subseteq NP$.

Now, Do we have $P = NP$?

- There is evidence that $P \neq NP$ basically because
  - the existence of languages that are NP-Complete.

And actually, it is worse

- We still cannot answer if $L \in NP \to \bar{L} \in NP$ (closure under complement).

Another way to see this

The $co - NP$ class

The class called co-NP is the set of languages L such that $\bar{L} \in NP$

Something Notable

We can restate the question of whether NP is closed under complement as whether $NP = co - NP$

In addition because P is closed under complement

We have $P \subseteq NP \cap co - NP$, however no one knows whether $P = NP \cap co - NP$.

Another way to see this

The $co - NP$ class

The class called co-NP is the set of languages L such that $\bar{L} \in NP$

Something Notable

We can restate the question of whether NP is closed under complement as whether $NP = co - NP$

In addition because P is closed under complement

We have $P \subseteq NP \cap co - NP$, however no one knows whether $P = NP \cap co - NP$.

Another way to see this

The $co - NP$ class

The class called co-NP is the set of languages L such that $\bar{L} \in NP$

Something Notable

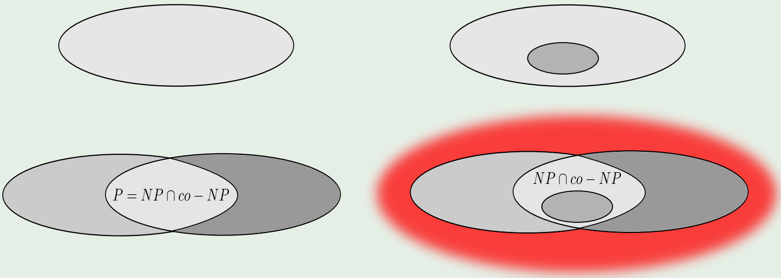
We can restate the question of whether NP is closed under complement as whether $NP = co - NP$

In addition because P is closed under complement

We have $P \subseteq NP \cap co - NP$, however no one knows whether $P = NP \cap co - NP$.

The four possibilities between the complexity classes

We have that



Exercises

From Cormen's book solve

- 34.2-1
- 34.2-2
- 34.2-5
- 34.2-6
- 34.2-9
- 34.2-10

Outline

Before entering reducibility

Why $P \neq NP$?

- Existence of NP-Complete problems.
- Problem!!! There is the following property:
  - *If any NP-Complete problem can be solved in polynomial time, then every problem in NP has a polynomial time solution.*

Before entering reducibility

Why $P \neq NP$?

- Existence of NP-Complete problems.
- Problem!!! There is the following property:
  - *If any NP-Complete problem can be solved in polynomial time, then every problem in NP has a polynomial time solution.*

Not only that

- The NP-Complete problems are the hardest in the NP class, and this is related the concept of polynomial time reducibility.

Before entering reducibility

Why $P \neq NP$?

- Existence of NP-Complete problems.
- Problem!!! There is the following property:
  - ***If any NP-Complete problem can be solved in polynomial time, then every problem in NP has a polynomial time solution.***

Not only that

- The NP-Complete problems are the hardest in the NP class, and this is related the concept of polynomial time reducibility.

Before entering reducibility

Why $P \neq NP$?

- Existence of NP-Complete problems.
- Problem!!! There is the following property:
  - ***If any NP-Complete problem can be solved in polynomial time, then every problem in NP has a polynomial time solution.***

Not only that

- The NP-Complete problems are the hardest in the NP class, and this is related the concept of polynomial time reducibility.

Reducibility

Rough definition

A problem M can be reduced to M' if any instance of M can be easily rephrased in terms of M'.

Formal definition

A language L is polynomial time reducible to a language L' written $L \leq_p L'$ if there exist a polynomial time computable function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ such that for all $x \in \{0,1\}^*$

$$x \in L \iff f(x) \in L'$$

Reducibility

Rough definition

A problem M can be reduced to M' if any instance of M can be easily rephrased in terms of M'.

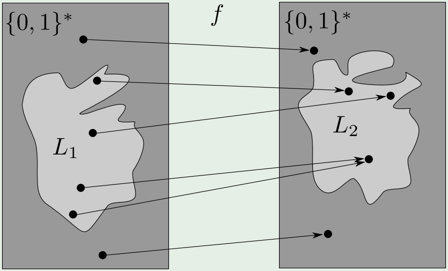
Formal definition

A language L is polynomial time reducible to a language L' written $L \leq_p L'$ if there exist a polynomial time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$

$$x \in L \iff f(x) \in L'$$

Graphically

The Mapping

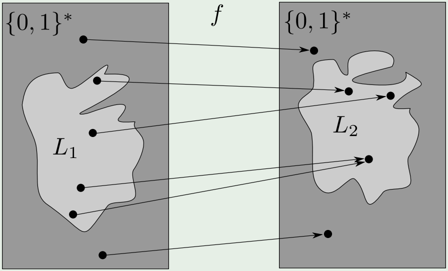


From the figure

- Here, f is called a reduction function, and the polynomial time algorithm F that computes f is called reduction algorithm.

Graphically

The Mapping



From the figure

- Here, f is called a reduction function, and the polynomial time algorithm F that computes f is called reduction algorithm.

Properties of f

Polynomial time reductions

Polynomial time reductions give us a powerful tool for proving that various languages belong to P.

How?

Lemma: If $L_1, L_2 \subseteq \{0,1\}^*$ are languages such that $L_1 \leq_p L_2$, then $L_2 \in P$ implies that $L_1 \in P$.

Proof

Properties of f

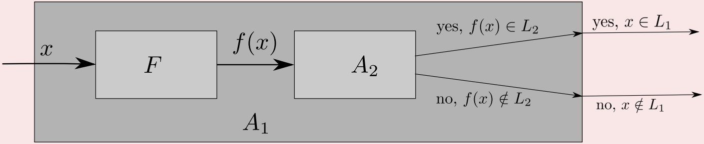
Polynomial time reductions

Polynomial time reductions give us a powerful tool for proving that various languages belong to P.

How?

Lemma: If $L_1, L_2 \subseteq \{0,1\}^*$ are languages such that $L_1 \leq_p L_2$, then $L_2 \in P$ implies that $L_1 \in P$.

Proof



Outline

NP-Completeness

Definition

A language $L \subseteq \{0,1\}^*$ is a NP-Complete problem (NPC) if:

1. $L \in NP$
2. $L' \leq_p L$ for every $L' \in NP$

Note

If a language L satisfies property 2, but not necessarily property 1, we say that L is NP-Hard (NPH).

NP-Completeness

Definition

A language $L \subseteq \{0,1\}^*$ is a NP-Complete problem (NPC) if:

1. $L \in NP$
2. $L' \leq_p L$ for every $L' \in NP$

Note

If a language L satisfies property 2, but not necessarily property 1, we say that L is NP-Hard (NPH).

By The Way

NP can also be defined as

The set of decision problems that can be solved in polynomial time on a non-deterministic Turing machine.

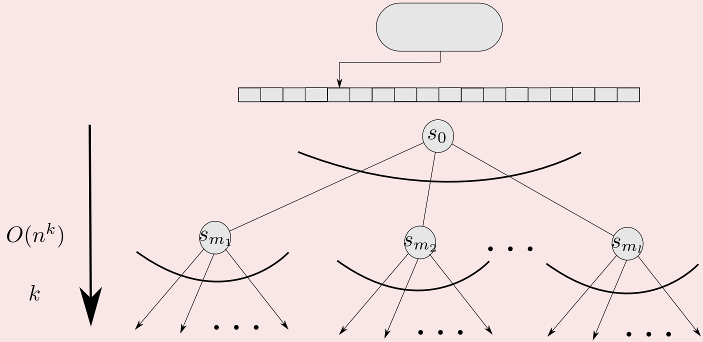
Actually, it looks like a multi-threaded backtracking

By The Way

NP can also be defined as

The set of decision problems that can be solved in polynomial time on a non-deterministic Turing machine.

Actually, it looks like a multi-threaded backtracking



Outline

Now, the infamous theorem

Theorem

If any NP-Complete problem is polynomial time solvable, then $P = NP$. Equivalently, if any problem in NP is not polynomial time solvable, then no NP-Complete problem is polynomial time solvable.

Proof

Suppose that $L \in P$ and also that $L \in NPC$. For any $L' \in NP$, we have $L' \leq_p L$ by property 2 of the definition of NPC. Thus, by the previous Lemma, we have that $L' \in P$.

Now, the infamous theorem

Theorem

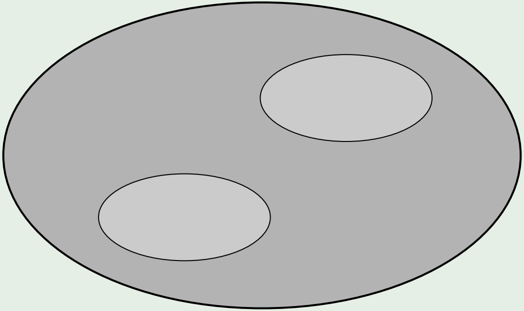
If any NP-Complete problem is polynomial time solvable, then $P = NP$. Equivalently, if any problem in NP is not polynomial time solvable, then no NP-Complete problem is polynomial time solvable.

Proof

Suppose that $L \in P$ and also that $L \in NPC$. For any $L' \in NP$, we have $L' \leq_p L$ by property 2 of the definition of NPC. Thus, by the previous Lemma, we have that $L' \in P$.

However

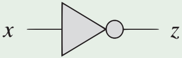
Most Theoretical Computer Scientist have the following view

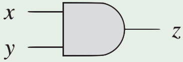


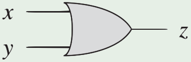
Outline

Our first NPC - Circuit Satisfiability

We have basic boolean combinatorial elements.







x	$\neg x$
0	1
1	0

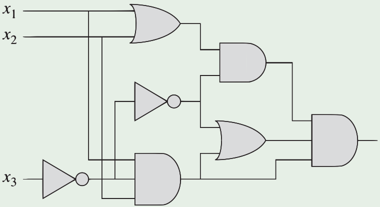
x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

Basic definition

Definition

A boolean combinatorial circuit consist of one or more boolean combinatorial elements interconnected with wires.



Circuit satisfiability problem

Problem

Given a boolean combinatorial circuit composed of AND, OR, and NOT gates, **Is it satisfiable? Output is ONE!!!**

Formally

$$CIRCUIT - SAT = \{\langle C\rangle \,|\, C \text{ is a satisfiable boolean combinatorial circuit}\}$$

Circuit satisfiability problem

Problem

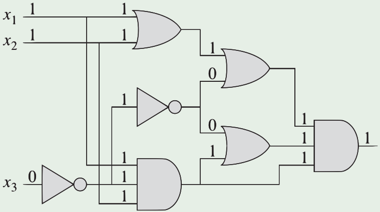
Given a boolean combinatorial circuit composed of AND, OR, and NOT gates, **Is it satisfiable? Output is ONE!!!**

Formally

$$CIRCUIT-SAT =$$
$\{\langle C\rangle \,|C$ is a satisfiable boolean combinatorial circuit$\}$

Circuit satisfiability problem

Example: An assignment that outputs ONE



Outline

First, It is NP-Problem

Lemma

The circuit-satisfiability belong to the class NP.

First, It is NP-Problem

Lemma

The circuit-satisfiability belong to the class NP.

Proof

We need to give a polynomial-time algorithm A such that

1. One of the inputs to A is a boolean combinatorial circuit C.
2. The other input is a certificate corresponding to an assignment of boolean values to the wires in C.

First, It is NP-Problem

Lemma

The circuit-satisfiability belong to the class NP.

Proof

We need to give a polynomial-time algorithm A such that

1. One of the inputs to A is a boolean combinatorial circuit C.
2. The other input is a certificate corresponding to an assignment of boolean values to the wires in C.

First, It is NP-Problem

Lemma

The circuit-satisfiability belong to the class NP.

Proof

We need to give a polynomial-time algorithm A such that

1. One of the inputs to A is a boolean combinatorial circuit C.
2. The other input is a certificate corresponding to an assignment of boolean values to the wires in C.

Second, It is NP-Hard

The general idea for A is:

- For each circuit gate check that the output value is correctly computed and corresponds to the values provided by the certificate.

Second, It is NP-Hard

The general idea for A is:

- For each circuit gate check that the output value is correctly computed and corresponds to the values provided by the certificate.

Then

- Then if the output of the entire circuit is one, the algorithm A outputs 1, otherwise 0.
- Because the certificate is polynomial in size with respect to the circuit $C \implies A$ runs in polynomial time.
  - Actually, with a good implementation, linear time is enough.

Second, It is NP-Hard

The general idea for A is:

- For each circuit gate check that the output value is correctly computed and corresponds to the values provided by the certificate.

Then

- Then if the output of the entire circuit is one, the algorithm A outputs 1, otherwise 0.
- Because the certificate is polynomial in size with respect to the circuit $C \implies A$ runs in polynomial time.
  - Actually, with a good implementation, linear time is enough.

Finally

- A cannot be fooled by any certificate to believe that a unsatisfiable circuit is accepted. Then CIRCUIT-SAT $\in$ NP.

Second, It is NP-Hard

The general idea for A is:

- For each circuit gate check that the output value is correctly computed and corresponds to the values provided by the certificate.

Then

- Then if the output of the entire circuit is one, the algorithm A outputs 1, otherwise 0.
- Because the certificate is polynomial in size with respect to the circuit $C \Longrightarrow A$ runs in polynomial time.
  - Actually, with a good implementation, linear time is enough.

Finally

- A cannot be fooled by any certificate to believe that a unsatisfiable circuit is accepted. Then CIRCUIT-SAT $\in$ NP.

Second, It is NP-Hard

The general idea for A is:

- For each circuit gate check that the output value is correctly computed and corresponds to the values provided by the certificate.

Then

- Then if the output of the entire circuit is one, the algorithm A outputs 1, otherwise 0.
- Because the certificate is polynomial in size with respect to the circuit $C \Longrightarrow A$ runs in polynomial time.
  - Actually, with a good implementation, linear time is enough.

Finally

- A cannot be fooled by any certificate to believe that a unsatisfiable circuit is accepted. Then CIRCUIT-SAT $\in$ NP.

Proving CIRCUIT-SAT is NP-Hard

Lemma

The circuit sat problem is NP-hard.

Proving CIRCUIT-SAT is NP-Hard

Lemma

The circuit sat problem is NP-hard.

Proof:

Given a language $L \in NP$, we want a polynomial-time algorithm F that can compute a reduction map f such that:

- It maps every binary string x to a circuit $C = f(x)$ such that $x \in L$ if and only if $C \in$CIRCUIT-SAT.

Proving CIRCUIT-SAT is NP-Hard

Lemma

The circuit sat problem is NP-hard.

Proof:

Given a language $L \in NP$, we want a polynomial-time algorithm F that can compute a reduction map f such that:

- It maps every binary string x to a circuit $C = f(x)$ such that $x \in L$ if and only if $C \in$CIRCUIT-SAT.

Now

First

- Given a $L \in NP$, there exists an algorithm A that verifies L in polynomial time.
- Now $T(n) = O\left(n^k\right)$ denotes the worst case time of A, and the length of the certificate is $O\left(n^k\right)$.

Now

First

- Given a $L \in NP$, there exists an algorithm A that verifies L in polynomial time.
- Now $T(n) = O\left(n^k\right)$ denotes the worst case time of A, and the length of the certificate is $O\left(n^k\right)$.

Thus

The algorithm F to be constructed will use the two input algorithm A to compute the reduction function f.

Now

First

- Given a $L \in NP$, there exists an algorithm A that verifies L in polynomial time.
- Now $T(n) = O\left(n^k\right)$ denotes the worst case time of A, and the length of the certificate is $O\left(n^k\right)$.

Thus

The algorithm F to be constructed will use the two input algorithm A to compute the reduction function f.

Basic ideas: A Computer Program

A Program

It can be seen as a sequence of instructions!!!

Basic ideas: A Computer Program

A Program

It can be seen as a sequence of instructions!!!

Each instruction

It encodes an operation to be performed, addresses of operand in memory, and a final address to store the result.

- It increments automatically upon fetching each instruction
- It can be changed by an instruction, so it can be used to implement loops and branches.

Basic ideas: A Computer Program

A Program

It can be seen as a sequence of instructions!!!

Each instruction

It encodes an operation to be performed, addresses of operand in memory, and a final address to store the result.

Each program has a counter, PC

- This counter keeps tracking of the instruction to be executed.
- It increments automatically upon fetching each instruction.
- It can be changed by an instruction, so it can be used to implement loops and branches.

Basic ideas: A Computer Program

A Program

It can be seen as a sequence of instructions!!!

Each instruction

It encodes an operation to be performed, addresses of operand in memory, and a final address to store the result.

Each program has a counter, PC

- This counter keeps tracking of the instruction to be executed.
- It increments automatically upon fetching each instruction.
- It can be changed by an instruction, so it can be used to implement loops and branches.

Basic ideas: A Computer Program

A Program

It can be seen as a sequence of instructions!!!

Each instruction

It encodes an operation to be performed, addresses of operand in memory, and a final address to store the result.

Each program has a counter, PC

- This counter keeps tracking of the instruction to be executed.
- It increments automatically upon fetching each instruction.
- It can be changed by an instruction, so it can be used to implement loops and branches.

Configuration

Something Notable

At any point during the execution of a program, the computer's memory holds the entire state of the computation.

Thus

We call any particular state of computer memory a **configuration**.

IMPORTANT

We can view the execution of an instruction as mapping one configuration to another.

Configuration

Something Notable

At any point during the execution of a program, the computer's memory holds the entire state of the computation.

Thus

We call any particular state of computer memory a **configuration**.

IMPORTANT

We can view the execution of an instruction as mapping one configuration to another.

Configuration

Something Notable

At any point during the execution of a program, the computer's memory holds the entire state of the computation.

Thus

We call any particular state of computer memory a **configuration**.

IMPORTANT

We can view the execution of an instruction as mapping one configuration to another.

Thus

We have that

The computer hardware that accomplishes this mapping can be implemented as a boolean combinational circuit.

Then

We denote this boolean circuit as M.

Thus

We have that

The computer hardware that accomplishes this mapping can be implemented as a boolean combinational circuit.

Then

We denote this boolean circuit as M.

What we want

Let L be any language in NP

There must exist an algorithm A that verifies L in polynomial time

Thus

The algorithm F that we shall construct uses the two-input algorithm A to compute the reduction function f.

- Remember
  - The running time of A is actually a polynomial in the total input size, which includes both an input string and a certificate.
  - The certificate is polynomial in the length n of the input.
    - Thus the running time is polynomial in n.

What we want

Let L be any language in NP

There must exist an algorithm A that verifies L in polynomial time

Thus

The algorithm F that we shall construct uses the two-input algorithm A to compute the reduction function f.

Now

Let $T(n) = O(n^k)$ denote the worst-case running time of algorithm A on a n-length input for some $k \geq 1$.

- Remember
  - The running time of A is actually a polynomial in the total input size which includes both an input string and a certificate.
  - The certificate is polynomial in the length n of the input.
    - Thus the running time is polynomial in n.

What we want

Let L be any language in NP

There must exist an algorithm A that verifies L in polynomial time

Thus

The algorithm F that we shall construct uses the two-input algorithm A to compute the reduction function f.

Now

Let $T(n) = O\left(n^k\right)$ denote the worst-case running time of algorithm A on a n-length input for some $k \geq 1$.

- Remember:
  - The running time of A is actually a polynomial in the total input size, which includes both an input string and a certificate.
  - The certificate is polynomial in the length n of the input.
    - **Thus the running time is polynomial in n.**

What we want

Let L be any language in NP

There must exist an algorithm A that verifies L in polynomial time

Thus

The algorithm F that we shall construct uses the two-input algorithm A to compute the reduction function f.

Now

Let $T(n) = O\left(n^k\right)$ denote the worst-case running time of algorithm A on a n-length input for some $k \geq 1$.

- Remember:
  - The running time of A is actually a polynomial in the total input size, which includes both an input string and a certificate.
  - The certificate is polynomial in the length n of the input.
    - Thus the running time is polynomial in n.

What we want

Let L be any language in NP

There must exist an algorithm A that verifies L in polynomial time

Thus

The algorithm F that we shall construct uses the two-input algorithm A to compute the reduction function f.

Now

Let $T(n) = O\left(n^k\right)$ denote the worst-case running time of algorithm A on a n-length input for some $k \geq 1$.

- Remember:
  - The running time of A is actually a polynomial in the total input size, which includes both an input string and a certificate.
  - The certificate is polynomial in the length n of the input.
    - Thus the running time is polynomial in n.

What we want

Let L be any language in NP

There must exist an algorithm A that verifies L in polynomial time

Thus

The algorithm F that we shall construct uses the two-input algorithm A to compute the reduction function f.

Now

Let $T(n) = O\left(n^k\right)$ denote the worst-case running time of algorithm A on a n-length input for some $k \geq 1$.

- Remember:
  - The running time of A is actually a polynomial in the total input size, which includes both an input string and a certificate.
  - The certificate is polynomial in the length n of the input.
    - **Thus the running time is polynomial in n.**

Outline

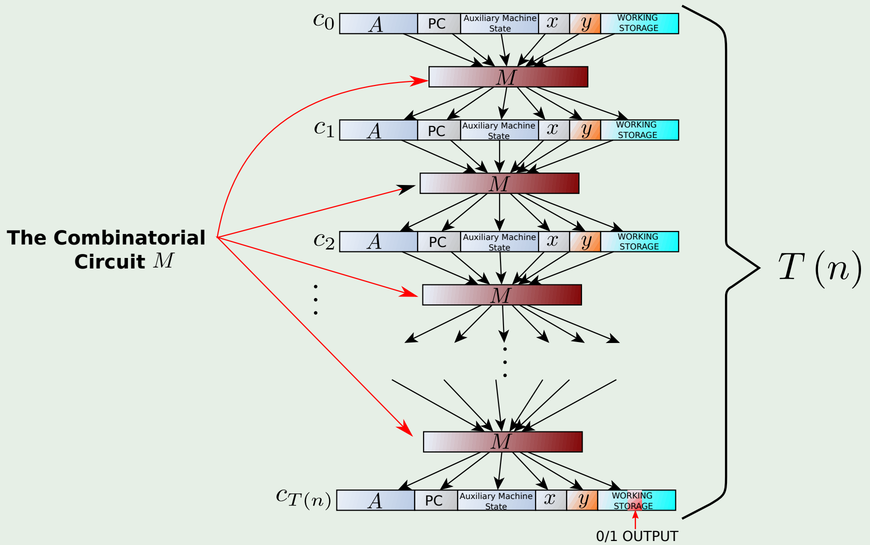
Basic ideas: Algorithm A representation

We can represent the computation of A as a sequence of configurations.

- Start with configuration c_0, then finish with configuration $c_{T(n)}$.

Then

The idea



Basic ideas: Algorithm A representation

Then, we need $C = f(x)$

- For this, we do:
  - $n = |x|$

Basic ideas: Algorithm A representation

Then, we need $C = f(x)$

- For this, we do:
  - $n = |x|$

Next

Then, we constructs a combinatorial circuit C' consisting of $T(n)$ copies of M

- The output of the c_i circuit finish as input in c_{i+1}.

Basic ideas: Algorithm A representation

Then, we need $C = f(x)$

- For this, we do:
  - $n = |x|$

Next

Then, we constructs a combinatorial circuit C' consisting of $T(n)$ copies of M

- The output of the c_i circuit finish as input in c_{i+1}.

Remark

The configuration finishes as values on the wires connecting copies.

Basic ideas: Algorithm A representation

Then, we need $C = f(x)$

- For this, we do:
  - $n = |x|$

Next

Then, we constructs a combinatorial circuit C' consisting of $T(n)$ copies of M

- The output of the c_i circuit finish as input in c_{i+1}.

Remark

The configuration finishes as values on the wires connecting copies.

The polynomial time algorithm

What F must do:

1. Given x it needs to compute circuit $C(x) = f(x)$.
2. Satisfiable $\iff$ there exists a certificate y such that $A(x, y) = 1$.

The polynomial time algorithm

What F must do:

1. Given x it needs to compute circuit $C(x) = f(x)$.
2. Satisfiable $\iff$ there exists a certificate y such that $A(x, y) = 1$.

The F process

Given x:

- It first computes $n = |x|$.

Next

- Then it computes C' (a combinatorial circuit) by using $T(n)$ copies of M.

Then

- Then, the initial configuration of C' consists in the input $A(x, y)$, the output is configuration $C_{T(n)}$

The F process

Given x:

- It first computes $n = |x|$.

Next

- Then it computes C' (a combinatorial circuit) by using $T(n)$ copies of M.

Then

- Then, the initial configuration of C' consists in the input $A(x, y)$, the output is configuration $C_{T(n)}$

The F process

Given x:

- It first computes $n = |x|$.

Next

- Then it computes C' (a combinatorial circuit) by using $T(n)$ copies of M.

Then

- Then, the initial configuration of C' consists in the input $A(x, y)$, the output is configuration $C_{T(n)}$

Finally C

We then use C' to construct C

- First, F modifies circuit C' in the following way:
  - It hardwires the inputs to C' corresponding to the program for A:
    - The initial program counter
    - The input x
    - The initial state of memory

Finally C

We then use C' to construct C

- First, F modifies circuit C' in the following way:
  - It hardwires the inputs to C' corresponding to the program for A:
    - The initial program counter
    - The input x
    - The initial state of memory

Finally C

We then use C' to construct C

- First, F modifies circuit C' in the following way:
  - It hardwires the inputs to C' corresponding to the program for A:
    - The initial program counter
    - The input x
    - The initial state of memory

Finally C

We then use C' to construct C

- First, F modifies circuit C' in the following way:
  - It hardwires the inputs to C' corresponding to the program for A:
    - ⋆ The initial program counter
    - ⋆ The input x
    - ⋆ The initial state of memory

Finally C

We then use C' to construct C

- First, F modifies circuit C' in the following way:
  - It hardwires the inputs to C' corresponding to the program for A:
    - The initial program counter
    - The input x
    - The initial state of memory

Further

Something Notable

The only remaining inputs to the circuit correspond to the certificate y.

Then

- All outputs to the circuit are ignored, except the one bit of $c_{T(n)}$ corresponding to a computation on $A(x, y)$.

Because the only free input is the certificate y

Ah!! We have that $C(y) = A(x, y)$!!!

Further

Something Notable

The only remaining inputs to the circuit correspond to the certificate y.

Then

- All outputs to the circuit are ignored, except the one bit of $c_{T(n)}$ corresponding to a computation on $A(x, y)$.

Because the only free input is the certificate y

Ah!! We have that $C(y) = A(x, y)$!!!

Further

Something Notable

The only remaining inputs to the circuit correspond to the certificate y.

Then

- All outputs to the circuit are ignored, except the one bit of $c_{T(n)}$ corresponding to a computation on $A(x,y)$.

Because the only free input is the certificate y

Ah!! We have that $C(y) = A(x,y)$!!!

Outline

What we need to prove

First

- F correctly computes a reduction function f.
  - **C is satisfiable if and only if there is a certificate y such that $A(x, y) = 1$.**

Second

- We need to show that F runs in polynomial time.

What we need to prove

First

- F correctly computes a reduction function f.
  - **C is satisfiable if and only if there is a certificate y such that $A(x, y) = 1$.**

Second

- We need to show that F runs in polynomial time.

First, F correctly computes a reduction function f

We do the following

To show that F correctly computes a reduction function, let us suppose that there exists a certificate y of length $O(n^k)$ such that $A(x, y) = 1$.

$\Longleftarrow$

- If we apply the bits of y to the inputs of C, the output of C is $C(y) = A(x, y) = 1$. Thus, if a certificate exists, then C is satisfiable.

$\Longrightarrow$

- Now, suppose that C is satisfiable. Hence, there exists an input y to C such that $C(y) = 1$, from which we conclude that $A(x, y) = 1$.

First, F correctly computes a reduction function f

We do the following

To show that F correctly computes a reduction function, let us suppose that there exists a certificate y of length $O(n^k)$ such that $A(x, y) = 1$.

$\Longleftarrow$

- If we apply the bits of y to the inputs of C, the output of C is $C(y) = A(x, y) = 1$. Thus, if a certificate exists, then C is satisfiable.

$\Longrightarrow$

- Now, suppose that C is satisfiable. Hence, there exists an input y to C such that $C(y) = 1$, from which we conclude that $A(x, y) = 1$.

First, F correctly computes a reduction function f

We do the following

To show that F correctly computes a reduction function, let us suppose that there exists a certificate y of length $O(n^k)$ such that $A(x,y)=1$.

$\Longleftarrow$

- If we apply the bits of y to the inputs of C, the output of C is $C(y)=A(x,y)=1$. Thus, if a certificate exists, then C is satisfiable.

$\Longrightarrow$

- Now, suppose that C is satisfiable. Hence, there exists an input y to C such that $C(y)=1$, from which we conclude that $A(x,y)=1$.

Outline

Next, we need to show that F runs in polynomial time

With respect to the polynomial reduction

The length of the input x is n, and the certificate y is $O(n^k)$.

Next

Circuit M implementing the computer hardware has polynomial size.

Properties

The circuit C consists of at most $t = O\left(n^k\right)$ copies of M.

Next, we need to show that F runs in polynomial time

With respect to the polynomial reduction

The length of the input x is n, and the certificate y is $O(n^k)$.

Next

Circuit M implementing the computer hardware has polynomial size.

Properties

The circuit C consists of at most $t = O\left(n^k\right)$ copies of M.

Next, we need to show that F runs in polynomial time

With respect to the polynomial reduction

The length of the input x is n, and the certificate y is $O(n^k)$.

Next

Circuit M implementing the computer hardware has polynomial size.

Properties

The circuit C consists of at most $t = O\left(n^k\right)$ copies of M.

Finally!!!

In conclusion

The language CIRCUIT-SAT is therefore at least as hard as any language in NP, and since it belongs to NP, it is NP-complete.

Theorem

The circuit satisfiability problem is NP-Complete.

Finally!!!

In conclusion

The language CIRCUIT-SAT is therefore at least as hard as any language in NP, and since it belongs to NP, it is NP-complete.

Theorem

The circuit satisfiability problem is NP-Complete.

Outline

Proving NP-Complete

Several theorems exist to make our life easier

We have the following

Lemma

If L is a Language such that $L' \leq_P L$ for some $L' \in NPC$, Then L is NP-Hard. Moreover, if $L \in NP$, then $L \in NPC$.

Proving NP-Complete

Several theorems exist to make our life easier

We have the following

Lemma

If L is a Language such that $L' \leq_P L$ for some $L' \in NPC$, Then L is NP-Hard. Moreover, if $L \in NP$, then $L \in NPC$.

So, we have the following strategy

Proceed as follows for proving that something is NP-Complete

1. Prove $L \in NP$.
2. Select a known NP-Complete language L'.
3. Describe an algorithm that computes function f mapping every instance $x \in \{0,1\}^*$ of L' to an instance $f(x)$ of L.
4. Prove that the function f satisfies: $x \in L'$ if and only if $f(x) \in L$ L for all $x \in \{0,1\}^*$.
5. Prove the polynomial time of the algorithm.

So, we have the following strategy

Proceed as follows for proving that something is NP-Complete

1. Prove $L \in NP$.
2. Select a known NP-Complete language L'.
3. Describe an algorithm that computes function f mapping every instance $x \in \{0,1\}^*$ of L' to an instance $f(x)$ of L.
4. Prove that the function f satisfies: $x \in L'$ if and only if $f(x) \in L$ L for all $x \in \{0,1\}^*$.
5. Prove the polynomial time of the algorithm.

So, we have the following strategy

Proceed as follows for proving that something is NP-Complete

1. Prove $L \in NP$.
2. Select a known NP-Complete language L'.
3. Describe an algorithm that computes function f mapping every instance $x \in \{0,1\}^*$ of L' to an instance $f(x)$ of L.
4. Prove that the function f satisfies: $x \in L'$ if and only if $f(x) \in L$ L for all $x \in \{0,1\}^*$.
5. Prove the polynomial time of the algorithm.

So, we have the following strategy

Proceed as follows for proving that something is NP-Complete

1. Prove $L \in NP$.
2. Select a known NP-Complete language L'.
3. Describe an algorithm that computes function f mapping every instance $x \in \{0,1\}^*$ of L' to an instance $f(x)$ of L.
4. Prove that the function f satisfies: $x \in L'$ if and only if $f(x) \in L$ L for all $x \in \{0,1\}^*$.
5. Prove the polynomial time of the algorithm.

So, we have the following strategy

Proceed as follows for proving that something is NP-Complete

1. Prove $L \in NP$.
2. Select a known NP-Complete language L'.
3. Describe an algorithm that computes function f mapping every instance $x \in \{0,1\}^*$ of L' to an instance $f(x)$ of L.
4. Prove that the function f satisfies: $x \in L'$ if and only if $f(x) \in L$ L for all $x \in \{0,1\}^*$.
5. Prove the polynomial time of the algorithm.

Exercises

From Cormen's book solve

- 34.3-1
- 34.3-2
- 34.3-5
- 34.3-6
- 34.3-7
- 34.3-8

Outline

Formula Satisfiability (SAT)

An instance of SAT is a boolean formula ϕ composed of

1. n boolean variables $x_1, x_2, ..., x_n$.
2. m boolean connectives ($\wedge$,$\vee$, $\neg$, $\rightarrow$, $\iff$).
3. Parentheses.

Formula Satisfiability (SAT)

An instance of SAT is a boolean formula ϕ composed of

1. n boolean variables $x_1, x_2, ..., x_n$.
2. m boolean connectives $(\wedge\ ,\vee,\ \neg,\ \rightarrow,\ \iff)$.
3. Parentheses.

A small example

$$\phi = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

Formula Satisfiability (SAT)

An instance of SAT is a boolean formula ϕ composed of

1. n boolean variables $x_1, x_2, ..., x_n$.
2. m boolean connectives ($\wedge$,$\vee$, $\neg$, $\rightarrow$, $\iff$).
3. Parentheses.

A small example

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

Formula Satisfiability (SAT)

An instance of SAT is a boolean formula ϕ composed of

1. n boolean variables $x_1, x_2, ..., x_n$.
2. m boolean connectives ($\wedge$,$\vee$, $\neg$, $\rightarrow$, $\iff$).
3. Parentheses.

A small example

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

The satisfiability problem asks whether a given boolean formula is satisfiable

In formal-language terms

$$\text{SAT} = \{\langle\phi\rangle \,|\phi \text{ is a satisfiable boolean formula}\}$$

The satisfiability problem asks whether a given boolean formula is satisfiable

In formal-language terms

$$\text{SAT} = \{\langle \phi \rangle \,|\phi \text{ is a satisfiable boolean formula}\}$$

Example: Given $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1\rangle$

$$\begin{aligned}\phi &= ((0 \to 0) \lor \neg((\neg 0 \leftrightarrow 1) \lor 1)) \land \neg 0 \\ &= (1 \lor \neg(1 \lor 1)) \land 1 \\ &= (1 \lor 0) \land 1 \\ &= 1\end{aligned}$$

The satisfiability problem asks whether a given boolean formula is satisfiable

In formal-language terms

$$\text{SAT} = \{\langle \phi \rangle \,|\phi \text{ is a satisfiable boolean formula}\}$$

Example: Given $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1 \rangle$

$$\begin{aligned} \phi &= ((0 \to 0) \lor \neg((\neg 0 \leftrightarrow 1) \lor 1)) \land \neg 0 \\ &= (1 \lor \neg (1 \lor 1)) \land 1 \\ &= (1 \lor 0) \land 1 \\ &= 1 \end{aligned}$$

The satisfiability problem asks whether a given boolean formula is satisfiable

In formal-language terms

$$\text{SAT} = \{\langle \phi \rangle \,|\, \phi \text{ is a satisfiable boolean formula}\}$$

Example: Given $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1 \rangle$

$$\begin{aligned}\phi &= ((0 \to 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0 \\ &= (1 \vee \neg (1 \vee 1)) \wedge 1 \\ &= (1 \vee 0) \wedge 1 \\ &= 1\end{aligned}$$

The satisfiability problem asks whether a given boolean formula is satisfiable

In formal-language terms

$$\text{SAT} = \{\langle\phi\rangle \,|\phi \text{ is a satisfiable boolean formula}\}$$

Example: Given $\langle x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1\rangle$

$$\begin{aligned}\phi &= ((0 \to 0) \vee \neg((\neg 0 \leftrightarrow 1) \vee 1)) \wedge \neg 0\\ &= (1 \vee \neg\,(1 \vee 1)) \wedge 1\\ &= (1 \vee 0) \wedge 1\\ &= 1\end{aligned}$$

Formula Satisfiability

Theorem

Satisfiability of boolean formulas is NP-Complete.

Formula Satisfiability

Theorem

Satisfiability of boolean formulas is NP-Complete.

Proof

1. The NP part is easy.
2. Now, the mapping from a NPC.

Formula Satisfiability

Theorem

Satisfiability of boolean formulas is NP-Complete.

Proof

1. The NP part is easy.
2. Now, the mapping from a NPC.

Showing that SAT belongs to NP

Certificate

It consists of a satisfying assignment for an input formula ϕ.

Then A does the following

The verifying algorithm simply replaces each variable in the formula with its responding value and then evaluates the expression.

- If the expression evaluates to 1, then the algorithm has verified that the formula is satisfiable.

Showing that SAT belongs to NP

Certificate

It consists of a satisfying assignment for an input formula ϕ.

Then A does the following

The verifying algorithm simply replaces each variable in the formula with its responding value and then evaluates the expression.

Properties

- This task is easy to do in polynomial time.
- If the expression evaluates to 1, then the algorithm has verified that the formula is satisfiable.

Showing that SAT belongs to NP

Certificate

It consists of a satisfying assignment for an input formula ϕ.

Then A does the following

The verifying algorithm simply replaces each variable in the formula with its responding value and then evaluates the expression.

Properties

- This task is easy to do in polynomial time.
- If the expression evaluates to 1, then the algorithm has verified that the formula is satisfiable.

Now, we try the mapping from CIRCUIT-SAT to SAT

Naïve algorithm

- We can use induction to express any boolean combinational circuit as a boolean formula.

Then

- We simply look at the gate that produces the circuit output and inductively express each of the gate's inputs as formulas.

Naively

- We then obtain the formula for the circuit by writing an expression that applies the gate's function to its inputs' formulas.

Now, we try the mapping from CIRCUIT-SAT to SAT

Naïve algorithm

- We can use induction to express any boolean combinational circuit as a boolean formula.

Then

- We simply look at the gate that produces the circuit output and inductively express each of the gate's inputs as formulas.

Naively

- We then obtain the formula for the circuit by writing an expression that applies the gate's function to its inputs' formulas.

Now, we try the mapping from CIRCUIT-SAT to SAT

Naïve algorithm

- We can use induction to express any boolean combinational circuit as a boolean formula.

Then

- We simply look at the gate that produces the circuit output and inductively express each of the gate's inputs as formulas.

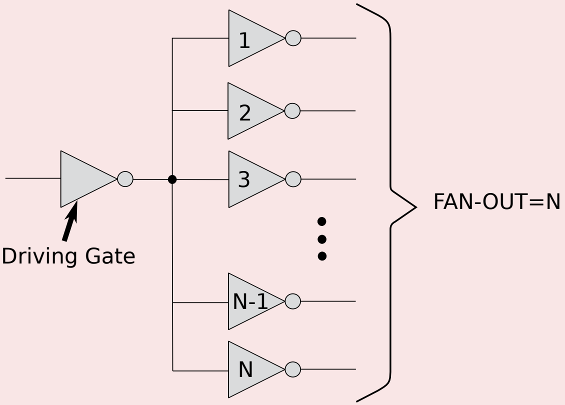
Naively

- We then obtain the formula for the circuit by writing an expression that applies the gate's function to its inputs' formulas.

Problem

PROBLEM

What happens if the circuit fan out? I.e. shared sub-formulas can make the expression to grow exponentially!!!



Instead, we use the following strategy

First

- For each wire x_i in the circuit C , the formula has a variable x_i
- Remember you need the last wire to be true!!!

Instead, we use the following strategy

First

- For each wire x_i in the circuit C , the formula has a variable x_i
- Remember you need the last wire to be true!!!

Then

We can now express how each gate operates as a small formula involving the variables of its incident wires.

Instead, we use the following strategy

First

- For each wire x_i in the circuit C , the formula has a variable x_i
- Remember you need the last wire to be true!!!

Then

We can now express how each gate operates as a small formula involving the variables of its incident wires.

Actually, we build a sequence of tautologies

$$x_{10} \longleftrightarrow (x_7 \wedge x_8 \wedge x_9)$$

- We call each of these small formulas a clause.

Instead, we use the following strategy

First

- For each wire x_i in the circuit C , the formula has a variable x_i
- Remember you need the last wire to be true!!!

Then

We can now express how each gate operates as a small formula involving the variables of its incident wires.

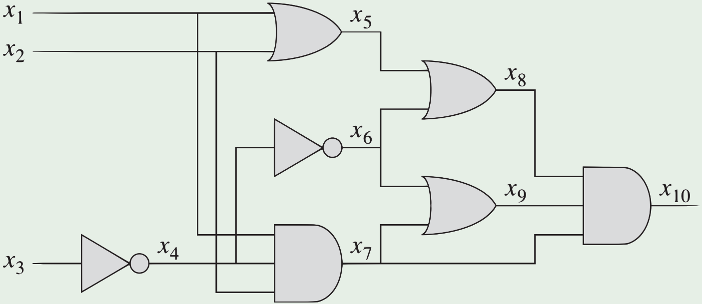
Actually, we build a sequence of tautologies

$$x_{10} \longleftrightarrow (x_7 \wedge x_8 \wedge x_9)$$

- We call each of these small formulas a clause.

Use CIRCUIT-SAT

We a circuit C $\in$CIRCUIT-SAT



Thus, we have the following clauses

The new boolean formula

$$
\begin{aligned}
\phi = x_{10} &\wedge (x_4 \leftrightarrow \neg x_3)\\
&\wedge (x_5 \leftrightarrow (x_1 \vee x_2))\\
&\wedge (x_6 \leftrightarrow \neg x_4)\\
&\wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4))\\
&\wedge (x_8 \leftrightarrow (x_5 \vee x_6))\\
&\wedge (x_9 \leftrightarrow (x_6 \vee x_7))\\
&\wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))
\end{aligned}
$$

Thus, we have the following clauses

The new boolean formula

$$
\begin{aligned}
\phi = x_{10} &\wedge (x_4 \leftrightarrow \neg x_3) \\
&\wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \\
&\wedge (x_6 \leftrightarrow \neg x_4) \\
&\wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\
&\wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \\
&\wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \\
&\wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))
\end{aligned}
$$

Thus, we have the following clauses

The new boolean formula

$$
\begin{aligned}
\phi = x_{10} & \wedge (x_4 \leftrightarrow \neg x_3) \\
& \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \\
& \wedge (x_6 \leftrightarrow \neg x_4) \\
& \wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\
& \wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \\
& \wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \\
& \wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))
\end{aligned}
$$

Thus, we have the following clauses

The new boolean formula

$$
\begin{aligned}
\phi = x_{10} & \wedge (x_4 \leftrightarrow \neg x_3) \\
& \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \\
& \wedge (x_6 \leftrightarrow \neg x_4) \\
& \wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\
& \wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \\
& \wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \\
& \wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))
\end{aligned}
$$

Thus, we have the following clauses

The new boolean formula

$$
\begin{aligned}
\phi = x_{10} & \wedge (x_4 \leftrightarrow \neg x_3) \\
& \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \\
& \wedge (x_6 \leftrightarrow \neg x_4) \\
& \wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\
& \wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \\
& \wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \\
& \wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))
\end{aligned}
$$

Thus, we have the following clauses

The new boolean formula

$$
\begin{aligned}
\phi = x_{10} & \wedge (x_4 \leftrightarrow \neg x_3) \\
& \wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \\
& \wedge (x_6 \leftrightarrow \neg x_4) \\
& \wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\
& \wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \\
& \wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \\
& \wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))
\end{aligned}
$$

Thus, we have the following clauses

The new boolean formula

$$\begin{aligned}\phi = x_{10} &\wedge (x_4 \leftrightarrow \neg x_3)\\ &\wedge (x_5 \leftrightarrow (x_1 \vee x_2))\\ &\wedge (x_6 \leftrightarrow \neg x_4)\\ &\wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4))\\ &\wedge (x_8 \leftrightarrow (x_5 \vee x_6))\\ &\wedge (x_9 \leftrightarrow (x_6 \vee x_7))\\ &\wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))\end{aligned}$$

Furthermore

Something Notable

Given that the circuit C is polynomial in size:

- it is straightforward to produce such a formula ϕ in polynomial time.

What do we want to prove?

Then

We want the following

$\Longrightarrow$

If C has a satisfying assignment then ϕ is satisfiable.

$\Longleftarrow$

If some assignment causes ϕ to evaluate to 1 then C is satisfiable.

What do we want to prove?

Then

We want the following

$\Longrightarrow$

If C has a satisfying assignment then ϕ is satisfiable.

$\Longleftarrow$

If some assignment causes ϕ to evaluate to 1 then C is satisfiable.

What do we want to prove?

Then

We want the following

$\Longrightarrow$

If C has a satisfying assignment then ϕ is satisfiable.

$\Longleftarrow$

If some assignment causes ϕ to evaluate to 1 then C is satisfiable.

Then

Satisfiable

If C has a satisfying assignment, then each wire of the circuit has a well-defined value, and the output of the circuit is 1.

Meaning

Therefore, when we assign wire values to variables in ϕ, each clause of ϕ evaluates to 1, and thus the conjunction of all evaluates to 1.

Conversely

Conversely, if some assignment causes ϕ to evaluate to 1, the circuit C is satisfiable by an analogous argument.

Then

Satisfiable

If C has a satisfying assignment, then each wire of the circuit has a well-defined value, and the output of the circuit is 1.

Meaning

Therefore, when we assign wire values to variables in ϕ, each clause of ϕ evaluates to 1, and thus the conjunction of all evaluates to 1.

Conversely

Conversely, if some assignment causes ϕ to evaluate to 1, the circuit C is satisfiable by an analogous argument.

Then

Satisfiable

If C has a satisfying assignment, then each wire of the circuit has a well-defined value, and the output of the circuit is 1.

Meaning

Therefore, when we assign wire values to variables in ϕ, each clause of ϕ evaluates to 1, and thus the conjunction of all evaluates to 1.

Conversely

Conversely, if some assignment causes ϕ to evaluate to 1, the circuit C is satisfiable by an analogous argument.

Then

We have proved that

$CIRCUIT - SAT \leq_p SAT$

Outline

However!

However!

Problem: SAT is still too complex.

Solution: Use 3-CNF

However!

However!

Problem: SAT is still too complex.

Solution: Use 3-CNF

Definition

First

- A literal in a boolean formula is an occurrence of a variable or its negation.

Second

- A boolean formula is in conjunctive normal form, or CNF, if it is expressed as an AND of clauses, each of which is the OR of one or more literals.

Third

- A boolean formula is in 3-Conjunctive normal form, or 3-CNF, if each clause has exactly three distinct literals.

$$(x_1 \vee \neg \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

Definition

First

- A literal in a boolean formula is an occurrence of a variable or its negation.

Second

- A boolean formula is in conjunctive normal form, or CNF, if it is expressed as an AND of clauses, each of which is the OR of one or more literals.

Third

- A boolean formula is in 3-Conjunctive normal form, or 3-CNF, if each clause has exactly three distinct literals.

$$(x_1 \vee \neg \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

Definition

First

- A literal in a boolean formula is an occurrence of a variable or its negation.

Second

- A boolean formula is in conjunctive normal form, or CNF, if it is expressed as an AND of clauses, each of which is the OR of one or more literals.

Third

- A boolean formula is in 3-Conjunctive normal form, or 3-CNF, if each clause has exactly three distinct literals.

$$(x_1 \vee \neg \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

3-CNF is NP-Complete

Theorem

Satisfiability of boolean formulas in 3-Conjunctive normal form is NP-Complete.

3-CNF is NP-Complete

Theorem

Satisfiability of boolean formulas in 3-Conjunctive normal form is NP-Complete.

Proof

- The NP part is similar to the previous theorem.
- The interesting part is proving that SAT $\leq_p$ 3-CNF

3-CNF is NP-Complete

Theorem

Satisfiability of boolean formulas in 3-Conjunctive normal form is NP-Complete.

Proof

- The NP part is similar to the previous theorem.
- The interesting part is proving that SAT$\leq_p$ 3-CNF

Proof NP-Complete of 3-CNF

Parse the formula

Example: $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$

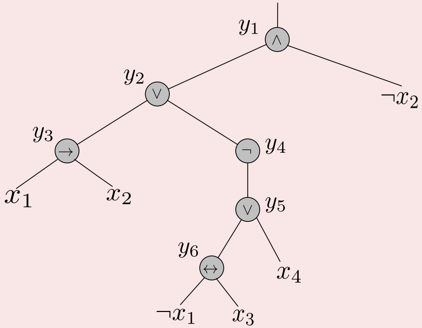
We can use ideas from parsing to create a syntax tree

Proof NP-Complete of 3-CNF

Parse the formula

Example: $\phi = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$

We can use ideas from parsing to create a syntax tree



Parsing allows us to parse the formula into ϕ'

This can be done by naming the nodes in the tree

$$
\begin{aligned}
\phi' = y_1 &\wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\
&\wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\
&\wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\
&\wedge (y_4 \leftrightarrow \neg y_5) \\
&\wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\
&\wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))
\end{aligned}
$$

Parsing allows us to parse the formula into ϕ'

This can be done by naming the nodes in the tree

$$
\begin{aligned}
\phi' = y_1 &\wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\
&\wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\
&\wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\
&\wedge (y_4 \leftrightarrow \neg y_5) \\
&\wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\
&\wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))
\end{aligned}
$$

Problem

We still not have the disjunctive parts... What can we do?

Parsing allows us to parse the formula into ϕ'

This can be done by naming the nodes in the tree

$$
\begin{aligned}
\phi' = y_1 &\wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\
&\wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\
&\wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\
&\wedge (y_4 \leftrightarrow \neg y_5) \\
&\wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\
&\wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))
\end{aligned}
$$

Problem

We still not have the disjunctive parts... What can we do?

Parsing allows us to parse the formula into ϕ'

This can be done by naming the nodes in the tree

$$
\begin{aligned}
\phi' = y_1 & \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\
& \wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\
& \wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\
& \wedge (y_4 \leftrightarrow \neg y_5) \\
& \wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\
& \wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))
\end{aligned}
$$

Problem

We still not have the disjunctive parts... What can we do?

Parsing allows us to parse the formula into ϕ'

This can be done by naming the nodes in the tree

$$
\begin{aligned}
\phi' = y_1 &\wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\
&\wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\
&\wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\
&\wedge (y_4 \leftrightarrow \neg y_5) \\
&\wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\
&\wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))
\end{aligned}
$$

Problem

We still not have the disjunctive parts... What can we do?

Parsing allows us to parse the formula into ϕ'

This can be done by naming the nodes in the tree

$$
\begin{aligned}
\phi' = y_1 &\wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\
&\wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\
&\wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\
&\wedge (y_4 \leftrightarrow \neg y_5) \\
&\wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\
&\wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))
\end{aligned}
$$

Problem

We still not have the disjunctive parts... What can we do?

Parsing allows us to parse the formula into ϕ'

This can be done by naming the nodes in the tree

$$\begin{aligned}\phi' = y_1 &\wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2))\\ &\wedge (y_2 \leftrightarrow (y_3 \vee y_4))\\ &\wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2))\\ &\wedge (y_4 \leftrightarrow \neg y_5)\\ &\wedge (y_5 \leftrightarrow (y_6 \vee x_4))\\ &\wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))\end{aligned}$$

Problem

We still not have the disjunctive parts... What can we do?

Proof

We can do the following

We can build the truth table of each clause ϕ'_i!

For example, the truth table of $\phi'_1 = y_1 \leftrightarrow (y_2 \wedge \neg x_2)$

y_1	y_2	x_3	$y_1 \leftrightarrow (y_2 \wedge \neg x_2)$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

Proof

We can do the following

We can build the truth table of each clause ϕ'_i!

For example, the truth table of $\phi'_1 = y_1 \leftrightarrow (y_2 \wedge \neg x_2)$

y_1	y_2	x_3	$y_1 \leftrightarrow (y_2 \wedge \neg x_2)$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

From this, we have

Disjunctive normal form (or DNF)

In each of the zeros we put a conjunction that evaluate to ONE

y_1	y_2	x_3	$y_1 \leftrightarrow (y_2 \land \neg x_2)$	
1	1	1	0	$y_1 \land y_2 \land x_3$
1	1	0	1	$\cdots$
1	0	1	0	$y_1 \land \neg y_2 \land x_3$
1	0	0	0	$y_1 \land \neg y_2 \land \neg x_3$
0	1	1	1	$\cdots$
0	1	0	0	$\neg y_1 \land y_2 \land \neg x_3$
0	0	1	1	$\cdots$
0	0	0	1	$\cdots$

Then, we use disjunctions to put all them together

We have then an OR of AND's

$$I=(y_1\wedge y_2\wedge x_3)\vee(y_1\wedge\neg y_2\wedge x_3)\vee(y_1\wedge\neg y_2\wedge\neg x_3)\vee(\neg y_1\wedge y_2\wedge\neg x_3)$$

Thus, we have that $\neg I\equiv\phi_1'$

y_1	y_2	x_3	$y_1\leftrightarrow(y_2\wedge\neg x_2)$	I	$\neg I$
1	1	1	0	1	0
1	1	0	1	0	1
1	0	1	0	1	0
1	0	0	0	1	0
0	1	1	1	0	1
0	1	0	0	1	0
0	0	1	1	0	1
0	0	0	1	0	1

Then, we use disjunctions to put all them together

We have then an OR of AND's

$$I=(y_1\land y_2\land x_3)\lor(y_1\land\neg y_2\land x_3)\lor(y_1\land\neg y_2\land\neg x_3)\lor(\neg y_1\land y_2\land\neg x_3)$$

Thus, we have that $\neg I\equiv\phi_1'$

y_1	y_2	x_3	$y_1\leftrightarrow(y_2\land\neg x_2)$	I	$\neg I$
1	1	1	0	1	0
1	1	0	1	0	1
1	0	1	0	1	0
1	0	0	0	1	0
0	1	1	1	0	1
0	1	0	0	1	0
0	0	1	1	0	1
0	0	0	1	0	1

Using DeMorgan's laws

We obtain

$$\phi_1'' = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$$

Now, we need to include more literals as necessary

Given C_i as a disjunctive part of the previous formula.

- If C_i has 3 distinct literals, then simply include C_i as a clause of ϕ.

Now, we need to include more literals as necessary

Given C_i as a disjunctive part of the previous formula.

- If C_i has 3 distinct literals, then simply include C_i as a clause of ϕ.

If C_i has 2 distinct literals

if $C_i = (I_i \vee I_2)$, where I_1 and I_2 are literals, then include

$$(I_1 \vee I_2 \vee p) \wedge (I_1 \vee I_2 \vee \neg p)$$

as clauses of ϕ.

- Why?

$$(I_1 \vee I_2 \vee p) \wedge (I_1 \vee I_2 \vee \neg p) = (I_1 \vee I_2) \vee (p \wedge \neg p) =$$
$$(I_1 \vee I_2) \vee (F) = I_1 \vee I_2$$

Now, we need to include more literals as necessary

Given C_i as a disjunctive part of the previous formula.

- If C_i has 3 distinct literals, then simply include C_i as a clause of ϕ.

If C_i has 2 distinct literals

if $C_i = (I_i \vee I_2)$, where I_1 and I_2 are literals, then include

$$(I_1 \vee I_2 \vee p) \wedge (I_1 \vee I_2 \vee \neg p)$$

as clauses of ϕ.

- Why?

$$(I_1 \vee I_2 \vee p) \wedge (I_1 \vee I_2 \vee \neg p) = (I_1 \vee I_2) \vee (p \wedge \neg p) =$$
$$(I_1 \vee I_2) \vee (F) = I_1 \vee I_2$$

Now, we need to include more literals as necessary

Given C_i as a disjunctive part of the previous formula.

- If C_i has 3 distinct literals, then simply include C_i as a clause of ϕ.

If C_i has 2 distinct literals

if $C_i = (I_i \vee I_2)$, where I_1 and I_2 are literals, then include

$$(I_1 \vee I_2 \vee p) \wedge (I_1 \vee I_2 \vee \neg p)$$

as clauses of ϕ.

- Why?

$$(I_1 \vee I_2 \vee p) \wedge (I_1 \vee I_2 \vee \neg p) = (I_1 \vee I_2) \vee (p \wedge \neg p) = \\ (I_1 \vee I_2) \vee (F) = I_1 \vee I_2$$

Now, we need to include more literals as necessary

If C_i has just 1 distinct literal I

- Then include $(I \lor p \lor q) \land (I \lor p \lor \neg q) \land (I \lor \neg p \lor q) \land (I \lor \neg p \lor \neg q)$ as clauses of ϕ.
- Why?

$$
\begin{aligned}
(I \lor p \lor q) \land (I \lor p \lor \neg q) \land \dots \\
(I \lor \neg p \lor q) \land (I \lor \neg p \lor \neg q) &= I \lor [(p \lor q) \land (p \lor \neg q) \land \dots \\
&\quad (\neg p \lor q) \land (\neg p \lor \neg q)] \\
&= I \lor [p \lor (q \land \neg q) \land \dots \\
&\quad (\neg p \lor (q \land \neg q))] \\
&= I \lor [(p \lor F) \land (\neg p \lor F)] \\
&= I \lor [p \land \neg p] \\
&= I \lor F = I
\end{aligned}
$$

Now, we need to include more literals as necessary

If C_i has just 1 distinct literal I

- Then include $(I \vee p \vee q) \wedge (I \vee p \vee \neg q) \wedge (I \vee \neg p \vee q) \wedge (I \vee \neg p \vee \neg q)$ as clauses of ϕ.
- Why?

$$
\begin{aligned}
(I \vee p \vee q) \wedge (I \vee p \vee \neg q) \wedge \ldots \\
(I \vee \neg p \vee q) \wedge (I \vee \neg p \vee \neg q) &= I \vee [(p \vee q) \wedge (p \vee \neg q) \wedge \ldots \\
&\quad (\neg p \vee q) \wedge (\neg p \vee \neg q)] \\
&= I \vee [p \vee (q \wedge \neg q) \wedge \ldots \\
&\quad (\neg p \vee (q \wedge \neg q))] \\
&= I \vee [(p \vee F) \wedge (\neg p \vee F)] \\
&= I \vee [p \wedge \neg p] \\
&= I \vee F = I
\end{aligned}
$$

Now, we need to include more literals as necessary

If C_i has just 1 distinct literal I

- Then include $(I \lor p \lor q) \land (I \lor p \lor \neg q) \land (I \lor \neg p \lor q) \land (I \lor \neg p \lor \neg q)$ as clauses of ϕ.
- Why?

$$
\begin{aligned}
(I \lor p \lor q) \land (I \lor p \lor \neg q) \land \ldots \\
(I \lor \neg p \lor q) \land (I \lor \neg p \lor \neg q) &= I \lor [(p \lor q) \land (p \lor \neg q) \land \ldots \\
&\quad (\neg p \lor q) \land (\neg p \lor \neg q)] \\
&= I \lor [p \lor (q \land \neg q) \land \ldots \\
&\quad (\neg p \lor (q \land \neg q))] \\
&= I \lor [(p \lor F) \land (\neg p \lor F)] \\
&= I \lor [p \land \neg p] \\
&= I \lor F = I
\end{aligned}
$$

Now, we need to include more literals as necessary

If C_i has just 1 distinct literal I

- Then include $(I \vee p \vee q) \wedge (I \vee p \vee \neg q) \wedge (I \vee \neg p \vee q) \wedge (I \vee \neg p \vee \neg q)$ as clauses of ϕ.
- Why?

$$
\begin{aligned}
(I \vee p \vee q) \wedge (I \vee p \vee \neg q) \wedge \dots \\
(I \vee \neg p \vee q) \wedge (I \vee \neg p \vee \neg q) &= I \vee [(p \vee q) \wedge (p \vee \neg q) \wedge \dots \\
&\quad (\neg p \vee q) \wedge (\neg p \vee \neg q)] \\
&= I \vee [p \vee (q \wedge \neg q) \wedge \dots \\
&\quad (\neg p \vee (q \wedge \neg q))] \\
&= I \vee [(p \vee F) \wedge (\neg p \vee F)] \\
&= I \vee [p \wedge \neg p] \\
&= I \vee F = I
\end{aligned}
$$

Now, we need to include more literals as necessary

If C_i has just 1 distinct literal I

- Then include $(I \vee p \vee q) \wedge (I \vee p \vee \neg q) \wedge (I \vee \neg p \vee q) \wedge (I \vee \neg p \vee \neg q)$ as clauses of ϕ.
- Why?

$$
\begin{aligned}
(I \vee p \vee q) \wedge (I \vee p \vee \neg q) \wedge \ldots \\
(I \vee \neg p \vee q) \wedge (I \vee \neg p \vee \neg q) &= I \vee [(p \vee q) \wedge (p \vee \neg q) \wedge \ldots \\
&\quad (\neg p \vee q) \wedge (\neg p \vee \neg q)] \\
&= I \vee [p \vee (q \wedge \neg q) \wedge \ldots \\
&\quad (\neg p \vee (q \wedge \neg q))] \\
&= I \vee [(p \vee F) \wedge (\neg p \vee F)] \\
&= I \vee [p \wedge \neg p] \\
&= I \vee F = I
\end{aligned}
$$

Now, we need to include more literals as necessary

If C_i has just 1 distinct literal I

- Then include $(I \lor p \lor q) \land (I \lor p \lor \neg q) \land (I \lor \neg p \lor q) \land (I \lor \neg p \lor \neg q)$ as clauses of ϕ.
- Why? $(I \lor p \lor q) \land (I \lor p \lor \neg q) \land \ldots$

$$
\begin{aligned}
(I \lor \neg p \lor q) \land (I \lor \neg p \lor \neg q) &= I \lor [(p \lor q) \land (p \lor \neg q) \land \ldots \\
&\quad (\neg p \lor q) \land (\neg p \lor \neg q)] \\
&= I \lor [p \lor (q \land \neg q) \land \ldots \\
&\quad (\neg p \lor (q \land \neg q))] \\
&= I \lor [(p \lor F) \land (\neg p \lor F)] \\
&= I \lor [p \land \neg p] \\
&= I \lor F = I
\end{aligned}
$$

Finally, we need to prove the polynomial reduction

First

- Constructing ϕ' from ϕ introduces at most 1 variable and 1 clause per connective in ϕ.

Second

- Constructing ϕ'' from ϕ' can introduce at most 8 clauses into ϕ'' for each clause from ϕ', since each clause of ϕ'' has at most 3 variables, and the truth table for each clause has at most $2^3 = 8$ rows.

Third

- The construction of ϕ''' from ϕ' introduces at most 4 clauses into ϕ''' for each clause of ϕ''.

Finally, we need to prove the polynomial reduction

First

- Constructing ϕ' from ϕ introduces at most 1 variable and 1 clause per connective in ϕ.

Second

- Constructing ϕ'' from ϕ' can introduce at most 8 clauses into ϕ'' for each clause from ϕ', since each clause of ϕ'' has at most 3 variables, and the truth table for each clause has at most $2^3 = 8$ rows.

Third

- The construction of ϕ''' from ϕ' introduces at most 4 clauses into ϕ''' for each clause of ϕ''.

Finally, we need to prove the polynomial reduction

First

- Constructing ϕ' from ϕ introduces at most 1 variable and 1 clause per connective in ϕ.

Second

- Constructing ϕ'' from ϕ' can introduce at most 8 clauses into ϕ'' for each clause from ϕ', since each clause of ϕ'' has at most 3 variables, and the truth table for each clause has at most $2^3 = 8$ rows.

Third

- The construction of ϕ''' from ϕ' introduces at most 4 clauses into ϕ''' for each clause of ϕ''.

Finally

Thus

$$SAT \leq_p 3-CNF$$

Theorem 34.10

Satisfiability of boolean formulas in 3-conjunctive normal form is NP-complete.

Now the next problem

The Clique Problem.

Finally

Thus

$$SAT \leq_p 3 - CNF$$

Theorem 34.10

Satisfiability of boolean formulas in 3-conjunctive normal form is NP-complete.

Now the next problem

The Clique Problem.

Finally

Thus

$$SAT \leq_p 3 - CNF$$

Theorem 34.10

Satisfiability of boolean formulas in 3-conjunctive normal form is NP-complete.

Now the next problem

The Clique Problem.

Excercises

From Cormen's book solve

- 34.4-1
- 34.4-2
- 34.4-5
- 34.4-6
- 34.4-7

Outline

The Clique Problem

Definition

A clique in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E.

As a decision problem

$CLIQUE = \{< G, k > \mid G \text{is a graph with a clique of size } k\}$

The Clique Problem

Definition

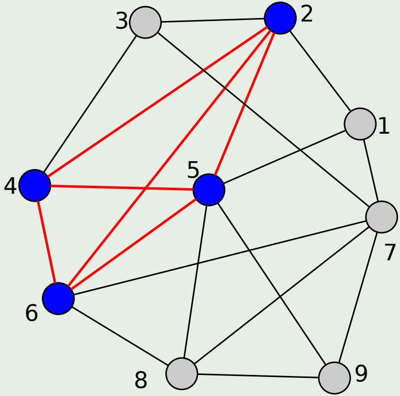
A clique in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E.

As a decision problem

$CLIQUE = \{< G, k > \mid G$is a graph with a clique of size $k\}$

Example

A Clique of size $k = 4$



The clique problem is NP-Complete

Theorem 34.11

The clique problem is NP-Complete.

Proof

1. To show that $CLIQUE \in NP$, for a given graph $G = (V, E)$ we use the set $V' \in V$ of vertices in the clique as certificate for G.

Thus

This is can be done in polynomial time because we only need to check all possibles pairs of $u, v \in V'$, which takes $|V'|(|V'| - 1)$.

The clique problem is NP-Complete

Theorem 34.11

The clique problem is NP-Complete.

Proof

1. To show that $CLIQUE \in NP$, for a given graph $G = (V, E)$ we use the set $V' \in V$ of vertices in the clique as certificate for G.

Thus

This is can be done in polynomial time because we only need to check all possibles pairs of $u, v \in V'$, which takes $|V'|(|V'|-1)$.

The clique problem is NP-Complete

Theorem 34.11

The clique problem is NP-Complete.

Proof

1. To show that $CLIQUE \in NP$, for a given graph $G = (V, E)$ we use the set $V' \in V$ of vertices in the clique as certificate for G.

Thus

This is can be done in polynomial time because we only need to check all possibles pairs of $u, v \in V'$, which takes $|V'| \left(|V'| - 1\right)$.

Proof

Now, we only need to prove that the problem is NP-Hard

Which is surprising, after all we are going from logic to graph problems!!!

Now

We start with an instance of 3-CNF-SAT

- $C_1 \wedge C_2 \wedge \ldots \wedge C_k$ a boolean 3-CNF formula with k clauses.

We know for each $1 \leq r \leq k$

$$C_r = l_1^r \vee l_2^r \vee l_3^r$$

Proof

Now, we only need to prove that the problem is NP-Hard

Which is surprising, after all we are going from logic to graph problems!!!

Now

We start with an instance of 3-CNF-SAT

- $C_1 \wedge C_2 \wedge ... \wedge C_k$ a boolean 3-CNF formula with k clauses.

We know for each $1 \leq r \leq k$

$$C_r = l_1^r \vee l_2^r \vee l_3^r$$

Proof

Now, we only need to prove that the problem is NP-Hard

Which is surprising, after all we are going from logic to graph problems!!!

Now

We start with an instance of 3-CNF-SAT

- $C_1 \wedge C_2 \wedge ... \wedge C_k$ a boolean 3-CNF formula with k clauses.

We know for each $1 \leq r \leq k$

$$C_r = l_1^r \vee l_2^r \vee l_3^r$$

Then

Now, we construct the following graph $G = (V, E)$

We place a triple of vertices v_1^r, v_2^r, v_3^r for each $C_r = l_1^r \vee l_2^r \vee l_3^r$.

Then

Now, we construct the following graph $G = (V, E)$

We place a triple of vertices v_1^r, v_2^r, v_3^r for each $C_r = l_1^r \lor l_2^r \lor l_3^r$.

We put an edge between two vertices v_i^r and v_j^s, if

- v_i^r and v_j^s are in different triples i.e. $r \neq s$.
- Their corresponding literals are consistent i.e. l_i^r is not the negation of l_j^s

Then

Now, we construct the following graph $G = (V, E)$

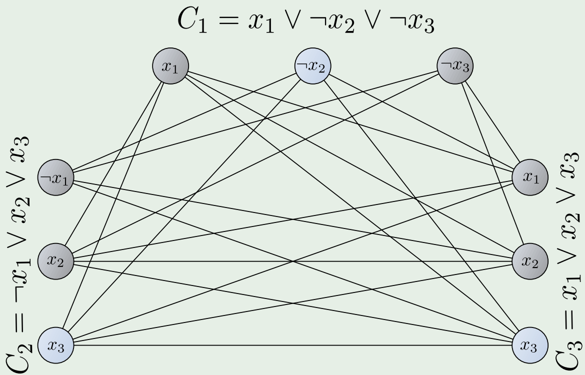
We place a triple of vertices v_1^r, v_2^r, v_3^r for each $C_r = l_1^r \vee l_2^r \vee l_3^r$.

We put an edge between two vertices v_i^r and v_j^s, if

- v_i^r and v_j^s are in different triples i.e. $r \neq s$.
- Their corresponding literals are consistent i.e. l_i^r is not the negation of l_j^s

Example

For $\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$



Now, we show that the transformation ϕ into G is a reduction

We start with the $\Longrightarrow$

- Suppose ϕ has a satisfying assignment.
- Thus each clause C_r contains at least one literal l_i^r mapping to 1, which corresponds to a vertex v_i^r.

Now, we show that the transformation ϕ into G is a reduction

We start with the $\Longrightarrow$

- Suppose ϕ has a satisfying assignment.
- Thus each clause C_r contains at least one literal l_i^r mapping to 1, which corresponds to a vertex v_i^r.

Now, we pick each of those literals

We finish with a set V' of such literals.

Now, we show that the transformation ϕ into G is a reduction

We start with the $\Longrightarrow$

- Suppose ϕ has a satisfying assignment.
- Thus each clause C_r contains at least one literal l_i^r mapping to 1, which corresponds to a vertex v_i^r.

Now, we pick each of those literals

We finish with a set V' of such literals.

V' is a clique, how?

Given two vertices v_i^r and $v_j^s \in V'$, with $r \neq s$, such that the corresponding literals l_i^r and l_j^s map to 1 by the satisfying assignment.

Now, we show that the transformation ϕ into G is a reduction

We start with the $\Longrightarrow$

- Suppose ϕ has a satisfying assignment.
- Thus each clause C_r contains at least one literal l_i^r mapping to 1, which corresponds to a vertex v_i^r.

Now, we pick each of those literals

We finish with a set V' of such literals.

V' is a clique, how?

Given two vertices v_i^r and $v_j^s \in V'$, with $r \neq s$, such that the corresponding literals l_i^r and l_j^s map to 1 by the satisfying assignment.

Then

This two literals

They cannot be complements.

Finally

By construction of G, the edge $\left(v_i^r, v_j^s\right)$ belongs to E.

Thus

We have a clique of size k in the graph $G = (V, E)$.

Then

This two literals

They cannot be complements.

Finally

By construction of G, the edge $\left(v_i^r, v_j^s\right)$ belongs to E.

Thus

We have a clique of size k in the graph $G = (V, E)$.

Then

This two literals

They cannot be complements.

Finally

By construction of G, the edge $\left(v_i^r, v_j^s\right)$ belongs to E.

Thus

We have a clique of size k in the graph $G = (V, E)$.

We now prove $\Longleftarrow$

Conversely

Suppose that G has a clique V' of size k.

Did you notice?

No edges in G connect vertices in the same triple, thus V' contains one vertex per triple.

Now

Now, we assign 1 to each literal l_i^r such that $v_i^r \in V'$

- Notice that we cannot assign 1 to both a literal and its complement by construction.

We now prove $\Longleftarrow$

Conversely

Suppose that G has a clique V' of size k.

Did you notice?

No edges in G connect vertices in the same triple, thus V' contains one vertex per triple.

Now

Now, we assign 1 to each literal l_i^r such that $v_i^r \in V'$

- Notice that we cannot assign 1 to both a literal and its complement by construction.

We now prove $\Longleftarrow$

Conversely

Suppose that G has a clique V' of size k.

Did you notice?

No edges in G connect vertices in the same triple, thus V' contains one vertex per triple.

Now

Now, we assign 1 to each literal l_i^r such that $v_i^r \in V'$

- Notice that we cannot assign 1 to both a literal and its complement by construction.

Finally

We have with that assignment

That each clause C_r is satisfied, thus ϕ is satisfied!!!

Note

Any variables that do not correspond to a vertex in the clique may be set arbitrarily.

Then

$$3 - CNF \leq_p CLIQUE$$

Finally

We have with that assignment

That each clause C_r is satisfied, thus ϕ is satisfied!!!

Note

Any variables that do not correspond to a vertex in the clique may be set arbitrarily.

Then

$$3-CNF \leq_p CLIQUE$$

Finally

We have with that assignment

That each clause C_r is satisfied, thus ϕ is satisfied!!!

Note

Any variables that do not correspond to a vertex in the clique may be set arbitrarily.

Then

$$3 - CNF \leq_p CLIQUE$$

Remarks

Something Notable

We have reduced an arbitrary instance of 3-CNF-SAT to an instance of CLIQUE with a particular structure.

Thus

It is possible to think that we have shown only that CLIQUE is NP-hard in graphs in which the vertices are restricted to occur in triples and in which there are no edges between vertices in the same triple.

Actually this is true

- But it is enough to prove that CLIQUE is NP-hard.
- Why? If we had a polynomial-time algorithm that solved CLIQUE in the general sense, we will solve in polynomial time the restricted version.

Remarks

Something Notable

We have reduced an arbitrary instance of 3-CNF-SAT to an instance of CLIQUE with a particular structure.

Thus

It is possible to think that we have shown only that CLIQUE is NP-hard in graphs in which the vertices are restricted to occur in triples and in which there are no edges between vertices in the same triple.

Actually this is true

- But it is enough to prove that CLIQUE is NP-hard.
- Why? If we had a polynomial-time algorithm that solved CLIQUE in the general sense, we will solve in polynomial time the restricted version.

Remarks

Something Notable

We have reduced an arbitrary instance of 3-CNF-SAT to an instance of CLIQUE with a particular structure.

Thus

It is possible to think that we have shown only that CLIQUE is NP-hard in graphs in which the vertices are restricted to occur in triples and in which there are no edges between vertices in the same triple.

Actually this is true

- But it is enough to prove that CLIQUE is NP-hard.
- Why? If we had a polynomial-time algorithm that solved CLIQUE in the general sense, we will solve in polynomial time the restricted version.

Remarks

In the opposite approach

Reducing instances of 3-CNF-SAT with a special structure to general instances of CLIQUE would not have sufficed.

Why not?

- Perhaps the instances of 3-CNF-SAT that we chose to reduce from were "easy," not reducing an NP-Hard problem to CLIQUE.
- Observe also that the reduction used the instance of 3-CNF-SAT, but not the solution.

Remarks

In the opposite approach

Reducing instances of 3-CNF-SAT with a special structure to general instances of CLIQUE would not have sufficed.

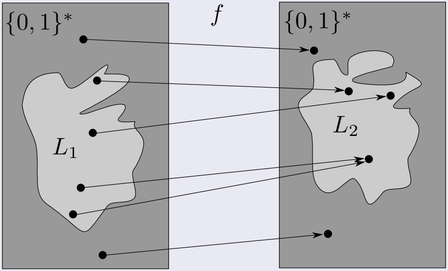
Why not?

- Perhaps the instances of 3-CNF-SAT that we chose to reduce from were "easy," not reducing an NP-Hard problem to CLIQUE.
- Observe also that the reduction used the instance of 3-CNF-SAT, but not the solution.

Thus

This would have been a serious error

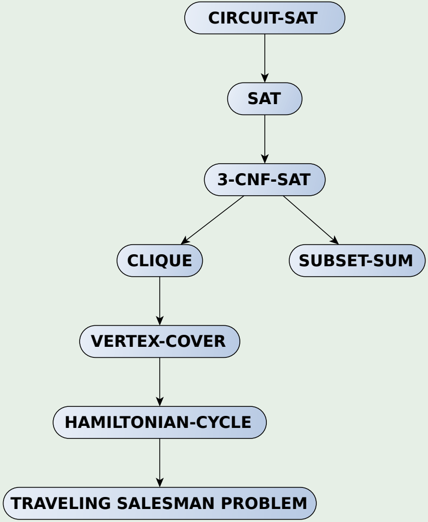
Remember the mapping:



Outline

Now, we have

Family of NP-Complete Problems



Excercises

From Cormen's book solve

- 34.5-1
- 34.5-2
- 34.5-3
- 34.5-4
- 34.5-5
- 34.5-7
- 34.5-8