THE P $\neq$ NP PROBLEM

Manindra Agarwal

IIT Kanpur

CNR Rao Lecture, 2008

Outline

The Traveling Salesman Problem

- Suppose that you are given the road map of India.
- You need to find a traversal that covers all the cities/towns/villages of population $\geq 1,000$.
- And the traversal should have a short distance, say, $\leq 9,000$ kms.
- You will have to generate a very large number of traversals to find out a short traversal.
- Suppose that you are also given a claimed short traversal.
- It is now easy to verify that given claimed traversal is indeed a short traversal.

The Traveling Salesman Problem

- Suppose that you are given the road map of India.
- You need to find a traversal that covers all the cities/towns/villages of population $\geq 1,000$.
- And the traversal should have a short distance, say, $\leq 9,000$ kms.
- You will have to generate a very large number of traversals to find out a short traversal.
- Suppose that you are also given a claimed short traversal.
- It is now easy to verify that given claimed traversal is indeed a short traversal.

The Traveling Salesman Problem

- Suppose that you are given the road map of India.
- You need to find a traversal that covers all the cities/towns/villages of population $\geq 1,000$.
- And the traversal should have a short distance, say, $\leq 9,000$ kms.
- You will have to generate a very large number of traversals to find out a short traversal.
- Suppose that you are also given a claimed short traversal.
- It is now easy to verify that given claimed traversal is indeed a short traversal.

The Bin Packing Problem

- Suppose you have a large container of volume 1000 cubic meter and 150 boxes of varying sizes with volumes between 10 to 25 cubic meters.
- You need to fit at least half of these boxes in the container.
- You will need to try out various combinations of 75 boxes (there are $\binom{150}{75} > 10^{40}$ combinations) and various ways of laying them in the container to find a fitting.
- Suppose that you are also given a set of 75 boxes and a way of laying them.
- It is now easy to verify if these 75 boxes layed out in the given way will fit in the container.

The Bin Packing Problem

- Suppose you have a large container of volume 1000 cubic meter and 150 boxes of varying sizes with volumes between 10 to 25 cubic meters.
- You need to fit at least half of these boxes in the container.
- You will need to try out various combinations of 75 boxes (there are $\binom{150}{75} > 10^{40}$ combinations) and various ways of laying them in the container to find a fitting.
- Suppose that you are also given a set of 75 boxes and a way of laying them.
- It is now easy to verify if these 75 boxes layed out in the given way will fit in the container.

The Bin Packing Problem

- Suppose you have a large container of volume 1000 cubic meter and 150 boxes of varying sizes with volumes between 10 to 25 cubic meters.
- You need to fit at least half of these boxes in the container.
- You will need to try out various combinations of 75 boxes (there are $\binom{150}{75} > 10^{40}$ combinations) and various ways of laying them in the container to find a fitting.
- Suppose that you are also given a set of 75 boxes and a way of laying them.
- It is now easy to verify if these 75 boxes layed out in the given way will fit in the container.

Hall-I Room Allocation

- Each wing of Hall-I has 72 rooms.
- Suppose from a batch of 540 students, 72 need to be housed in C-wing.
- There are several students that are "incompatible" with each other, and so no such pair should be present in the wing.
- If there are a large number of incompatibilities, you will need to try out many combinations to get a correct one.
- Suppose you are also given the names of 72 students to be housed.
- It is now easy to verify if they are all compatible.

Hall-I Room Allocation

- Each wing of Hall-I has 72 rooms.
- Suppose from a batch of 540 students, 72 need to be housed in C-wing.
- There are several students that are "incompatible" with each other, and so no such pair should be present in the wing.
- If there are a large number of incompatibilities, you will need to try out many combinations to get a correct one.
- Suppose you are also given the names of 72 students to be housed.
- It is now easy to verify if they are all compatible.

Hall-I Room Allocation

- Each wing of Hall-I has 72 rooms.
- Suppose from a batch of 540 students, 72 need to be housed in C-wing.
- There are several students that are "incompatible" with each other, and so no such pair should be present in the wing.
- If there are a large number of incompatibilities, you will need to try out many combinations to get a correct one.
- Suppose you are also given the names of 72 students to be housed.
- It is now easy to verify if they are all compatible.

Discovery versus Verification

- In all these problems, finding a solution appears to be far more difficult than checking the correctness of a given solution.
- Informally, this makes sense as discovering a solution is often much more difficult than verifying its correctness.
- Can we formally prove this?
- Leads to the P versus NP problem.

Discovery versus Verification

- In all these problems, finding a solution appears to be far more difficult than checking the correctness of a given solution.
- Informally, this makes sense as discovering a solution is often much more difficult than verifying its correctness.
- Can we formally prove this?
- Leads to the P versus NP problem.

Discovery versus Verification

- In all these problems, finding a solution appears to be far more difficult than checking the correctness of a given solution.
- Informally, this makes sense as discovering a solution is often much more difficult than verifying its correctness.
- Can we formally prove this?
- Leads to the P versus NP problem.

Outline

Formalizing Easy-to-solve

- A problem is easy to solve if the solution can be computed quickly.
- Gives rise to two questions:
  - How is it computed?
  - How do we define "quickly"?

Formalizing Easy-to-solve

- A problem is easy to solve if the solution can be computed quickly.
- Gives rise to two questions:
  - How is it computed?
  - How do we define "quickly"?

Computating Method

- We will use an algorithm to compute.
- In practice, the algorithm will run on a computer via a computer program.

Computating Method

- We will use an algorithm to compute.
- In practice, the algorithm will run on a computer via a computer program.

Algorithms

- An algorithm is a set of precise instructions for computation.
- The algorithm can perform usual computational steps, e.g., assignments, arithmetic and boolean operations, loops.
- For us, an algorithm will always have input presented as a sequence of bits.
- The input size is the number of bits in the input to the algorithm.
- The algorithm stops after outputting the solution.

Algorithms

- An algorithm is a set of precise instructions for computation.
- The algorithm can perform usual computational steps, e.g., assignments, arithmetic and boolean operations, loops.
- For us, an algorithm will always have input presented as a sequence of bits.
- The input size is the number of bits in the input to the algorithm.
- The algorithm stops after outputting the solution.

Algorithms

- An algorithm is a set of precise instructions for computation.
- The algorithm can perform usual computational steps, e.g., assignments, arithmetic and boolean operations, loops.
- For us, an algorithm will always have input presented as a sequence of bits.
- The input size is the number of bits in the input to the algorithm.
- The algorithm stops after outputing the solution.

Algorithms

- An algorithm is a set of precise instructions for computation.
- The algorithm can perform usual computational steps, e.g., assignments, arithmetic and boolean operations, loops.
- For us, an algorithm will always have input presented as a sequence of bits.
- The input size is the number of bits in the input to the algorithm.
- The algorithm stops after outputting the solution.

Algorithms

- An algorithm is a set of precise instructions for computation.
- The algorithm can perform usual computational steps, e.g., assignments, arithmetic and boolean operations, loops.
- For us, an algorithm will always have input presented as a sequence of bits.
- The input size is the number of bits in the input to the algorithm.
- The algorithm stops after outputing the solution.

Time Measurement

- Let A be an algorithm and x be an input to it.
- Let $T_A(x)$ denote the number of steps of the algorithm on input x.
- Let $T_A(n)$ denote the maximum of $T_A(x)$ over all inputs x of size n.
- We will use $T_A(n)$ to quantify the time taken by algorithm A to solve a problem on different input sizes.
- For example, an algorithm A that adds two n bit numbers using school method has $T_A(n) = O(n)$.
- An algorithm B that multiplies two n bits numbers using school method has $T_A(n) = O(n^2)$.

Time Measurement

- Let A be an algorithm and x be an input to it.
- Let $T_A(x)$ denote the number of steps of the algorithm on input x.
- Let $T_A(n)$ denote the maximum of $T_A(x)$ over all inputs x of size n.
- We will use $T_A(n)$ to quantify the time taken by algorithm A to solve a problem on different input sizes.
- For example, an algorithm A that adds two n bit numbers using school method has $T_A(n) = O(n)$.
- An algorithm B that multiplies two n bits numbers using school method has $T_A(n) = O(n^2)$.

Time Measurement

- Let A be an algorithm and x be an input to it.
- Let $T_A(x)$ denote the number of steps of the algorithm on input x.
- Let $T_A(n)$ denote the maximum of $T_A(x)$ over all inputs x of size n.
- We will use $T_A(n)$ to quantify the time taken by algorithm A to solve a problem on different input sizes.
- For example, an algorithm A that adds two n bit numbers using school method has $T_A(n) = O(n)$.
- An algorithm B that multiplies two n bits numbers using school method has $T_A(n) = O(n^2)$.

Time Complexity of Problems

- A problem has time complexity $T_A(n)$ if there is an algorithm A that solves the problem on every input.
- Addition has time complexity $O(n)$.
- Multiplication has time complexity $O(n^2)$.

Time Complexity of Problems

- A problem has time complexity $T_A(n)$ if there is an algorithm A that solves the problem on every input.
- Addition has time complexity $O(n)$.
- Multiplication has time complexity $O(n^2)$.

Time Complexity of Problems

- A problem has time complexity $T_A(n)$ if there is an algorithm A that solves the problem on every input.
- Addition has time complexity $O(n)$.
- Multiplication has time complexity $O(n^2)$.

Quantifying Easy-to-compute

- The problems of adding and multiplying are definitely easy.
- Also, if a problem is easy, and another problem can be solved in time $n \cdot T(n)$ where $T(n)$ is the time complexity of the easy problem, then the new problem is also easy.
- This leads to the following definition: A problem is efficiently solvable if its time complexity is $n^{O(1)}$.
- Such problems are also called polynomial-time problems.

Quantifying Easy-to-compute

- The problems of adding and multiplying are definitely easy.
- Also, if a problem is easy, and another problem can be solved in time $n \cdot T(n)$ where $T(n)$ is the time complexity of the easy problem, then the new problem is also easy.
- This leads to the following definition: A problem is efficiently solvable if its time complexity is $n^{O(1)}$.
- Such problems are also called polynomial-time problems.

Quantifying Easy-to-compute

- The problems of adding and multiplying are definitely easy.
- Also, if a problem is easy, and another problem can be solved in time $n \cdot T(n)$ where $T(n)$ is the time complexity of the easy problem, then the new problem is also easy.
- This leads to the following definition: A problem is efficiently solvable if its time complexity is $n^{O(1)}$.
- Such problems are also called polynomial-time problems.

The Class P

The class P contains all efficiently solvable problems.

Caveat

A problem with time complexity n^{1000} is not efficiently solvable, but such problems do not arise in practice.

The Class P

The class P contains all efficiently solvable problems.

Caveat

A problem with time complexity n^{1000} is not efficiently solvable, but such problems do not arise in practice.

The Class NP

- This class contains all problems whose solutions can be efficiently verified.
- We need two properties:
  - The solution to an input should be of size similar to the input; so for an input of size n, the solution size is bounded by $n^{O(1)}$.
  - The problem of verifying the correctness of a given solution to a given input is in P.

The class NP contains all problems satisfying the above two properties.

The Class NP

- This class contains all problems whose solutions can be efficiently verified.
- We need two properties:
  - The solution to an input should be of size similar to the input; so for an input of size n, the solution size is bounded by $n^{O(1)}$,
  - The problem of verifying the correctness of a given solution to a given input is in P.

The class NP contains all problems satisfying the above two properties.

The Class NP

- This class contains all problems whose solutions can be efficiently verified.
- We need two properties:
  - The solution to an input should be of size similar to the input; so for an input of size n, the solution size is bounded by $n^{O(1)}$,
  - The problem of verifying the correctness of a given solution to a given input is in P.

The class NP contains all problems satisfying the above two properties.

Is $P \neq NP$?

- $P = NP$ means that for all problems whose solutions can be efficiently verified, the solutions can be efficiently generated too.
- It is widely believed that $P \neq NP$.
- This problem is listed as one of the seven most important unsolved problems in mathematics.
- There is a \$1 million prize for anyone who proves $P = NP$ or $P \neq NP$!

Is $P \neq NP$?

- $P = NP$ means that for all problems whose solutions can be efficiently verified, the solutions can be efficiently generated too.
- It is widely believed that $P \neq NP$.
- This problem is listed as one of the seven most important unsolved problems in mathematics.
- There is a \$1 million prize for anyone who proves $P = NP$ or $P \neq NP$!

Is $P \neq NP$?

- $P = NP$ means that for all problems whose solutions can be efficiently verified, the solutions can be efficiently generated too.
- It is widely believed that $P \neq NP$.
- This problem is listed as one of the seven most important unsolved problems in mathematics.
- There is a $1 million prize for anyone who proves $P = NP$ or $P \neq NP$!

Is $P \neq NP$?

- $\mathsf{P} = \mathsf{NP}$ means that for all problems whose solutions can be efficiently verified, the solutions can be efficiently generated too.
- It is widely believed that $\mathsf{P} \neq \mathsf{NP}$.
- This problem is listed as one of the seven most important unsolved problems in mathematics.
- There is a \$1 million prize for anyone who proves $\mathsf{P} = \mathsf{NP}$ or $\mathsf{P} \neq \mathsf{NP}$!

Outline

Diagonalization

- Diagonalization is a classical method first used by Cantor (1878) to prove that the infinity of reals is bigger than the infinity of integers.
- Since then, it has been used extensively in Computability Theory for seperating classes.
- The earliest attempts to seperate P from NP were through diagonalization.

Diagonalization

- Diagonalization is a classical method first used by Cantor (1878) to prove that the infinity of reals is bigger than the infinity of integers.
- Since then, it has been used extensively in Computability Theory for seperating classes.
- The earliest attempts to seperate P from NP were through diagonalization.

A Simple Diagonalization

- Each algorithm can be written down as a sequence of bits, and hence can be viewed as a number.
- Let A_1, A_2, $\ldots$ be the infinite sequence of algorithms such that
  - Algorithm A_i is represented by number i,
  - Algorithm A_i stops within $n^{\log \log i} + \log i$ steps on inputs of size n.
- All the algorithms in this enumeration are polynomial-time.
- For every problem in P, there is an algorithm in the above enumeration that solves it.

A Simple Diagonalization

- Each algorithm can be written down as a sequence of bits, and hence can be viewed as a number.
- Let A_1, A_2, $\ldots$ be the infinite sequence of algorithms such that
  - Algorithm A_i is represented by number i,
  - Algorithm A_i stops within $n^{\log\log i} + \log i$ steps on inputs of size n.
- All the algorithms in this enumeration are polynomial-time.
- For every problem in P, there is an algorithm in the above enumeration that solves it.

A Simple Diagonalization

- Each algorithm can be written down as a sequence of bits, and hence can be viewed as a number.
- Let A_1, A_2, ... be the infinite sequence of algorithms such that
  - Algorithm A_i is represented by number i,
  - Algorithm A_i stops within $n^{\log\log i} + \log i$ steps on inputs of size n.
- All the algorithms in this enumeration are polynomial-time.
- For every problem in P, there is an algorithm in the above enumeration that solves it.

A Simple Diagonalization

- Each algorithm can be written down as a sequence of bits, and hence can be viewed as a number.
- Let A_1, A_2, ... be the infinite sequence of algorithms such that
  - Algorithm A_i is represented by number i,
  - Algorithm A_i stops within $n^{\log\log i} + \log i$ steps on inputs of size n.
- All the algorithms in this enumeration are polynomial-time.
- For every problem in P, there is an algorithm in the above enumeration that solves it.

A Simple Diagonalization

- Define a new problem as: given i as input, output 1 if A_i outputs 0 on input i, else output 0.
- How much time does this problem take to solve?
  - An algorithm to solve the problem, given input i, needs to run the algorithm A_i on i for at most $(\log i)^{\log\log i} + \log i$ steps.
  - Let n be the length of input i; hence $n = \log i$.
  - So the algorithm takes time $O(n^{\log n})$ on inputs of size n.

A Simple Diagonalization

- Define a new problem as: given i as input, output 1 if A_i outputs 0 on input i, else output 0.
- How much time does this problem take to solve?
  - An algorithm to solve the problem, given input i, needs to run the algorithm A_i on i for at most $(\log i)^{\log \log i} + \log i$ steps.
  - Let n be the length of input i; hence $n = \log i$.
  - So the algorithm takes time $O(n^{\log n})$ on inputs of size n.

A Simple Diagonalization

- Define a new problem as: given i as input, output 1 if A_i outputs 0 on input i, else output 0.
- How much time does this problem take to solve?
  - An algorithm to solve the problem, given input i, needs to run the algorithm A_i on i for at most $(\log i)^{\log\log i} + \log i$ steps.
  - Let n be the length of input i; hence $n = \log i$.
  - So the algorithm takes time $O(n^{\log n})$ on inputs of size n.

A Simple Diagonalization

- Suppose algorithm A_j from the above sequence also solves this problem.
- What does A_j output on input j?
  - A_j outputs 1 if A_j on j outputs 0.
  - A_j outputs 0 if A_j on j does not output 0.
- Hence such an A_j cannot exist!
- Therefore, the problem is not in P.

A Simple Diagonalization

- Suppose algorithm A_j from the above sequence also solves this problem.
- What does A_j output on input j?
  - A_j outputs 1 if A_j on j outputs 0.
  - A_j outputs 0 if A_j on j does not output 0.
- Hence such an A_j cannot exist!
- Therefore, the problem is not in P.

A Simple Diagonalization

- Suppose algorithm A_j from the above sequence also solves this problem.
- What does A_j output on input j?
  - A_j outputs 1 if A_j on j outputs 0.
  - A_j outputs 0 if A_j on j does not output 0.
- Hence such an A_j cannot exist!
- Therefore, the problem is not in P.

A Simple Diagonalization

- Suppose algorithm A_j from the above sequence also solves this problem.
- What does A_j output on input j?
  - A_j outputs 1 if A_j on j outputs 0.
  - A_j outputs 0 if A_j on j does not output 0.
- Hence such an A_j cannot exist!
- Therefore, the problem is not in P.

A Simple Diagonalization

- Suppose algorithm A_j from the above sequence also solves this problem.
- What does A_j output on input j?
  - A_j outputs 1 if A_j on j outputs 0.
  - A_j outputs 0 if A_j on j does not output 0.
- Hence such an A_j cannot exist!
- Therefore, the problem is not in P.

Seperating P from NP Using Diagonalization

- It is not clear if the problem defined above is in the class NP.
- Can one define a problem in NP that diagonalizes over all polynomial-time algorithms as above?
- Unlikely!

Seperating P from NP Using Diagonalization

- It is not clear if the problem defined above is in the class NP.
- Can one define a problem in NP that diagonalizes over all polynomial-time algorithms as above?
- Unlikely!

Seperating P from NP Using Diagonalization

- It is not clear if the problem defined above is in the class NP.
- Can one define a problem in NP that diagonalizes over all polynomial-time algorithms as above?
- Unlikely!

The Relativization Barrier

- Suppose we are given algorithm A for free.
- This means that we can use A as subroutine in any algorithm and execution of A does not count towards the time taken.
- We can now define the classes P and NP relative to A.
- These classes are represented as P^A and NP^A.
- Such computations can be thought of as happening in another world where A can be efficiently executed!
- We can ask the same question as before: is $\mathrm{P}^A \neq \mathrm{NP}^A$?

The Relativization Barrier

- Suppose we are given algorithm A for free.
- This means that we can use A as subroutine in any algorithm and execution of A does not count towards the time taken.
- We can now define the classes P and NP relative to A.
- These classes are represented as P^A and NP^A.
- Such computations can be thought of as happening in another world where A can be efficiently executed!
- We can ask the same question as before: is $\mathrm{P}^A \neq \mathrm{NP}^A$?

The Relativization Barrier

- Suppose we are given algorithm A for free.
- This means that we can use A as subroutine in any algorithm and execution of A does not count towards the time taken.
- We can now define the classes P and NP relative to A.
- These classes are represented as P^A and NP^A.
- Such computations can be thought of as happening in another world where A can be efficiently executed!
- We can ask the same question as before: is $\mathrm{P}^A \neq \mathrm{NP}^A$?

The Relativization Barrier

- Suppose we are given algorithm A for free.
- This means that we can use A as subroutine in any algorithm and execution of A does not count towards the time taken.
- We can now define the classes P and NP relative to A.
- These classes are represented as P^A and NP^A.
- Such computations can be thought of as happening in another world where A can be efficiently executed!
- We can ask the same question as before: is $\mathrm{P}^A \neq \mathrm{NP}^A$?

The Relativization Barrier

- Baker, Gill and Solovay (1975) proved that there exists an algorithm A such that $\mathrm{P}^A = \mathrm{NP}^A$ and there exists an algorithm B such that $\mathrm{P}^B \neq \mathrm{NP}^B$.
- So any proof that works under all relativizations cannot show $\mathrm{P} = \mathrm{NP}$ or $\mathrm{P} \neq \mathrm{NP}$.
- All the standard diagonalization arguments work under all relativizations.
- Hence, they are useless for proving $\mathrm{P} \neq \mathrm{NP}$!

The Relativization Barrier

- Baker, Gill and Solovay (1975) proved that there exists an algorithm A such that $\mathsf{P}^A = \mathsf{NP}^A$ and there exists an algorithm B such that $\mathsf{P}^B \neq \mathsf{NP}^B$.
- So any proof that works under all relativizations cannot show $\mathsf{P} = \mathsf{NP}$ or $\mathsf{P} \neq \mathsf{NP}$.
- All the standard diagonalization arguments work under all relativizations.
- Hence, they are useless for proving $\mathsf{P} \neq \mathsf{NP}$!

The Relativization Barrier

- Baker, Gill and Solovay (1975) proved that there exists an algorithm A such that $\mathsf{P}^A = \mathsf{NP}^A$ and there exists an algorithm B such that $\mathsf{P}^B \neq \mathsf{NP}^B$.
- So any proof that works under all relativizations cannot show $\mathsf{P} = \mathsf{NP}$ or $\mathsf{P} \neq \mathsf{NP}$.
- All the standard diagonalization arguments work under all relativizations.
- Hence, they are useless for proving $\mathsf{P} \neq \mathsf{NP}$!

Outline

The Circuit Model of Computation

- Algorithms provide a dynamic view of computation.
- A static view of computation should be comparatively easier to analyze.
- This is provided by circuits.

The Circuit Model of Computation

- Algorithms provide a dynamic view of computation.
- A static view of computation should be comparatively easier to analyze.
- This is provided by circuits.

The Circuit Model of Computation

- Algorithms provide a dynamic view of computation.
- A static view of computation should be comparatively easier to analyze.
- This is provided by circuits.

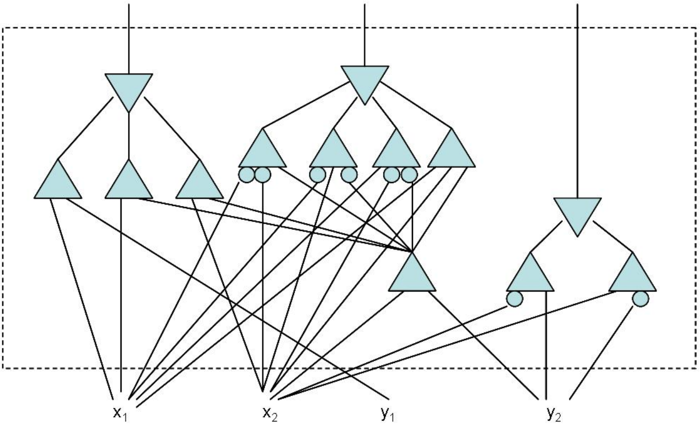
The Circuit Model of Computation

- Any algorithm is eventually executed by a computer consisting of electronic circuits.
- The working of these circuits on an input of size n can be viewed as a boolean circuit operating on n bits.

The Circuit Model of Computation

- Any algorithm is eventually executed by a computer consisting of electronic circuits.
- The working of these circuits on an input of size n can be viewed as a boolean circuit operating on n bits.

Circuit Adding Two 2 Bit Numbers



The Circuit Model of Computation

- Unlike an algorithm, a circuit can operate only on a fixed input size.
- Hence, for any problem, we need to use an infinite family of circuits to solve it.
- We only consider circuits consisting of AND, OR, and NOT gates.
- Both AND and OR gates can have any number of inputs.
- The size of a circuit is the number of gates in it.
- We measure the size as a function of input size.

The Circuit Model of Computation

- Unlike an algorithm, a circuit can operate only on a fixed input size.
- Hence, for any problem, we need to use an infinite family of circuits to solve it.
- We only consider circuits consisting of AND, OR, and NOT gates.
- Both AND and OR gates can have any number of inputs.
- The size of a circuit is the number of gates in it.
- We measure the size as a function of input size.

The Circuit Model of Computation

- Unlike an algorithm, a circuit can operate only on a fixed input size.
- Hence, for any problem, we need to use an infinite family of circuits to solve it.
- We only consider circuits consisting of AND, OR, and NOT gates.
- Both AND and OR gates can have any number of inputs.
- The size of a circuit is the number of gates in it.
- We measure the size as a function of input size.

Lower Bounds on Circuit Size

- It is easy to show that: A problem is in P iff it has a circuit family of size $n^{O(1)}$.
- So if we can show that a problem in NP does not have a circuit family of size $n^{O(1)}$, we have shown $P \neq NP$.
- This approach was initiated in 1980s and was considered to be very promising.
- It met with many initial successes.

Lower Bounds on Circuit Size

- It is easy to show that: A problem is in P iff it has a circuit family of size $n^{O(1)}$.
- So if we can show that a problem in NP does not have a circuit family of size $n^{O(1)}$, we have shown $P \neq NP$.
- This approach was initiated in 1980s and was considered to be very promising.
- It met with many initial successes.

Lower Bounds on Circuit Size

- It is easy to show that: A problem is in P iff it has a circuit family of size $n^{O(1)}$.
- So if we can show that a problem in NP does not have a circuit family of size $n^{O(1)}$, we have shown $\mathrm{P} \neq \mathrm{NP}$.
- This approach was initiated in 1980s and was considered to be very promising.
- It met with many initial successes.

Known Lower Bounds

- Razborov (1985) showed that there is a problem in NP that requires superpolynomial size monotone circuits.
  - Monotone circuits are circuits without NOT gates.
- Hastad (1986) showed that there is a problem in NP that requires superpolynomial size constant depth circuits.
  - Constant depth circuits are circuits such that the number of gates between any output and input line is a constant.
- While neither of the two results showed $P \neq NP$, they showed the promise of the approach.
- However, no further progress was made in the next 7-8 years.

Known Lower Bounds

- Razborov (1985) showed that there is a problem in NP that requires superpolynomial size monotone circuits.
  - Monotone circuits are circuits without NOT gates.
- Hastad (1986) showed that there is a problem in NP that requires superpolynomial size constant depth circuits.
  - Constant depth circuits are circuits such that the number of gates between any output and input line is a constant.
- While neither of the two results showed $P \neq NP$, they showed the promise of the approach.
- However, no further progress was made in the next 7-8 years.

Known Lower Bounds

- Razborov (1985) showed that there is a problem in NP that requires superpolynomial size monotone circuits.
  - Monotone circuits are circuits without NOT gates.
- Hastad (1986) showed that there is a problem in NP that requires superpolynomial size constant depth circuits.
  - Constant depth circuits are circuits such that the number of gates between any output and input line is a constant.
- While neither of the two results showed $P \neq NP$, they showed the promise of the approach.
- However, no further progress was made in the next 7-8 years.

Known Lower Bounds

- Razborov (1985) showed that there is a problem in NP that requires superpolynomial size monotone circuits.
  - Monotone circuits are circuits without NOT gates.
- Hastad (1986) showed that there is a problem in NP that requires superpolynomial size constant depth circuits.
  - Constant depth circuits are circuits such that the number of gates between any output and input line is a constant.
- While neither of the two results showed $P \neq NP$, they showed the promise of the approach.
- However, no further progress was made in the next 7-8 years.

Natural Proofs

- Razborov and Rudich (1994) defined the notion of natural proofs.
- These proofs refer to cerain types of lower bound proofs for circuits.
- These type of proofs have two properties:
  - Abundance: the lower bound can be proven with high probability by randomly picking a proof.
  - Easily verifiable: given a proof, it is easy to see if it is a correct proof.

Natural Proofs

- Razborov and Rudich (1994) defined the notion of natural proofs.
- These proofs refer to cerain types of lower bound proofs for circuits.
- These type of proofs have two properties:
  - Abundance: the lower bound can be proven with high probability by randomly picking a proof.
  - Easily verifiable: given a proof, it is easy to see if it is a correct proof.

The Natural Proof Barrier

- Razborov and Rudich showed that all the previous lower bound proofs on circuits are natural proofs.
- Also, if a widely believed conjecture is true, then natural proofs cannot be used to prove better lower bounds.
- This explained why no progress was made on cicuitr lower bounds!
- The conjecture they used was that pseudo-random generators exist.
- The next approach uses these!

The Natural Proof Barrier

- Razborov and Rudich showed that all the previous lower bound proofs on circuits are natural proofs.
- Also, if a widely believed conjecture is true, then natural proofs cannot be used to prove better lower bounds.
- This explained why no progress was made on cicuitr lower bounds!
- The conjecture they used was that pseudo-random generators exist.
- The next approach uses these!

The Natural Proof Barrier

- Razborov and Rudich showed that all the previous lower bound proofs on circuits are natural proofs.
- Also, if a widely believed conjecture is true, then natural proofs cannot be used to prove better lower bounds.
- This explained why no progress was made on cicuitr lower bounds!
- The conjecture they used was that pseudo-random generators exist.
- The next approach uses these!

Outline

Randomized Algorithms

- Many problems can be efficiently solved using a randomized algorithm.
- Such an algorithm tosses a few random coins during computation and uses their result to compute the solution with high probability.
- For example, finding a large prime number: randomly pick a large number and check if it is prime. Repeat a few times until a prime is found.

Randomized Algorithms

- Many problems can be efficiently solved using a randomized algorithm.
- Such an algorithm tosses a few random coins during computation and uses their result to compute the solution with high probability.
- For example, finding a large prime number: randomly pick a large number and check if it is prime. Repeat a few times until a prime is found.

Example: 3SAT

- A problem instance consists of m clauses, each over 3 variables.
- A clause is a disjunction of variables and their negations: $x_3 \vee \bar{x}_7 \vee x_9$.
- A variable can be either true or false.
- The problem is to determine an assignment to variables that make all clauses true.
- This problem is NP-complete: if it can be solved in P then NP = P.
- However, it is easy to find an assignment making at least $\frac{7}{8}m$ clauses true: randomly assign values to variables and see it this makes at least $\frac{7}{8}m$ clauses true.

Example: 3SAT

- A problem instance consists of m clauses, each over 3 variables.
- A clause is a disjunction of variables and their negations: $x_3 \vee \bar{x}_7 \vee x_9$.
- A variable can be either true or false.
- The problem is to determine an assignment to variables that make all clauses true.
- This problem is NP-complete: if it can be solved in P then NP = P.
- However, it is easy to find an assignment making at least $\frac{7}{8}m$ clauses true: randomly assign values to variables and see it this makes at least $\frac{7}{8}m$ clauses true.

Example: 3SAT

- A problem instance consists of m clauses, each over 3 variables.
- A clause is a disjunction of variables and their negations: $x_3 \vee \bar{x}_7 \vee x_9$.
- A variable can be either true or false.
- The problem is to determine an assignment to variables that make all clauses true.
- This problem is NP-complete: if it can be solved in P then NP = P.
- However, it is easy to find an assignment making at least $\frac{7}{8}m$ clauses true: randomly assign values to variables and see it this makes at least $\frac{7}{8}m$ clauses true.

Example: 3SAT

- Under this assignment, each clause will be true with probability exactly $\frac{7}{8}$.
- Hence, expected number of true clauses will be exactly $\frac{7}{8}m$.
- This implies that with probability at least $\frac{1}{2}$, a random assignment will make at least $\frac{7}{8}m$ clauses true.

Example: 3SAT

- Under this assignment, each clause will be true with probability exactly $\frac{7}{8}$.
- Hence, expected number of true clauses will be exactly $\frac{7}{8}m$.
- This implies that with probability at least $\frac{1}{2}$, a random assignment will make at least $\frac{7}{8}m$ clauses true.

Example: 3SAT

- Under this assignment, each clause will be true with probability exactly $\frac{7}{8}$.
- Hence, expected number of true clauses will be exactly $\frac{7}{8}m$.
- This implies that with probability at least $\frac{1}{2}$, a random assignment will make at least $\frac{7}{8}m$ clauses true.

Generating Random Bits

- In practice, however, there is no way to generate random bits without using quantum measurements.
- So how does one provide "coin tossing" operation to such algorithms?
- A good way is to provide a sequence of bits to the algorithm that appear random to it.
- In other words, this sequence of bits fools the algorithm into believing that it is random sequence.
- This is not possible if the algorithm has enough time to differentiate it from a random sequence.
- However, the algorithm is efficient, and so has only polynomial time available.
- So this limitation can be turned against it!

Generating Random Bits

- In practice, however, there is no way to generate random bits without using quantum measurements.
- So how does one provide "coin tossing" operation to such algorithms?
- A good way is to provide a sequence of bits to the algorithm that appear random to it.
- In other words, this sequence of bits fools the algorithm into believing that it is random sequence.
- This is not possible if the algorithm has enough time to differentiate it from a random sequence.
- However, the algorithm is efficient, and so has only polynomial time available.
- So this limitation can be turned against it!

Generating Random Bits

- In practice, however, there is no way to generate random bits without using quantum measurements.
- So how does one provide "coin tossing" operation to such algorithms?
- A good way is to provide a sequence of bits to the algorithm that appear random to it.
- In other words, this sequence of bits fools the algorithm into believing that it is random sequence.
- This is not possible if the algorithm has enough time to differentiate it from a random sequence.
- However, the algorithm is efficient, and so has only polynomial time available.
- So this limitation can be turned against it!

Generating Random Bits

- In practice, however, there is no way to generate random bits without using quantum measurements.
- So how does one provide "coin tossing" operation to such algorithms?
- A good way is to provide a sequence of bits to the algorithm that appear random to it.
- In other words, this sequence of bits fools the algorithm into believing that it is random sequence.
- This is not possible if the algorithm has enough time to differentiate it from a random sequence.
- However, the algorithm is efficient, and so has only polynomial time available.
- So this limitation can be turned against it!

Pseudo-random generators

- Pseudo-random generators are algorithms that produce seemingly random bits which fool a whole class of algorithms.
- The strength of a pseudo-random generator is determined by how much real randomness they need to produce their output, and what class of algorithms they fool.
- Idea developed in 1990s.
- Has become a fundamental concept in theory of computation.

Pseudo-random generators

- Pseudo-random generators are algorithms that produce seemingly random bits which fool a whole class of algorithms.
- The strength of a pseudo-random generator is determined by how much real randomness they need to produce their output, and what class of algorithms they fool.
- Idea developed in 1990s.
- Has become a fundamental concept in theory of computation.

Pseudo-random generators

- Pseudo-random generators are algorithms that produce seemingly random bits which fool a whole class of algorithms.
- The strength of a pseudo-random generator is determined by how much real randomness they need to produce their output, and what class of algorithms they fool.
- Idea developed in 1990s.
- Has become a fundamental concept in theory of computation.

Example: Pseudo-random Generator Fooling 3SAT Algorithm

- Instead of using random values for variables, pick them in 3-wise independent fashion.
- This guarantees that each clause will be true with probability exactly $\frac{7}{8}$.
- The expected number of true clauses will remain the same by linearity of expectation principle.
- How does one generate 3-wise independent assignment?

Example: Pseudo-random Generator Fooling 3SAT Algorithm

- Instead of using random values for variables, pick them in 3-wise independent fashion.
- This guarantees that each clause will be true with probability exactly $\frac{7}{8}$.
- The expected number of true clauses will remain the same by linearity of expectation principle.
- How does one generate 3-wise independent assignment?

3-Wise Independent Source

- Fix a finite field F of size 2^k with $n \leq 2^k < 2n$ (n is the number of variables).
- Pick 3 elements a, b, c randomly from F.
- Let $e_1, \ldots, e_n$ be n distinct elements of F.
- Define $d_i = a \cdot e_i^2 + b \cdot e_i + c$.
- If the first bit of d_i is 0, assign variable x_i value false, else true.

3-Wise Independent Source

- Fix a finite field F of size 2^k with $n \leq 2^k < 2n$ (n is the number of variables).
- Pick 3 elements a, b, c randomly from F.
- Let $e_1, \ldots, e_n$ be n distinct elements of F.
- Define $d_i = a \cdot e_i^2 + b \cdot e_i + c$.
- If the first bit of d_i is 0, assign variable x_i value false, else true.

3-Wise Independent Source

- Fix a finite field F of size 2^k with $n \leq 2^k < 2n$ (n is the number of variables).
- Pick 3 elements a, b, c randomly from F.
- Let $e_1, \ldots, e_n$ be n distinct elements of F.
- Define $d_i = a \cdot e_i^2 + b \cdot e_i + c$.
- If the first bit of d_i is 0, assign variable x_i value false, else true.

Derandomizing 3SAT Algorithm

- This assignment results in exactly the same property: with probability at least $\frac{1}{2}$, an assignment will make at least $\frac{7}{8}m$ clauses true.
- But this still requires randomness (in choosing a, b and c).
- Recall: F is such that $|F| = 2^k \leq 2n$.
- Hence, the number of possibilites for a are $2n$ (same for b and c).
- So we can try out all possibilities (at most $8n^3$) for these!
- We will find at least half of them to be "good" ones for us.
- Therefore we get a deterministic algorithm that efficiently solves the problem.

Derandomizing 3SAT Algorithm

- This assignment results in exactly the same property: with probability at least $\frac{1}{2}$, an assignment will make at least $\frac{7}{8}m$ clauses true.
- But this still requires randomness (in choosing a, b and c).
- Recall: F is such that $|F| = 2^k \leq 2n$.
- Hence, the number of possibilites for a are $2n$ (same for b and c).
- So we can try out all possibilities (at most $8n^3$) for these!
- We will find at least half of them to be "good" ones for us.
- Therefore we get a deterministic algorithm that efficiently solves the problem.

Derandomizing 3SAT Algorithm

- This assignment results in exactly the same property: with probability at least $\frac{1}{2}$, an assignment will make at least $\frac{7}{8}m$ clauses true.
- But this still requires randomness (in choosing a, b and c).
- Recall: F is such that $|F| = 2^k \leq 2n$.
- Hence, the number of possibilites for a are $2n$ (same for b and c).
- So we can try out all possibilities (at most $8n^3$) for these!
- We will find at least half of them to be "good" ones for us.
- Therefore we get a deterministic algorithm that efficiently solves the problem.

Derandomizing 3SAT Algorithm

- This assignment results in exactly the same property: with probability at least $\frac{1}{2}$, an assignment will make at least $\frac{7}{8}m$ clauses true.
- But this still requires randomness (in choosing a, b and c).
- Recall: F is such that $|F| = 2^k \leq 2n$.
- Hence, the number of possibilites for a are $2n$ (same for b and c).
- So we can try out all possibilities (at most $8n^3$) for these!
- We will find at least half of them to be "good" ones for us.
- Therefore we get a deterministic algorithm that efficiently solves the problem.

Formal Definition

Definition

Function f is an optimal pseudo-random generator if:

- f maps $c \log n$ bit input to n bit output, c is a fixed constant,
- Every output bit can be computed in time $\log^{O(1)} n$,
- For every circuit C of size n on n inputs:

$$| \Pr_x[C(x) = 1] - \Pr_y[C(f(y)) = 1] | \leq \frac{1}{n}.$$

Derandomization

Theorem

If optimal pseudo-random generators exist then all problems that can be solved using efficient randomized algorithms are in P.

- Randomized efficient algorithms can be viewed as small sized circuits with random bits as inputs.
- These circuits can be made to output 1 or 0 depending on whether the solution has been found.
- Replacing the random bits with the output of an optimal pseudo-random generator will not change the probability of finding a solution by much.
- Finally, one can go through all possible $c \log n$ inputs to the generator to find one that will yield a solution.

Derandomization

Theorem

If optimal pseudo-random generators exist then all problems that can be solved using efficient randomized algorithms are in P.

- Randomized efficient algorithms can be viewed as small sized circuits with random bits as inputs.
- These circuits can be made to output 1 or 0 depending on whether the solution has been found.
- Replacing the random bits with the output of an optimal pseudo-random generator will not change the probability of finding a solution by much.
- Finally, one can go through all possible $c \log n$ inputs to the generator to find one that will yield a solution.

Derandomization

Theorem

If optimal pseudo-random generators exist then all problems that can be solved using efficient randomized algorithms are in P.

- Randomized efficient algorithms can be viewed as small sized circuits with random bits as inputs.
- These circuits can be made to output 1 or 0 depending on whether the solution has been found.
- Replacing the random bits with the output of an optimal pseudo-random generator will not change the probability of finding a solution by much.
- Finally, one can go through all possible $c \log n$ inputs to the generator to find one that will yield a solution.

Lower Bounds

It was proved by Nisan and Wigderson (1989) that:

Theorem

If optimal pseudo-random generators exist then $P \neq NP$.

Current Status

- This approach does not suffer from the natural proof barrier.
- It will have to cross relativization barrier since an algorithm defining a generator must be non-relativizable.
- The aim here is to find an efficient algorithm for a problem.
- And this shows that no efficient algorithm exists for a number of other problems!
- Over the last few years, generators have been defined that fool special classes of circuits.

Current Status

- This approach does not suffer from the natural proof barrier.
- It will have to cross relativization barrier since an algorithm defining a generator must be non-relativizable.
- The aim here is to find an efficient algorithm for a problem.
- And this shows that no efficient algorithm exists for a number of other problems!
- Over the last few years, generators have been defined that fool special classes of circuits.

Current Status

- This approach does not suffer from the natural proof barrier.
- It will have to cross relativization barrier since an algorithm defining a generator must be non-relativizable.
- The aim here is to find an efficient algorithm for a problem.
- And this shows that no efficient algorithm exists for a number of other problems!
- Over the last few years, generators have been defined that fool special classes of circuits.

Current Status

- This approach does not suffer from the natural proof barrier.
- It will have to cross relativization barrier since an algorithm defining a generator must be non-relativizable.
- The aim here is to find an efficient algorithm for a problem.
- And this shows that no efficient algorithm exists for a number of other problems!
- Over the last few years, generators have been defined that fool special classes of circuits.

Is there a barrier out there against this approach too?

OR

Is this the right approach for proving P $\neq$ NP?