Is P = NP?

Abhiram Ranade

April 4, 2016

Recap and Outline

The story so far..

Recap and Outline

The story so far..

- We do not have polytime algorithms for many problems such as IS, VC, CSAT, CNFSAT, Knapsack, Travelling salesman problem, Graph colouring, and many more.

Recap and Outline

The story so far..

- We do not have polytime algorithms for many problems such as IS, VC, CSAT, CNFSAT, Knapsack, Travelling salesman problem, Graph colouring, and many more.
- However, we can try reducing them to each other which establishes the relative difficulty of finding polytime algorithms for them.

Recap and Outline

The story so far..

- We do not have polytime algorithms for many problems such as IS, VC, CSAT, CNFSAT, Knapsack, Travelling salesman problem, Graph colouring, and many more.
- However, we can try reducing them to each other which establishes the relative difficulty of finding polytime algorithms for them.

Today:

Recap and Outline

The story so far..

- We do not have polytime algorithms for many problems such as IS, VC, CSAT, CNFSAT, Knapsack, Travelling salesman problem, Graph colouring, and many more.
- However, we can try reducing them to each other which establishes the relative difficulty of finding polytime algorithms for them.

Today:

- Turns out these problems are similar in an intuitive sense, and this similarity can be formalized.

Recap and Outline

The story so far..

- We do not have polytime algorithms for many problems such as IS, VC, CSAT, CNFSAT, Knapsack, Travelling salesman problem, Graph colouring, and many more.
- However, we can try reducing them to each other which establishes the relative difficulty of finding polytime algorithms for them.

Today:

- Turns out these problems are similar in an intuitive sense, and this similarity can be formalized.

They belong to a class "NP".

Recap and Outline

The story so far..

- We do not have polytime algorithms for many problems such as IS, VC, CSAT, CNFSAT, Knapsack, Travelling salesman problem, Graph colouring, and many more.
- However, we can try reducing them to each other which establishes the relative difficulty of finding polytime algorithms for them.

Today:

- Turns out these problems are similar in an intuitive sense, and this similarity can be formalized.

  They belong to a class "NP".
- Formalizing the similarity has formal benefit too.

Recap and Outline

The story so far..

- We do not have polytime algorithms for many problems such as IS, VC, CSAT, CNFSAT, Knapsack, Travelling salesman problem, Graph colouring, and many more.
- However, we can try reducing them to each other which establishes the relative difficulty of finding polytime algorithms for them.

Today:

- Turns out these problems are similar in an intuitive sense, and this similarity can be formalized.

  They belong to a class "NP".
- Formalizing the similarity has formal benefit too.

  Cook's theorem

Recap and Outline

The story so far..

- We do not have polytime algorithms for many problems such as IS, VC, CSAT, CNFSAT, Knapsack, Travelling salesman problem, Graph colouring, and many more.
- However, we can try reducing them to each other which establishes the relative difficulty of finding polytime algorithms for them.

Today:

- Turns out these problems are similar in an intuitive sense, and this similarity can be formalized.

  They belong to a class "NP".
- Formalizing the similarity has formal benefit too.

  Cook's theorem
- The classes NPC and NPH.

The class NP: class of puzzle like problems

The class NP: class of puzzle like problems

"A good puzzle is one whose answer is difficult to discover, but given the answer it is check that it is indeed correct."

The class NP: class of puzzle like problems

"A good puzzle is one whose answer is difficult to discover, but given the answer it is check that it is indeed correct."

Examples:

The class NP: class of puzzle like problems

"A good puzzle is one whose answer is difficult to discover, but given the answer it is check that it is indeed correct."

Examples:

- Sudoku

The class NP: class of puzzle like problems

"A good puzzle is one whose answer is difficult to discover, but given the answer it is check that it is indeed correct."

Examples:

- Sudoku
- Jigsaw puzzles

The class NP: class of puzzle like problems

"A good puzzle is one whose answer is difficult to discover, but given the answer it is check that it is indeed correct."

Examples:

- Sudoku
- Jigsaw puzzles

"Elegant math theorems" are also like that. Proofs are easy in hindsight.

The class NP: class of puzzle like problems

"A good puzzle is one whose answer is difficult to discover, but given the answer it is check that it is indeed correct."

Examples:

- Sudoku
- Jigsaw puzzles

"Elegant math theorems" are also like that. Proofs are easy in hindsight.

VC, IS, TSP, CSAT, SAT are also similar!

The class NP: class of puzzle like problems

"A good puzzle is one whose answer is difficult to discover, but given the answer it is check that it is indeed correct."

Examples:

- Sudoku
- Jigsaw puzzles

"Elegant math theorems" are also like that. Proofs are easy in hindsight.

VC, IS, TSP, CSAT, SAT are also similar!

Deciding whether a graph does have an IS of size k seems to be difficult.

The class NP: class of puzzle like problems

"A good puzzle is one whose answer is difficult to discover, but given the answer it is check that it is indeed correct."

Examples:

- Sudoku
- Jigsaw puzzles

"Elegant math theorems" are also like that. Proofs are easy in hindsight.

VC, IS, TSP, CSAT, SAT are also similar!

Deciding whether a graph does have an IS of size k seems to be difficult.

But if someone gives you a proof that a graph has a size k IS, can we check it quickly?

Example: Proof/Evidence for IS

Example: Proof/Evidence for IS

IS(G,k) : Does G have an independent set of size k?

Example: Proof/Evidence for IS

IS(G,k) : Does G have an independent set of size k?

Algorithm answers "YES": what kind of proof can it have?

Example: Proof/Evidence for IS

IS(G,k) : Does G have an independent set of size k?

Algorithm answers "YES": what kind of proof can it have?

If I want to prove that "There exist an animal having 3 heads and 2 tails", the most convincing proof would be to show such an animal.

Example: Proof/Evidence for IS

IS(G,k) : Does G have an independent set of size k?

Algorithm answers "YES": what kind of proof can it have?

If I want to prove that "There exist an animal having 3 heads and 2 tails", the most convincing proof would be to show such an animal.

The most direct proof for "Does G have a size k independent set?" would be to show a size k independent set.

Example: Proof/Evidence for IS

IS(G,k) : Does G have an independent set of size k?

Algorithm answers "YES": what kind of proof can it have?

If I want to prove that "There exist an animal having 3 heads and 2 tails", the most convincing proof would be to show such an animal.

The most direct proof for "Does G have a size k independent set?" would be to show a size k independent set.

- Good news 1: This proof is short: IS can be specified in only n bits.

Example: Proof/Evidence for IS

IS(G,k) : Does G have an independent set of size k?

Algorithm answers "YES": what kind of proof can it have?

If I want to prove that "There exist an animal having 3 heads and 2 tails", the most convincing proof would be to show such an animal.

The most direct proof for "Does G have a size k independent set?" would be to show a size k independent set.

- Good news 1: This proof is short: IS can be specified in only n bits.
- Good news 2: Given a candidate set of vertices, it is possible to check in polytime whether it is independent and has size k.

Example: Proof/Evidence for IS

IS(G,k) : Does G have an independent set of size k?

Algorithm answers "YES": what kind of proof can it have?

If I want to prove that "There exist an animal having 3 heads and 2 tails", the most convincing proof would be to show such an animal.

The most direct proof for "Does G have a size k independent set?" would be to show a size k independent set.

- Good news 1: This proof is short: IS can be specified in only n bits.
- Good news 2: Given a candidate set of vertices, it is possible to check in polytime whether it is independent and has size k.

Algorithm answers "No":

Example: Proof/Evidence for IS

IS(G,k) : Does G have an independent set of size k?

Algorithm answers "YES": what kind of proof can it have?

If I want to prove that "There exist an animal having 3 heads and 2 tails", the most convincing proof would be to show such an animal.

The most direct proof for "Does G have a size k independent set?" would be to show a size k independent set.

- Good news 1: This proof is short: IS can be specified in only n bits.
- Good news 2: Given a candidate set of vertices, it is possible to check in polytime whether it is independent and has size k.

Algorithm answers "No":
Not clear if there even exists a short proof for this..

Example: Proof/Evidence for IS

IS(G,k) : Does G have an independent set of size k?

Algorithm answers "YES": what kind of proof can it have?

If I want to prove that "There exist an animal having 3 heads and 2 tails", the most convincing proof would be to show such an animal.

The most direct proof for "Does G have a size k independent set?" would be to show a size k independent set.

- Good news 1: This proof is short: IS can be specified in only n bits.
- Good news 2: Given a candidate set of vertices, it is possible to check in polytime whether it is independent and has size k.

Algorithm answers "No":
Not clear if there even exists a short proof for this..
Perhaps we must go over all possible subsets?

Example: Proof/Evidence for IS

IS(G,k) : Does G have an independent set of size k?

Algorithm answers "YES": what kind of proof can it have?

If I want to prove that "There exist an animal having 3 heads and 2 tails", the most convincing proof would be to show such an animal.

The most direct proof for "Does G have a size k independent set?" would be to show a size k independent set.

- Good news 1: This proof is short: IS can be specified in only n bits.
- Good news 2: Given a candidate set of vertices, it is possible to check in polytime whether it is independent and has size k.

Algorithm answers "No":
Not clear if there even exists a short proof for this..
Perhaps we must go over all possible subsets?
Similar to Sudoku? How do you prove that there is no solution to a given Sudoku puzzle?

Example 2: Hamiltonian Cycle (HC)

Example 2: Hamiltonian Cycle (HC)

Hamiltonian Cycle(G): Does a given graph G contain a cycle that passes through every vertex exactly once?

Example 2: Hamiltonian Cycle (HC)

Hamiltonian Cycle(G): Does a given graph G contain a cycle that passes through every vertex exactly once?

YES answer: The cycle itself could be given as the proof of its existence.

Example 2: Hamiltonian Cycle (HC)

Hamiltonian Cycle(G): Does a given graph G contain a cycle that passes through every vertex exactly once?

YES answer: The cycle itself could be given as the proof of its existence.

Given a candidate sequence of vertices, we can check in polytime whether it is indeed a cycle, whether it passes through every vertex exactly once, and whether it is a subgraph.

Example 2: Hamiltonian Cycle (HC)

Hamiltonian Cycle(G): Does a given graph G contain a cycle that passes through every vertex exactly once?

YES answer: The cycle itself could be given as the proof of its existence.

Given a candidate sequence of vertices, we can check in polytime whether it is indeed a cycle, whether it passes through every vertex exactly once, and whether it is a subgraph.

NO answer: Not clear if a short proof even exists.

Example 2: Hamiltonian Cycle (HC)

Hamiltonian Cycle(G): Does a given graph G contain a cycle that passes through every vertex exactly once?

YES answer: The cycle itself could be given as the proof of its existence.

Given a candidate sequence of vertices, we can check in polytime whether it is indeed a cycle, whether it passes through every vertex exactly once, and whether it is a subgraph.

NO answer: Not clear if a short proof even exists.

The situation is similar for problems such as ILP, CSAT, VC, Knapsack..

Example 2: Hamiltonian Cycle (HC)

Hamiltonian Cycle(G): Does a given graph G contain a cycle that passes through every vertex exactly once?

YES answer: The cycle itself could be given as the proof of its existence.

Given a candidate sequence of vertices, we can check in polytime whether it is indeed a cycle, whether it passes through every vertex exactly once, and whether it is a subgraph.

NO answer: Not clear if a short proof even exists.

The situation is similar for problems such as ILP, CSAT, VC, Knapsack..

- YES answers can be backed by short, easily verifiable proofs.

Example 2: Hamiltonian Cycle (HC)

Hamiltonian Cycle(G): Does a given graph G contain a cycle that passes through every vertex exactly once?

YES answer: The cycle itself could be given as the proof of its existence.

Given a candidate sequence of vertices, we can check in polytime whether it is indeed a cycle, whether it passes through every vertex exactly once, and whether it is a subgraph.

NO answer: Not clear if a short proof even exists.

The situation is similar for problems such as ILP, CSAT, VC, Knapsack..

- YES answers can be backed by short, easily verifiable proofs.
- NO answers: not clear!

Deciding vs. proving

Deciding vs. proving

Current view of $IS(G, k)$

Deciding vs. proving

Current view of $IS(G, k)$

Instance = Question : Does G have an independent set of size k?

Deciding vs. proving

Current view of $IS(G, k)$
Instance = Question : Does G have an independent set of size k?
Goal: Decide whether the answer is YES or NO.

Deciding vs. proving

Current view of $IS(G, k)$

Instance = Question : Does G have an independent set of size k?

Goal: Decide whether the answer is YES or NO.

Alternate view:

Deciding vs. proving

Current view of $IS(G, k)$

Instance = Question : Does G have an independent set of size k?

Goal: Decide whether the answer is YES or NO.

Alternate view:

Instance = Statement : G has an independent set of size k.

Deciding vs. proving

Current view of $IS(G, k)$

Instance = Question : Does G have an independent set of size k?

Goal: Decide whether the answer is YES or NO.

Alternate view:

Instance = Statement : G has an independent set of size k.

Goal: Prove or disprove the statement.

Deciding vs. proving

Current view of $IS(G, k)$

Instance = Question : Does G have an independent set of size k?

Goal: Decide whether the answer is YES or NO.

Alternate view:

Instance = Statement : G has an independent set of size k.

Goal: Prove or disprove the statement.

Alternate view of Sudoku:

Deciding vs. proving

Current view of $IS(G, k)$

Instance = Question : Does G have an independent set of size k?

Goal: Decide whether the answer is YES or NO.

Alternate view:

Instance = Statement : G has an independent set of size k.

Goal: Prove or disprove the statement.

Alternate view of Sudoku:

Statement: "Numbers can be filled to satisfy requirements"

Deciding vs. proving

Current view of $IS(G, k)$

Instance = Question : Does G have an independent set of size k?

Goal: Decide whether the answer is YES or NO.

Alternate view:

Instance = Statement : G has an independent set of size k.

Goal: Prove or disprove the statement.

Alternate view of Sudoku:

Statement: "Numbers can be filled to satisfy requirements"

"Proof": Here are the required numbers.

Deciding vs. proving

Current view of $IS(G, k)$

Instance = Question : Does G have an independent set of size k?

Goal: Decide whether the answer is YES or NO.

Alternate view:

Instance = Statement : G has an independent set of size k.

Goal: Prove or disprove the statement.

Alternate view of Sudoku:

Statement: "Numbers can be filled to satisfy requirements"

"Proof": Here are the required numbers.

NP: Class of problems whose instances have short, easily verifiable proofs.

Deciding vs. proving

Current view of $IS(G, k)$

Instance = Question : Does G have an independent set of size k?

Goal: Decide whether the answer is YES or NO.

Alternate view:

Instance = Statement : G has an independent set of size k.

Goal: Prove or disprove the statement.

Alternate view of Sudoku:

Statement: "Numbers can be filled to satisfy requirements"

"Proof": Here are the required numbers.

NP: Class of problems whose instances have short, easily verifiable proofs.

Proof for instance (G, k) of IS: The independent set itself.

Deciding vs. proving

Current view of $IS(G, k)$
Instance = Question : Does G have an independent set of size k?
Goal: Decide whether the answer is YES or NO.
Alternate view:
Instance = Statement : G has an independent set of size k.
Goal: Prove or disprove the statement.

Alternate view of Sudoku:
Statement: "Numbers can be filled to satisfy requirements"
"Proof": Here are the required numbers.

NP: Class of problems whose instances have short, easily verifiable proofs.
Proof for instance (G, k) of IS: The independent set itself.

Formal definition: Class of problems having a "polytime verifier".

Polynomial time verifier:

A polytime verifier for a decision problem Q is a function Verify that takes as input strings x,y, where x is an instance of Q, and y has length polynomial in x.

Polynomial time verifier:

A polytime verifier for a decision problem Q is a function Verify that takes as input strings x,y, where x is an instance of Q, and y has length polynomial in x.

- x is true $\Rightarrow$ Verify(x,y) must returns true for some y.

Polynomial time verifier:

A polytime verifier for a decision problem Q is a function Verify that takes as input strings x,y, where x is an instance of Q, and y has length polynomial in x.

- x is true $\Rightarrow$ Verify(x,y) must returns true for some y.
- x is false $\Rightarrow$ Verify(x,y) must return false for all y.

Polynomial time verifier:

A polytime verifier for a decision problem Q is a function Verify that takes as input strings x,y, where x is an instance of Q, and y has length polynomial in x.

- x is true $\Rightarrow$ Verify(x,y) must returns true for some y.
- x is false $\Rightarrow$ Verify(x,y) must return false for all y.
- Verify must run in time polynomial in $|x|$.

Polynomial time verifier:

A polytime verifier for a decision problem Q is a function Verify that takes as input strings x,y, where x is an instance of Q, and y has length polynomial in x.

- x is true $\Rightarrow$ Verify(x,y) must returns true for some y.
- x is false $\Rightarrow$ Verify(x,y) must return false for all y.
- Verify must run in time polynomial in $|x|$.

Definition does not explicitly mention proving!

Polynomial time verifier:

A polytime verifier for a decision problem Q is a function Verify that takes as input strings x,y, where x is an instance of Q, and y has length polynomial in x.

- x is true $\Rightarrow$ Verify(x,y) must returns true for some y.
- x is false $\Rightarrow$ Verify(x,y) must return false for all y.
- Verify must run in time polynomial in $|x|$.

Definition does not explicitly mention proving!

But that is the only interpretation possible!

Polynomial time verifier:

A polytime verifier for a decision problem Q is a function Verify that takes as input strings x,y, where x is an instance of Q, and y has length polynomial in x.

- x is true $\Rightarrow$ Verify(x,y) must returns true for some y.
- x is false $\Rightarrow$ Verify(x,y) must return false for all y.
- Verify must run in time polynomial in $|x|$.

Definition does not explicitly mention proving!

But that is the only interpretation possible!

- y = potential proof that x is true.

Polynomial time verifier:

A polytime verifier for a decision problem Q is a function Verify that takes as input strings x,y, where x is an instance of Q, and y has length polynomial in x.

- x is true $\Rightarrow$ Verify(x,y) must returns true for some y.
- x is false $\Rightarrow$ Verify(x,y) must return false for all y.
- Verify must run in time polynomial in $|x|$.

Definition does not explicitly mention proving!

But that is the only interpretation possible!

- y = potential proof that x is true.
- x is true $\Rightarrow$ Verify(x,y) must return true for some y.

Polynomial time verifier:

A polytime verifier for a decision problem Q is a function Verify that takes as input strings x,y, where x is an instance of Q, and y has length polynomial in x.

- x is true $\Rightarrow$ Verify(x,y) must returns true for some y.
- x is false $\Rightarrow$ Verify(x,y) must return false for all y.
- Verify must run in time polynomial in $|x|$.

Definition does not explicitly mention proving!

But that is the only interpretation possible!

- y $=$ potential proof that x is true.
- x is true $\Rightarrow$ Verify(x,y) must return true for some y.

True statements must have a verifiable proof.

Polynomial time verifier:

A polytime verifier for a decision problem Q is a function Verify that takes as input strings x,y, where x is an instance of Q, and y has length polynomial in x.

- x is true $\Rightarrow$ Verify(x,y) must returns true for some y.
- x is false $\Rightarrow$ Verify(x,y) must return false for all y.
- Verify must run in time polynomial in $|x|$.

Definition does not explicitly mention proving!

But that is the only interpretation possible!

- y = potential proof that x is true.
- x is true $\Rightarrow$ Verify(x,y) must return true for some y.
  True statements must have a verifiable proof.
- x is false $\Rightarrow$ Verify must return false for all y.

Polynomial time verifier:

A polytime verifier for a decision problem Q is a function Verify that takes as input strings x,y, where x is an instance of Q, and y has length polynomial in x.

- x is true ⇒ Verify(x,y) must returns true for some y.
- x is false ⇒ Verify(x,y) must return false for all y.
- Verify must run in time polynomial in $|x|$.

Definition does not explicitly mention proving!

But that is the only interpretation possible!

- y = potential proof that x is true.
- x is true ⇒ Verify(x,y) must return true for some y.

  True statements must have a verifiable proof.
- x is false ⇒ Verify must return false for all y.

  No y should fool Verify into certifying that x is true.

Polynomial time verifier:

A polytime verifier for a decision problem Q is a function Verify that takes as input strings x,y, where x is an instance of Q, and y has length polynomial in x.

- x is true $\Rightarrow$ Verify(x,y) must returns true for some y.
- x is false $\Rightarrow$ Verify(x,y) must return false for all y.
- Verify must run in time polynomial in $|x|$.

Definition does not explicitly mention proving!

But that is the only interpretation possible!

- y = potential proof that x is true.
- x is true $\Rightarrow$ Verify(x,y) must return true for some y.
  True statements must have a verifiable proof.
- x is false $\Rightarrow$ Verify must return false for all y.
  No y should fool Verify into certifying that x is true.
- y has length poly($|x|$) : proof is short.

Polynomial time verifier:

A polytime verifier for a decision problem Q is a function Verify that takes as input strings x,y, where x is an instance of Q, and y has length polynomial in x.

- x is true ⇒ Verify(x,y) must returns true for some y.
- x is false ⇒ Verify(x,y) must return false for all y.
- Verify must run in time polynomial in $|x|$.

Definition does not explicitly mention proving!

But that is the only interpretation possible!

- y = potential proof that x is true.
- x is true ⇒ Verify(x,y) must return true for some y.

  True statements must have a verifiable proof.
- x is false ⇒ Verify must return false for all y.

  No y should fool Verify into certifying that x is true.
- y has length poly($|x|$) : proof is short.
- Verify runs in polytime: proofs can be checked quickly.

Theorem: IS $\in$ NP

Theorem: IS $\in$ NP

Proof: We present the verifier function Verify.

Theorem: IS $\in$ NP

Proof: We present the verifier function Verify.

- Argument x to Verify:

Theorem: IS $\in$ NP

Proof: We present the verifier function Verify.

- Argument x to Verify: Instance of IS $=$ G, k.

Theorem: IS $\in$ NP

Proof: We present the verifier function Verify.

- Argument x to Verify: Instance of IS = G, k.
- Argument y to Verify:

Theorem: IS $\in$ NP

Proof: We present the verifier function Verify.

- Argument x to Verify: Instance of IS $=$ G, k.
- Argument y to Verify: "Proof".

Theorem: IS $\in$ NP

Proof: We present the verifier function Verify.

- Argument x to Verify: Instance of IS = G, k.
- Argument y to Verify: "Proof". We will choose y = subset of vertices, represented using sequence of n bits.

Theorem: IS $\in$ NP

Proof: We present the verifier function Verify.

- Argument x to Verify: Instance of IS = G, k.
- Argument y to Verify: "Proof". We will choose y = subset of vertices, represented using sequence of n bits.

Verify(G,k,y){
Check if y is independent in G.
For each edge (u,v) return false if y[u]=y[v]=1.
check if y contains k vertices.
Return false if sum of y[i] is not k.
If all checks succeed return true.
}

Theorem: IS $\in$ NP

Proof: We present the verifier function Verify.

- Argument x to Verify: Instance of IS = G, k.
- Argument y to Verify: "Proof". We will choose y = subset of vertices, represented using sequence of n bits.

Verify(G,k,y){
Check if y is independent in G.
For each edge (u,v) return false if y[u]=y[v]=1.
check if y contains k vertices.
Return false if sum of y[i] is not k.
If all checks succeed return true.
}

- If G has size k independent set, setting y = that IS will cause Verify to return true.

Theorem: IS $\in$ NP

Proof: We present the verifier function Verify.

- Argument x to Verify: Instance of IS = G, k.
- Argument y to Verify: "Proof". We will choose y = subset of vertices, represented using sequence of n bits.

Verify(G,k,y){
Check if y is independent in G.
For each edge (u,v) return false if y[u]=y[v]=1.
check if y contains k vertices.
Return false if sum of y[i] is not k.
If all checks succeed return true.
}

- If G has size k independent set, setting y = that IS will cause Verify to return true.
- If G does not have a size k IS, Verify will return false for all y.

Theorem: IS $\in$ NP

Proof: We present the verifier function Verify.

- Argument x to Verify: Instance of IS = G, k.
- Argument y to Verify: "Proof". We will choose y = subset of vertices, represented using sequence of n bits.

Verify(G,k,y){
Check if y is independent in G.
For each edge (u,v) return false if y[u]=y[v]=1.
check if y contains k vertices.
Return false if sum of y[i] is not k.
If all checks succeed return true.
}

- If G has size k independent set, setting y = that IS will cause Verify to return true.
- If G does not have a size k IS, Verify will return false for all y.
- Verify runs in polytime.

Exercise

Show that CSAT has a polytime verifier.

Exercise

Show that CSAT has a polytime verifier.

Key part of the answer: What proof string will you use?

Exercise

Show that CSAT has a polytime verifier.

Key part of the answer: What proof string will you use?

Proof: The satisfying assignment itself.

Exercise

Show that CSAT has a polytime verifier.

Key part of the answer: What proof string will you use?

Proof: The satisfying assignment itself.

Verify will use the proof string to compute the values produced by the gates, in topological sort order, and return true iff circuit output is true.

Exercise

Show that CSAT has a polytime verifier.

Key part of the answer: What proof string will you use?

Proof: The satisfying assignment itself.

Verify will use the proof string to compute the values produced by the gates, in topological sort order, and return true iff circuit output is true.

Make sure that Verify satisfies required conditions..

The class NP ("Non deterministic polynomial time")

The class NP ("Non deterministic polynomial time")

NP: Class of decision problems having polytime verifiers.

The class NP ("Non deterministic polynomial time")

NP: Class of decision problems having polytime verifiers.

"Non deterministic polynomial time" will be explained later

The class NP ("Non deterministic polynomial time")

NP: Class of decision problems having polytime verifiers.

"Non deterministic polynomial time" will be explained later

$Q \in NP$:

The class NP ("Non deterministic polynomial time")

NP: Class of decision problems having polytime verifiers.

"Non deterministic polynomial time" will be explained later

$Q \in NP$:

x : instance of Q

The class NP ("Non deterministic polynomial time")

NP: Class of decision problems having polytime verifiers.

"Non deterministic polynomial time" will be explained later

$Q \in NP$:

x : instance of Q

If x is true there is a short proof.

The class NP ("Non deterministic polynomial time")

NP: Class of decision problems having polytime verifiers.

"Non deterministic polynomial time" will be explained later

$Q \in NP$:

x : instance of Q

If x is true there is a short proof.

A purported proof of x can be checked in polytime.

The class NP ("Non deterministic polynomial time")

NP: Class of decision problems having polytime verifiers.

"Non deterministic polynomial time" will be explained later

$Q \in NP$:

x : instance of Q

If x is true there is a short proof.

A purported proof of x can be checked in polytime.

P (decision version): Class of problems whose instances can be solved in polytime.

The class NP ("Non deterministic polynomial time")

NP: Class of decision problems having polytime verifiers.

"Non deterministic polynomial time" will be explained later

$Q \in NP$:

x : instance of Q

If x is true there is a short proof.

A purported proof of x can be checked in polytime.

P (decision version): Class of problems whose instances can be solved in polytime.

instances interpreted as statements can be proved/disproved in polytime.

The class NP ("Non deterministic polynomial time")

NP: Class of decision problems having polytime verifiers.

"Non deterministic polynomial time" will be explained later

$Q \in NP$:
x : instance of Q
If x is true there is a short proof.
A purported proof of x can be checked in polytime.

P (decision version): Class of problems whose instances can be solved in polytime.
instances interpreted as statements can be proved/disproved in polytime.
Execution transcript = proof or disproof. (assuming correct algorithm)

The class NP ("Non deterministic polynomial time")

NP: Class of decision problems having polytime verifiers.

"Non deterministic polynomial time" will be explained later

$Q \in NP$:

x : instance of Q

If x is true there is a short proof.

A purported proof of x can be checked in polytime.

P (decision version): Class of problems whose instances can be solved in polytime.

instances interpreted as statements can be proved/disproved in polytime.

Execution transcript = proof or disproof. (assuming correct algorithm)

$Q \in P$:

The class NP ("Non deterministic polynomial time")

NP: Class of decision problems having polytime verifiers.

"Non deterministic polynomial time" will be explained later

$Q \in NP$:
x : instance of Q
If x is true there is a short proof.
A purported proof of x can be checked in polytime.

P (decision version): Class of problems whose instances can be solved in polytime.
instances interpreted as statements can be proved/disproved in polytime.
Execution transcript = proof or disproof. (assuming correct algorithm)

$Q \in P$:
x : instance of Q

The class NP ("Non deterministic polynomial time")

NP: Class of decision problems having polytime verifiers.

"Non deterministic polynomial time" will be explained later

$Q \in NP$:
x : instance of Q
If x is true there is a short proof.
A purported proof of x can be checked in polytime.

P (decision version): Class of problems whose instances can be solved in polytime.
instances interpreted as statements can be proved/disproved in polytime.
Execution transcript = proof or disproof. (assuming correct algorithm)

$Q \in P$:
x : instance of Q
A proof/disproof for x can be found in polytime.

The big question: Is P = NP?

The big question: Is P = NP?

Psychological/philosophical interpretation: Is checking a proof as easy as discovering it?

The big question: Is P $=$ NP?

Psychological/philosophical interpretation: Is checking a proof as easy as discovering it?

It would seem that checking is strictly easier than discovering.

The big question: Is P = NP?

Psychological/philosophical interpretation: Is checking a proof as easy as discovering it?

It would seem that checking is strictly easier than discovering.

Computational interpretation: Is it possible to solve problems such as IS, VC, TSP, CSAT, Knapsack in polytime?

The big question: Is P $=$ NP?

Psychological/philosophical interpretation: Is checking a proof as easy as discovering it?

It would seem that checking is strictly easier than discovering.

Computational interpretation: Is it possible to solve problems such as IS, VC, TSP, CSAT, Knapsack in polytime?

It would seem that solving is strictly harder than checking a solution.

The big question: Is P = NP?

Psychological/philosophical interpretation: Is checking a proof as easy as discovering it?

It would seem that checking is strictly easier than discovering.

Computational interpretation: Is it possible to solve problems such as IS, VC, TSP, CSAT, Knapsack in polytime?

It would seem that solving is strictly harder than checking a solution.

Most people indeed believe that P $\neq$ NP.

The big question: Is $P = NP$?

Psychological/philosophical interpretation: Is checking a proof as easy as discovering it?

It would seem that checking is strictly easier than discovering.

Computational interpretation: Is it possible to solve problems such as IS, VC, TSP, CSAT, Knapsack in polytime?

It would seem that solving is strictly harder than checking a solution.

Most people indeed believe that $P \neq NP$.

However, we can establish containment.

Theorem: P $\subseteq$ NP

Theorem: P $\subseteq$ NP

Proof: Let $Q \in P$ be any decision problem. We give its verifier:

Theorem: P $\subseteq$ NP

Proof: Let $Q \in P$ be any decision problem. We give its verifier:

```
Verify(x,y){
Solve the instance x.
Return the result.
}
```

Prove/disprove instance x.

Theorem: P $\subseteq$ NP

Proof: Let $Q \in P$ be any decision problem. We give its verifier:

Verify(x,y){
Solve the instance x. Prove/disprove instance x.
Return the result.
}

If x is TRUE: Verify outputs TRUE for all y.

Theorem: P $\subseteq$ NP

Proof: Let $Q \in P$ be any decision problem. We give its verifier:

Verify(x,y){
Solve the instance x. Prove/disprove instance x.
Return the result.
}

If x is TRUE: Verify outputs TRUE for all y.
Required only for some y.

Theorem: P $\subseteq$ NP

Proof: Let $Q \in P$ be any decision problem. We give its verifier:

Verify(x,y){
Solve the instance x. — Prove/disprove instance x.
Return the result.
}

If x is TRUE: Verify outputs TRUE for all y.

Required only for some y.

If x is FALSE: Verify outputs FALSE for all y.

Theorem: P $\subseteq$ NP

Proof: Let $Q \in P$ be any decision problem. We give its verifier:

```
Verify(x,y){
Solve the instance x.                Prove/disprove instance x.
Return the result.
}
```

If x is TRUE: Verify outputs TRUE for all y.

Required only for some y.

If x is FALSE: Verify outputs FALSE for all y.

Solving x takes polytime because Q $\in$ P.

$\square$

Theorem: P $\subseteq$ NP

Proof: Let $Q \in P$ be any decision problem. We give its verifier:

Verify(x,y){
Solve the instance x. Prove/disprove instance x.
Return the result.
}

If x is TRUE: Verify outputs TRUE for all y.
Required only for some y.
If x is FALSE: Verify outputs FALSE for all y.
Solving x takes polytime because Q $\in$ P.

$\square$

We are ignoring y completely; allowed by the definition!

Theorem: P $\subseteq$ NP

Proof: Let $Q \in P$ be any decision problem. We give its verifier:

Verify(x,y){
Solve the instance x. Prove/disprove instance x.
Return the result.
}

If x is TRUE: Verify outputs TRUE for all y.
Required only for some y.
If x is FALSE: Verify outputs FALSE for all y.
Solving x takes polytime because Q $\in$ P.

$\square$

We are ignoring y completely; allowed by the definition!

So current belief: P $\subset$ NP.

Remarks

Remarks

- Instead of the term "proof/evidence", the terms "certificate" or "witness" are also used.

Remarks

- Instead of the term "proof/evidence", the terms "certificate" or "witness" are also used.
- You are perhaps asking yourself: "Sure, we have defined a new class, and it sounds reasonable, but is it really useful?"

Remarks

- Instead of the term "proof/evidence", the terms "certificate" or "witness" are also used.
- You are perhaps asking yourself: "Sure, we have defined a new class, and it sounds reasonable, but is it really useful?"
- We show next that all problems in NP have a Karp reduction to CSAT.

Remarks

- Instead of the term "proof/evidence", the terms "certificate" or "witness" are also used.
- You are perhaps asking yourself: "Sure, we have defined a new class, and it sounds reasonable, but is it really useful?"
- We show next that all problems in NP have a Karp reduction to CSAT.
- Thus showing that a problem has a polytime verifier is an easy way to reduce it to CSAT.

Remarks

- Instead of the term "proof/evidence", the terms "certificate" or "witness" are also used.
- You are perhaps asking yourself: "Sure, we have defined a new class, and it sounds reasonable, but is it really useful?"
- We show next that all problems in NP have a Karp reduction to CSAT.
- Thus showing that a problem has a polytime verifier is an easy way to reduce it to CSAT.
- So it is not just a cute definition or taxonomy.

Cook's theorem

If Q $\in$ NP, then Q $\leq_K$ CSAT.

Cook's theorem

If $Q \in NP$, then $Q \leq_K CSAT$.

Proof Sketch:

Cook's theorem

If Q $\in$ NP, then Q $\leq_K$ CSAT.

Proof Sketch:

Instance-map(Instance q of Q)$\{$

Cook's theorem

If Q $\in$ NP, then Q $\leq_K$ CSAT.

Proof Sketch:

Instance-map(Instance q of Q)$\{$

Let C = computer loaded with the verifier V(x,y) for Q.

Cook's theorem

If Q $\in$ NP, then Q $\leq_K$ CSAT.

Proof Sketch:

Instance-map(Instance q of Q){

Let C = computer loaded with the verifier V(x,y) for Q.

Further suppose that we set the first argument x of V to be q.

Cook's theorem

If Q $\in$ NP, then Q $\leq_K$ CSAT.

Proof Sketch:

Instance-map(Instance q of Q){
Let C = computer loaded with the verifier V(x,y) for Q.
Further suppose that we set the first argument x of V to be q.
C is a circuit with input = y, second input of V.

Cook's theorem

If Q $\in$ NP, then Q $\leq_K$ CSAT.

Proof Sketch:

Instance-map(Instance q of Q)$\{$

Let C $=$ computer loaded with the verifier V(x,y) for Q.

Further suppose that we set the first argument x of V to be q.

C is a circuit with input $=$ y, second input of V.

Return C.

Cook's theorem

If Q $\in$ NP, then Q $\leq_K$ CSAT.

Proof Sketch:

Instance-map(Instance q of Q){
Let C $=$ computer loaded with the verifier V(x,y) for Q.
Further suppose that we set the first argument x of V to be q.
C is a circuit with input $=$ y, second input of V.
Return C.

C is not a combinational circuit, will fix later.

Cook's theorem

If $Q \in NP$, then $Q \leq_K$ CSAT.

Proof Sketch:

Instance-map(Instance q of Q){

Let C = computer loaded with the verifier V(x,y) for Q.

Further suppose that we set the first argument x of V to be q.

C is a circuit with input = y, second input of V.

Return C.

C is not a combinational circuit, will fix later.

}

Cook's theorem

If Q $\in$ NP, then Q $\leq_K$ CSAT.

Proof Sketch:

Instance-map(Instance q of Q){
Let C = computer loaded with the verifier V(x,y) for Q.
Further suppose that we set the first argument x of V to be q.
C is a circuit with input = y, second input of V.
Return C.

C is not a combinational circuit, will fix later.

}

Instance q has answer YES $\Rightarrow$ A proof y exists for q.

Cook's theorem

If Q $\in$ NP, then Q $\leq_K$ CSAT.

Proof Sketch:

Instance-map(Instance q of Q){

Let C = computer loaded with the verifier V(x,y) for Q.

Further suppose that we set the first argument x of V to be q.

C is a circuit with input = y, second input of V.

Return C.

C is not a combinational circuit, will fix later.

}

Instance q has answer YES $\Rightarrow$ A proof y exists for q.

$\Rightarrow$ C will output 1 if y = proof

Cook's theorem

If Q $\in$ NP, then Q $\leq_K$ CSAT.

Proof Sketch:

Instance-map(Instance q of Q){

Let C = computer loaded with the verifier V(x,y) for Q.

Further suppose that we set the first argument x of V to be q.

C is a circuit with input = y, second input of V.

Return C.

C is not a combinational circuit, will fix later.

}

Instance q has answer YES $\Rightarrow$ A proof y exists for q.

$\Rightarrow$ C will output 1 if y = proof

C outputs 1 $\Rightarrow$ y can be assigned values to make output = 1.

Cook's theorem

If Q $\in$ NP, then Q $\leq_K$ CSAT.

Proof Sketch:

Instance-map(Instance q of Q){
Let C = computer loaded with the verifier V(x,y) for Q.
Further suppose that we set the first argument x of V to be q.
C is a circuit with input = y, second input of V.
Return C.

C is not a combinational circuit, will fix later.

}

Instance q has answer YES $\Rightarrow$ A proof y exists for q.
$\Rightarrow$ C will output 1 if y = proof

C outputs 1 $\Rightarrow$ y can be assigned values to make output = 1.
$\Rightarrow$ A proof y exists for q.

Cook's theorem

If Q $\in$ NP, then Q $\leq_K$ CSAT.

Proof Sketch:

Instance-map(Instance q of Q){

Let C = computer loaded with the verifier V(x,y) for Q.

Further suppose that we set the first argument x of V to be q.

C is a circuit with input = y, second input of V.

Return C.

C is not a combinational circuit, will fix later.

}

Instance q has answer YES $\Rightarrow$ A proof y exists for q.

$\Rightarrow$ C will output 1 if y = proof

C outputs 1 $\Rightarrow$ y can be assigned values to make output = 1.

$\Rightarrow$ A proof y exists for q.

$\Rightarrow$ q has answer YES.

Cook's theorem

If Q $\in$ NP, then Q $\leq_K$ CSAT.

Proof Sketch:

Instance-map(Instance q of Q){

Let C = computer loaded with the verifier V(x,y) for Q.

Further suppose that we set the first argument x of V to be q.

C is a circuit with input = y, second input of V.

Return C.

C is not a combinational circuit, will fix later.

}

Instance q has answer YES $\Rightarrow$ A proof y exists for q.

$\Rightarrow$ C will output 1 if y = proof

C outputs 1 $\Rightarrow$ y can be assigned values to make output = 1.

$\Rightarrow$ A proof y exists for q.

$\Rightarrow$ q has answer YES.

Does this run in polytime?

Changing C to a combinational circuit

Changing C to a combinational circuit

Sequential circuit $=$ memory $+$ combinational circuit (CC).

Changing C to a combinational circuit

Sequential circuit = memory + combinational circuit (CC).

On clock high: memory gated onto combinational circuit, on clock low output of combinational circuit latched into memory.

Changing C to a combinational circuit

Sequential circuit = memory + combinational circuit (CC).

On clock high: memory gated onto combinational circuit, on clock low output of combinational circuit latched into memory.

Note: all memory gated out, all gated in

Changing C to a combinational circuit

Sequential circuit = memory + combinational circuit (CC).

On clock high: memory gated onto combinational circuit, on clock low output of combinational circuit latched into memory.

Note: all memory gated out, all gated in

If sequential circuit finishes execution in T steps, then we can replace it by a pipeline of T copies of CC.

Changing C to a combinational circuit

Sequential circuit = memory + combinational circuit (CC).

On clock high: memory gated onto combinational circuit, on clock low output of combinational circuit latched into memory.

Note: all memory gated out, all gated in

If sequential circuit finishes execution in T steps, then we can replace it by a pipeline of T copies of CC.

Output of circuit : wire that was feeding to register storing answer.

Changing C to a combinational circuit

Sequential circuit = memory + combinational circuit (CC).

On clock high: memory gated onto combinational circuit, on clock low output of combinational circuit latched into memory.

Note: all memory gated out, all gated in

If sequential circuit finishes execution in T steps, then we can replace it by a pipeline of T copies of CC.

Output of circuit : wire that was feeding to register storing answer.

T = poly($|x|$) $\Rightarrow$ memory size = $O(T)$

Changing C to a combinational circuit

Sequential circuit = memory + combinational circuit (CC).

On clock high: memory gated onto combinational circuit, on clock low output of combinational circuit latched into memory.

Note: all memory gated out, all gated in

If sequential circuit finishes execution in T steps, then we can replace it by a pipeline of T copies of CC.

Output of circuit : wire that was feeding to register storing answer.

T = poly($|x|$) $\Rightarrow$ memory size = $O(T)$

T = poly($|x|$) $\Rightarrow$ New circuit has size poly($|x|$).

Changing C to a combinational circuit

Sequential circuit $=$ memory $+$ combinational circuit (CC).

On clock high: memory gated onto combinational circuit, on clock low output of combinational circuit latched into memory.

Note: all memory gated out, all gated in

If sequential circuit finishes execution in T steps, then we can replace it by a pipeline of T copies of CC.

Output of circuit : wire that was feeding to register storing answer.

T $=$ poly($|x|$) $\Rightarrow$ memory size $= O(T)$

T $=$ poly($|x|$) $\Rightarrow$ New circuit has size poly($|x|$).

The new circuit is entirely combinational.

Changing C to a combinational circuit

Sequential circuit = memory + combinational circuit (CC).

On clock high: memory gated onto combinational circuit, on clock low output of combinational circuit latched into memory.

Note: all memory gated out, all gated in

If sequential circuit finishes execution in T steps, then we can replace it by a pipeline of T copies of CC.

Output of circuit : wire that was feeding to register storing answer.

T $= \text{poly}(|x|) \Rightarrow$ memory size $= O(T)$

T $= \text{poly}(|x|) \Rightarrow$ New circuit has size $\text{poly}(|x|)$.

The new circuit is entirely combinational.

The new circuit can be constructed in time $\text{poly}(|x|)$.

NP-completeness

NP-completeness

The term "complete" is used in the sense of "containing the essence", or "being the hardest".

NP-completeness

The term "complete" is used in the sense of "containing the essence", or "being the hardest".

Definition: A problem Q is said to be NP-complete if Q is in NP, and every problem R $\in$ NP reduces to Q.

NP-completeness

The term "complete" is used in the sense of "containing the essence", or "being the hardest".

Definition: A problem Q is said to be NP-complete if Q is in NP, and every problem R $\in$ NP reduces to Q.

Theorem: CSAT is NP-complete.

NP-completeness

The term "complete" is used in the sense of "containing the essence", or "being the hardest".

Definition: A problem Q is said to be NP-complete if Q is in NP, and every problem R $\in$ NP reduces to Q.

Theorem: CSAT is NP-complete.
Proof: CSAT has a polytime verifier, and hence it is in NP.

NP-completeness

The term "complete" is used in the sense of "containing the essence", or "being the hardest".

Definition: A problem Q is said to be NP-complete if Q is in NP, and every problem R $\in$ NP reduces to Q.

Theorem: CSAT is NP-complete.
Proof: CSAT has a polytime verifier, and hence it is in NP.
By Cook's theorem, all R $\in$ NP reduce to it. $\square$

NP-completeness

The term "complete" is used in the sense of "containing the essence", or "being the hardest".

Definition: A problem Q is said to be NP-complete if Q is in NP, and every problem R $\in$ NP reduces to Q.

Theorem: CSAT is NP-complete.
Proof: CSAT has a polytime verifier, and hence it is in NP.
By Cook's theorem, all R $\in$ NP reduce to it. $\square$

Theorem: 3SAT, ILP, VC, IS are NP-complete.

NP-completeness

The term "complete" is used in the sense of "containing the essence", or "being the hardest".

Definition: A problem Q is said to be NP-complete if Q is in NP, and every problem R $\in$ NP reduces to Q.

Theorem: CSAT is NP-complete.
Proof: CSAT has a polytime verifier, and hence it is in NP.
By Cook's theorem, all R $\in$ NP reduce to it. $\square$

Theorem: 3SAT, ILP, VC, IS are NP-complete.

Proof Sketch: It is easy to show these problems are in NP.

NP-completeness

The term "complete" is used in the sense of "containing the essence", or "being the hardest".

Definition: A problem Q is said to be NP-complete if Q is in NP, and every problem R $\in$ NP reduces to Q.

Theorem: CSAT is NP-complete.
Proof: CSAT has a polytime verifier, and hence it is in NP.
By Cook's theorem, all R $\in$ NP reduce to it. $\square$

Theorem: 3SAT, ILP, VC, IS are NP-complete.

Proof Sketch: It is easy to show these problems are in NP.
We showed that CSAT reduces to them. But all R $\in$ NP reduce to CSAT. $\square$

NP-completeness

The term "complete" is used in the sense of "containing the essence", or "being the hardest".

Definition: A problem Q is said to be NP-complete if Q is in NP, and every problem R $\in$ NP reduces to Q.

Theorem: CSAT is NP-complete.
Proof: CSAT has a polytime verifier, and hence it is in NP.
By Cook's theorem, all R $\in$ NP reduce to it. $\square$

Theorem: 3SAT, ILP, VC, IS are NP-complete.

Proof Sketch: It is easy to show these problems are in NP.
We showed that CSAT reduces to them. But all R $\in$ NP reduce to CSAT. $\square$

Strategy for proving that a problem R is NPC:

NP-completeness

The term "complete" is used in the sense of "containing the essence", or "being the hardest".

Definition: A problem Q is said to be NP-complete if Q is in NP, and every problem R $\in$ NP reduces to Q.

Theorem: CSAT is NP-complete.
Proof: CSAT has a polytime verifier, and hence it is in NP.
By Cook's theorem, all R $\in$ NP reduce to it. $\square$

Theorem: 3SAT, ILP, VC, IS are NP-complete.

Proof Sketch: It is easy to show these problems are in NP.
We showed that CSAT reduces to them. But all R $\in$ NP reduce to CSAT. $\square$

Strategy for proving that a problem R is NPC:
Prove that (1) R $\in$ NP. (2) Some NPC problem reduces to R.

Remarks

Remarks

- If some Q $\leq_C$ R, where Q is NPC, then R is said to be NP-hard.

Remarks

- If some $Q \leq_C R$, where Q is NPC, then R is said to be NP-hard.
  NPC = NP $\cap$ NPH.

Remarks

- If some Q $\leq_C$ R, where Q is NPC, then R is said to be NP-hard.
  NPC $=$ NP $\cap$ NPH.
- Usually it is easy to show that a given problem is in NP. The key point usually is what proof you will use. Be sure to mention this very clearly.

Remarks

- If some Q $\leq_C$ R, where Q is NPC, then R is said to be NP-hard.
  NPC $=$ NP $\cap$ NPH.
- Usually it is easy to show that a given problem is in NP. The key point usually is what proof you will use. Be sure to mention this very clearly.
- Proving NP-hardness is often non trivial. Note the direction of reduction. Some known NP-complete problem must be reduced to the problem you are considering. Doing it in the reverse direction is useless!