Theoretical Computer Science

The Mathematics of Problem Solving

Tim Jackman

BU Summer Challenge

July 16th, 2025

Theory

- Today we'll be talking about *theoretical computer science*

Theory

- Today we'll be talking about *theoretical computer science*
- This area studies the mathematical foundation of *computation*

Theory

- Today we'll be talking about *theoretical computer science*
- This area studies the mathematical foundation of *computation*
- Given a problem, we can ask:

Theory

- Today we'll be talking about *theoretical computer science*
- This area studies the mathematical foundation of *computation*
- Given a problem, we can ask:
    - Can a computer solve it?

Theory

- Today we'll be talking about *theoretical computer science*
- This area studies the mathematical foundation of *computation*
- Given a problem, we can ask:
  - Can a computer solve it?
  - How fast can a computer solve it?

Theory

- Today we'll be talking about *theoretical computer science*
- This area studies the mathematical foundation of *computation*
- Given a problem, we can ask:
  - Can a computer solve it?
  - How fast can a computer solve it?
  - What's the *fastest* a computer could ever solve it?

Theory

- Today we'll be talking about *theoretical computer science*
- This area studies the mathematical foundation of *computation*
- Given a problem, we can ask:
  - Can a computer solve it?
  - How fast can a computer solve it?
  - What's the *fastest* a computer could ever solve it?
  - How much space does a computer need to solve it?

Theory

- Today we'll be talking about *theoretical computer science*
- This area studies the mathematical foundation of *computation*
- Given a problem, we can ask:
  - Can a computer solve it?
  - How fast can a computer solve it?
  - What's the *fastest* a computer could ever solve it?
  - How much space does a computer need to solve it?
- Today we'll be focused again on speed

What Is A Problem?

- There are multiple types of problems: yes-no, search problems (e.g. find any x such that...), optimization (i.e. find the minimum size solution), etc.

What Is A Problem?

- There are multiple types of problems: yes-no, search problems (e.g. find any x such that...), optimization (i.e. find the minimum size solution), etc.
- Today we will be looking at *decision problems*: problems where the answer is “yes” or ”no”

What Is A Problem?

- There are multiple types of problems: yes-no, search problems (e.g. find any x such that...), optimization (i.e. find the minimum size solution), etc.
- Today we will be looking at *decision problems*: problems where the answer is “yes” or ”no”
- Problems are *sets* of specific *instances*, for example “is x prime?” is a problem, “is 100010101 prime?” is an instance

What Is A Problem?

- There are multiple types of problems: yes-no, search problems (e.g. find any x such that...), optimization (i.e. find the minimum size solution), etc.
- Today we will be looking at *decision problems*: problems where the answer is "yes" or "no"
- Problems are *sets* of specific *instances*, for example "is x prime?" is a problem, "is 100010101 prime?" is an instance
- Not all decision problems are *solvable* in general: "will program A ever stop running on input x?"

What Is A Problem?

- There are multiple types of problems: yes-no, search problems (e.g. find any x such that...), optimization (i.e. find the minimum size solution), etc.
- Today we will be looking at *decision problems*: problems where the answer is "yes" or "no"
- Problems are *sets* of specific *instances*, for example "is x prime?" is a problem, "is 100010101 prime?" is an instance
- Not all decision problems are *solvable* in general: "will program A ever stop running on input x?"
- Today we'll be talking about solvable decision problems, and how fast they can be solved.

Some Problems Can Be Solved Fast

- We've seen a number of solvable decision problems:

Some Problems Can Be Solved Fast

- We've seen a number of solvable decision problems:
  - Is String s in String[] arg?

Some Problems Can Be Solved Fast

- We've seen a number of solvable decision problems:
  - Is String s in String[] arg?
  - Is String s in the sorted String[] arg?

Some Problems Can Be Solved Fast

- We've seen a number of solvable decision problems:
  - Is String s in String[] arg?
  - Is String s in the sorted String[] arg?
  - Is String s a palindrome?

Some Problems Can Be Solved Fast

- We've seen a number of solvable decision problems:
  - Is String s in String[] arg?
  - Is String s in the sorted String[] arg?
  - Is String s a palindrome?
  - Does this array have duplicates?

Some Problems Can Be Solved Fast

- We've seen a number of solvable decision problems:
  - Is String s in String[] arg?
  - Is String s in the sorted String[] arg?
  - Is String s a palindrome?
  - Does this array have duplicates?
- These are all *efficiently* solvable in *polynomial time* - $\mathcal{O}(n^k)$ where k is a *constant*

Some Problems Can Be Solved Fast

- We've seen a number of solvable decision problems:
  - Is String s in String[] arg?
  - Is String s in the sorted String[] arg?
  - Is String s a palindrome?
  - Does this array have duplicates?
- These are all *efficiently* solvable in *polynomial time* - $\mathcal{O}(n^k)$ where k is a *constant*
- The set of all decision problems solvable in polynomial time is called $\mathcal{P}$

Some Problems Can Be Verified Fast

- Recall the problem of checking whether a Boolean formula can ever evaluate to `true`

Some Problems Can Be Verified Fast

- Recall the problem of checking whether a Boolean formula can ever evaluate to `true`
  - This is called the *Boolean Satisfiability Problem*

Some Problems Can Be Verified Fast

- Recall the problem of checking whether a Boolean formula can ever evaluate to `true`
  - This is called the *Boolean Satisfiability Problem*
  - Seems to require *brute force* over all 2^n combinations of assignments

Some Problems Can Be Verified Fast

- Recall the problem of checking whether a Boolean formula can ever evaluate to `true`
  - This is called the *Boolean Satisfiability Problem*
  - Seems to require *brute force* over all 2^n combinations of assignments
- Given a *purported* solution, it's easy to *verify*: plug in and evaluate - $O(n)$

Some Problems Can Be Verified Fast

- Recall the problem of checking whether a Boolean formula can ever evaluate to `true`
  - This is called the *Boolean Satisfiability Problem*
  - Seems to require *brute force* over all 2^n combinations of assignments
- Given a *purported* solution, it's easy to *verify*: plug in and evaluate - $O(n)$
- Verification Problems: Given a specific "solution," answer "yes/no" to the decision problem

Some Problems Can Be Verified Fast

- Recall the problem of checking whether a Boolean formula can ever evaluate to `true`
  - This is called the *Boolean Satisfiability Problem*
  - Seems to require *brute force* over all 2^n combinations of assignments
- Given a *purported* solution, it's easy to *verify*: plug in and evaluate - $O(n)$
- Verification Problems: Given a specific "solution," answer "yes/no" to the decision problem
  - If the "solution" is "bad" we can say "no", even if the answer is actually "yes"

Some Problems Can Be Verified Fast

- Recall the problem of checking whether a Boolean formula can ever evaluate to `true`
  - This is called the *Boolean Satisfiability Problem*
  - Seems to require *brute force* over all 2^n combinations of assignments
- Given a *purported* solution, it's easy to *verify*: plug in and evaluate - $O(n)$
- Verification Problems: Given a specific "solution," answer "yes/no" to the decision problem
  - If the "solution" is "bad" we can say "no", even if the answer is actually "yes"
  - Only requirement is that if the solution is "good", we say "yes"

Some Problems Can Be Verified Fast

- Recall the problem of checking whether a Boolean formula can ever evaluate to `true`
  - This is called the *Boolean Satisfiability Problem*
  - Seems to require *brute force* over all 2^n combinations of assignments
- Given a *purported* solution, it's easy to *verify*: plug in and evaluate - $O(n)$
- Verification Problems: Given a specific "solution," answer "yes/no" to the decision problem
  - If the "solution" is "bad" we can say "no", even if the answer is actually "yes"
  - Only requirement is that if the solution is "good", we say "yes"
- The set of *efficiently verifiable* (polynomial time) decision problems is called $\mathcal{NP}$

Some Problems Can Be Verified Fast

- Recall the problem of checking whether a Boolean formula can ever evaluate to `true`
  - This is called the *Boolean Satisfiability Problem*
  - Seems to require *brute force* over all 2^n combinations of assignments
- Given a *purported* solution, it's easy to *verify*: plug in and evaluate - $O(n)$
- Verification Problems: Given a specific "solution," answer "yes/no" to the decision problem
  - If the "solution" is "bad" we can say "no", even if the answer is actually "yes"
  - Only requirement is that if the solution is "good", we say "yes"
- The set of *efficiently verifiable* (polynomial time) decision problems is called $\mathcal{NP}$
- $\mathcal{NP}$ stands for *Nondeterministic Polynomial Time*, not *Not Polynomial*

The Biggest Question in Computer Science

- Every problem in $\mathcal{P}$ is also in $\mathcal{NP}$

The Biggest Question in Computer Science

- Every problem in $\mathcal{P}$ is also in $\mathcal{NP}$
- Is the reverse also true? I.e. Is $\mathcal{P} = \mathcal{NP}$?

The Biggest Question in Computer Science

- Every problem in $\mathcal{P}$ is also in $\mathcal{NP}$
- Is the reverse also true? I.e. Is $\mathcal{P} = \mathcal{NP}$?
    - This is called the “P vs NP” problem

The Biggest Question in Computer Science

- Every problem in $\mathcal{P}$ is also in $\mathcal{NP}$
- Is the reverse also true? I.e. Is $\mathcal{P} = \mathcal{NP}$?
    - This is called the "P vs NP" problem
    - There is a $1,000,000 prize for solving this problem

The Biggest Question in Computer Science

- Every problem in $\mathcal{P}$ is also in $\mathcal{NP}$
- Is the reverse also true? I.e. Is $\mathcal{P} = \mathcal{NP}$?
  - This is called the "P vs NP" problem
  - There is a $1,000,000 prize for solving this problem
  - Your name would go down in history

The Biggest Question in Computer Science

- Every problem in $\mathcal{P}$ is also in $\mathcal{NP}$
- Is the reverse also true? I.e. Is $\mathcal{P} = \mathcal{NP}$?
  - This is called the "P vs NP" problem
  - There is a $1,000,000 prize for solving this problem
  - Your name would go down in history

The Biggest Question in Computer Science

- Every problem in $\mathcal{P}$ is also in $\mathcal{NP}$
- Is the reverse also true? I.e. Is $\mathcal{P} = \mathcal{NP}$?
  - This is called the "P vs NP" problem
  - There is a $1,000,000 prize for solving this problem
  - Your name would go down in history
- Far reaching consequences if they *are* the same:

The Biggest Question in Computer Science

- Every problem in $\mathcal{P}$ is also in $\mathcal{NP}$
- Is the reverse also true? I.e. Is $\mathcal{P} = \mathcal{NP}$?
  - This is called the “P vs NP” problem
  - There is a $1,000,000 prize for solving this problem
  - Your name would go down in history
- Far reaching consequences if they *are* the same:
  - Is x the password to your bank account?

The Biggest Question in Computer Science

- Every problem in $\mathcal{P}$ is also in $\mathcal{NP}$
- Is the reverse also true? I.e. Is $\mathcal{P} = \mathcal{NP}$?
  - This is called the "P vs NP" problem
  - There is a $1,000,000 prize for solving this problem
  - Your name would go down in history
- Far reaching consequences if they *are* the same:
  - Is x the password to your bank account?
  - A lot of *cryptography* depend on verification being easy but solving being hard

Solving Problems to Solve Other Problems

- We've seen how problems become "easy" once we solve another problem:

Solving Problems to Solve Other Problems

- We've seen how problems become "easy" once we solve another problem:
  - Finding duplicates in an array is simple once we sort the array

Solving Problems to Solve Other Problems

- We've seen how problems become "easy" once we solve another problem:
  - Finding duplicates in an array is simple once we sort the array
- If solving problem A allows us to solve problem B, we say *B reduces to A*, written $B \leq A$

Solving Problems to Solve Other Problems

- We've seen how problems become "easy" once we solve another problem:
  - Finding duplicates in an array is simple once we sort the array
- If solving problem A allows us to solve problem B, we say *B reduces to A*, written $B \leq A$
- These *reductions* use a solution to A a subroutine to solve B

Solving Problems to Solve Other Problems

- We've seen how problems become "easy" once we solve another problem:
  - Finding duplicates in an array is simple once we sort the array
- If solving problem A allows us to solve problem B, we say *B reduces to A*, written $B \leq A$
- These *reductions* use a solution to A a subroutine to solve B
- Sometimes we can "transform" an instance of problem B into an instance of problem A

Solving Problems to Solve Other Problems

- We've seen how problems become "easy" once we solve another problem:
  - Finding duplicates in an array is simple once we sort the array
- If solving problem A allows us to solve problem B, we say *B reduces to A*, written $B \leq A$
- These *reductions* use a solution to A a subroutine to solve B
- Sometimes we can "transform" an instance of problem B into an instance of problem A
  - Once we do this "transformation," we can just call the solution to A and copy it's answer

Solving Problems to Solve Other Problems

- We've seen how problems become "easy" once we solve another problem:
  - Finding duplicates in an array is simple once we sort the array
- If solving problem A allows us to solve problem B, we say *B reduces to A*, written $B \leq A$
- These *reductions* use a solution to A a subroutine to solve B
- Sometimes we can "transform" an instance of problem B into an instance of problem A
  - Once we do this "transformation," we can just call the solution to A and copy it's answer
- If the reduction and the solution to A are both *efficient* then we can solve B efficiently

Solving Problems to Solve Other Problems

- We've seen how problems become "easy" once we solve another problem:
  - Finding duplicates in an array is simple once we sort the array
- If solving problem A allows us to solve problem B, we say *B reduces to A*, written $B \leq A$
- These *reductions* use a solution to A a subroutine to solve B
- Sometimes we can "transform" an instance of problem B into an instance of problem A
  - Once we do this "transformation," we can just call the solution to A and copy it's answer
- If the reduction and the solution to A are both *efficient* then we can solve B efficiently
- An efficient reduction says that problem A is "harder" than B

The Hardest Verification Problems

- It turns out that every $\mathcal{NP}$ problem *efficiently reduces* to Boolean Satisfiability

The Hardest Verification Problems

- It turns out that every $\mathcal{NP}$ problem *efficiently reduces* to Boolean Satisfiability
- If we can solve Boolean Satisfiability efficiently, then $\mathcal{NP} = \mathcal{P}$

The Hardest Verification Problems

- It turns out that every $\mathcal{NP}$ problem *efficiently reduces* to Boolean Satisfiability
- If we can solve Boolean Satisfiability efficiently, then $\mathcal{NP} = \mathcal{P}$
- This property is called "NP-completeness"

The Hardest Verification Problems

- It turns out that every $\mathcal{NP}$ problem *efficiently reduces* to Boolean Satisfiability
- If we can solve Boolean Satisfiability efficiently, then $\mathcal{NP} = \mathcal{P}$
- This property is called "NP-completeness"
- If we show Boolean Satisfiability *reduces to* another problem, that means that problem is (likely) "hard"

The Hardest Verification Problems

- It turns out that every $\mathcal{NP}$ problem *efficiently reduces* to Boolean Satisfiability
- If we can solve Boolean Satisfiability efficiently, then $\mathcal{NP} = \mathcal{P}$
- This property is called "NP-completeness"
- If we show Boolean Satisfiability *reduces to* another problem, that means that problem is (likely) "hard"
    - It certainly can't be solved in polynomial time with the current state of computer science

The Hardest Verification Problems

- It turns out that every $\mathcal{NP}$ problem *efficiently reduces* to Boolean Satisfiability
- If we can solve Boolean Satisfiability efficiently, then $\mathcal{NP} = \mathcal{P}$
- This property is called "NP-completeness"
- If we show Boolean Satisfiability *reduces to* another problem, that means that problem is (likely) "hard"
    - It certainly can't be solved in polynomial time with the current state of computer science
- Let's try showing a real problem is "hard"

A Real Example - Super Mario Maker

- Super Mario Maker 2 is a Nintendo Switch Game where you make and share Mario levels

A Real Example - Super Mario Maker

- Super Mario Maker 2 is a Nintendo Switch Game where you make and share Mario levels
- Players can play random mario levels that have been uploaded

A Real Example - Super Mario Maker

- Super Mario Maker 2 is a Nintendo Switch Game where you make and share Mario levels
- Players can play random mario levels that have been uploaded
- Nintendo doesn't like "impossible" levels, so you cannot upload until you demonstrate it is beatable

A Real Example - Super Mario Maker

- Super Mario Maker 2 is a Nintendo Switch Game where you make and share Mario levels
- Players can play random mario levels that have been uploaded
- Nintendo doesn't like "impossible" levels, so you cannot upload until you demonstrate it is beatable
- Some people are really good at designing hard levels, but cannot beat them themselves

A Real Example - Super Mario Maker

- Super Mario Maker 2 is a Nintendo Switch Game where you make and share Mario levels
- Players can play random mario levels that have been uploaded
- Nintendo doesn't like “impossible” levels, so you cannot upload until you demonstrate it is beatable
- Some people are really good at designing hard levels, but cannot beat them themselves
- If they made a Super Mario Maker 3, could Nintendo write an efficient program to check whether a level is beatable?

Reducing to Mario Maker

- We will "transform" a general Boolean Satisfiability instance into a general Mario Maker level

Reducing to Mario Maker

- We will "transform" a general Boolean Satisfiability instance into a general Mario Maker level
    - We will actually reduce 3-SAT: Boolean Satisfiability for formulas of the form

$$(x_1 \vee x_7 \vee \neg x_{10}) \wedge (x_2 \vee \neg x_8 \vee \neg x_{10}) \wedge (\ldots) \ldots$$

Reducing to Mario Maker

- We will "transform" a general Boolean Satisfiability instance into a general Mario Maker level
  - We will actually reduce 3-SAT: Boolean Satisfiability for formulas of the form

$$(x_1 \vee x_7 \vee \neg x_{10}) \wedge (x_2 \vee \neg x_8 \vee \neg x_{10}) \wedge (\ldots) \ldots$$

  - Each $\vee$ of three variables (or their negations) is called a *clause*

Reducing to Mario Maker

- We will “transform” a general Boolean Satisfiability instance into a general Mario Maker level
  - We will actually reduce 3-SAT: Boolean Satisfiability for formulas of the form

$$(x_1 \vee x_7 \vee \neg x_{10}) \wedge (x_2 \vee \neg x_8 \vee \neg x_{10}) \wedge (\ldots) \ldots$$

  - Each $\vee$ of three variables (or their negations) is called a *clause*
  - 3-SAT is also NP-complete

Reducing to Mario Maker

- We will "transform" a general Boolean Satisfiability instance into a general Mario Maker level
  - We will actually reduce 3-SAT: Boolean Satisfiability for formulas of the form

$$(x_1 \vee x_7 \vee \neg x_{10}) \wedge (x_2 \vee \neg x_8 \vee \neg x_{10}) \wedge (\ldots) \ldots$$

  - Each $\vee$ of three variables (or their negations) is called a *clause*
  - 3-SAT is also NP-complete
- We will take each *clause* and each variable and represent it in Mario

Reducing to Mario Maker

- We will "transform" a general Boolean Satisfiability instance into a general Mario Maker level
  - We will actually reduce 3-SAT: Boolean Satisfiability for formulas of the form

$$(x_1 \vee x_7 \vee \neg x_{10}) \wedge (x_2 \vee \neg x_8 \vee \neg x_{10}) \wedge (\ldots) \ldots$$

  - Each $\vee$ of three variables (or their negations) is called a *clause*
  - 3-SAT is also NP-complete
- We will take each *clause* and each variable and represent it in Mario
- Beating the Mario level will be equivalent to finding an assignment of the variables that

Reducing to Mario Maker

- We will break the problem down like always, and try to solve each individual part:

Reducing to Mario Maker

- We will break the problem down like always, and try to solve each individual part:
- How can we represent a variable? What does it mean for a variable to be true/false?

Reducing to Mario Maker

- We will break the problem down like always, and try to solve each individual part:
- How can we represent a variable? What does it mean for a variable to be true/false?
    - Power ups?

Reducing to Mario Maker

- We will break the problem down like always, and try to solve each individual part:
- How can we represent a variable? What does it mean for a variable to be true/false?
  - Power ups?
  - Not very extendable, there's a limited number of power ups in the game

Reducing to Mario Maker

- We will break the problem down like always, and try to solve each individual part:
- How can we represent a variable? What does it mean for a variable to be true/false?
  - Power ups?
  - Not very extendable, there's a limited number of power ups in the game
  - Shells for each variable, and they can either move or sit

Reducing to Mario Maker

- We will break the problem down like always, and try to solve each individual part:
- How can we represent a variable? What does it mean for a variable to be true/false?
  - Power ups?
  - Not very extendable, there's a limited number of power ups in the game
  - Shells for each variable, and they can either move or sit
- How do we represent a clause? How to represent $\vee$?

Reducing to Mario Maker

- We will break the problem down like always, and try to solve each individual part:
- How can we represent a variable? What does it mean for a variable to be true/false?
  - Power ups?
  - Not very extendable, there's a limited number of power ups in the game
  - Shells for each variable, and they can either move or sit
- How do we represent a clause? How to represent $\vee$?
  - Each clause will be a room, and you can only get to the other side if the clause evaluates to true

Reducing to Mario Maker

- We will break the problem down like always, and try to solve each individual part:
- How can we represent a variable? What does it mean for a variable to be true/false?
  - Power ups?
  - Not very extendable, there's a limited number of power ups in the game
  - Shells for each variable, and they can either move or sit
- How do we represent a clause? How to represent $\vee$?
  - Each clause will be a room, and you can only get to the other side if the clause evaluates to true
  - Mario can only jump so high, maybe a vine is revealed if the variable is correctly set

Reducing to Mario Maker

- We will break the problem down like always, and try to solve each individual part:
- How can we represent a variable? What does it mean for a variable to be true/false?
  - Power ups?
  - Not very extendable, there's a limited number of power ups in the game
  - Shells for each variable, and they can either move or sit
- How do we represent a clause? How to represent $\vee$?
  - Each clause will be a room, and you can only get to the other side if the clause evaluates to true
  - Mario can only jump so high, maybe a vine is revealed if the variable is correctly set
- Representing $\wedge$ is easy: we just chain rooms together

Reducing to Mario Maker

- This reduction is "straightforward" to compute

Reducing to Mario Maker

- This reduction is “straightforward” to compute
- If Nintendo made an efficient course checker, then they could solve Boolean Satisfiability

Reducing to Mario Maker

- This reduction is “straightforward” to compute
- If Nintendo made an efficient course checker, then they could solve Boolean Satisfiability
- So it looks like level creators will just need to “git gud” as gamers say

Reducing to Mario Maker

- This reduction is "straightforward" to compute
- If Nintendo made an efficient course checker, then they could solve Boolean Satisfiability
- So it looks like level creators will just need to "git gud" as gamers say

Can you satisfy:

$$(a \vee b \vee d) \wedge (a \vee b \vee \neg c) \wedge (a \vee d \vee \neg b) \wedge (b \vee c \vee \neg d) \wedge$$
$$(b \vee \neg a \vee \neg c) \wedge (c \vee d \vee \neg a) \wedge (c \vee \neg b \vee \neg d) \wedge (\neg a \vee \neg b \vee \neg c)$$

Reducing to Mario Maker

- This reduction is "straightforward" to compute
- If Nintendo made an efficient course checker, then they could solve Boolean Satisfiability
- So it looks like level creators will just need to "git gud" as gamers say

Can you satisfy:

$$(a \lor b \lor d) \land (a \lor b \lor \neg c) \land (a \lor d \lor \neg b) \land (b \lor c \lor \neg d) \land$$
$$(b \lor \neg a \lor \neg c) \land (c \lor d \lor \neg a) \land (c \lor \neg b \lor \neg d) \land (\neg a \lor \neg b \lor \neg c)$$

Let's load up Mario Maker and find out!

Temporary page!

LaTeX was unable to guess the total number of pages correctly. A
was some unprocessed data that should have been added to the
this extra page has been added to receive it.
If you rerun the document (without altering it) this surplus page
away, because LaTeX now knows how many pages to expect for t
document.