Intractability: P and NP

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 11

Checking algorithms

- Checking algorithm C for problem
- Takes in an input instance I and a solution “certificate” S for I
- C outputs yes if S represents a valid solution for I, no otherwise

The class NP

- Checking algorithm C that verifies a solution S for input instance I in time polynomial in size(I)
  - Factorization, satisfiability, travelling salesman, vertex cover, independent set, ... are all in NP

The class NP

- Checking algorithm C that verifies a solution S for input instance I in time polynomial in size(I)
  - Factorization, satisfiability, travelling salesman, vertex cover, independent set, ... are all in NP
- If we convert an optimization problem to a checking problem by providing a bound, we add only a log factor
  - Binary search through solution space

So these are also included in NP class

The class NP

- Checking algorithm C that verifies a solution S for input instance I in time polynomial in size(I)
  - Factorization, satisfiability, travelling salesman, vertex cover, independent set, ... are all in NP
- If we convert an optimization problem to a checking problem by providing a bound, we add only a log factor
  - Binary search through solution space

Why "NP"?

The class NP

- Checking algorithm C that verifies a solution S for input instance I in time polynomial in size(I)
  - Factorization, satisfiability, travelling salesman, vertex cover, independent set, ... are all in NP
- If we convert an optimization problem to a checking problem by providing a bound, we add only a log factor
  - Binary search through solution space

Why "NP"?

- Non-deterministic Polynomial time

The class NP

- Checking algorithm C that verifies a solution S for input instance I in time polynomial in size(I)
  - Factorization, satisfiability, travelling salesman, vertex cover, independent set, ... are all in NP
- If we convert an optimization problem to a checking problem by providing a bound, we add only a log factor
  - Binary search through solution space

Why "NP"?

- Non-deterministic Polynomial time
- "Guess" a solution and check it

The class NP

- Checking algorithm C that verifies a solution S for input instance I in time polynomial in size(I)
  - Factorization, satisfiability, travelling salesman, vertex cover, independent set, ... are all in NP
- If we convert an optimization problem to a checking problem by providing a bound, we add only a log factor
  - Binary search through solution space

Why "NP"?

- Non-deterministic Polynomial time
- "Guess" a solution and check it
- Origins in computability theory

The class NP

- Checking algorithm C that verifies a solution S for input instance I in time polynomial in size(I)
  - Factorization, satisfiability, travelling salesman, vertex cover, independent set, . . . are all in NP
- If we convert an optimization problem to a checking problem by providing a bound, we add only a log factor
  - Binary search through solution space

Why "NP"?

- Non-deterministic Polynomial time
- "Guess" a solution and check it
- Origins in computability theory
- Non-deterministic Turing machines . . .

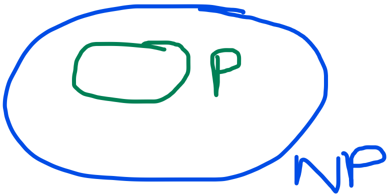
P and NP

- P is the class of problems with regular polynomial time algorithms (worst-case complexity)

P is the class of problems for which we have efficient solutions

Examples: Binary search, sorting algorithms like merge sort, quick sort etc

- P is the class of problems with regular polynomial time algorithms (worst-case complexity)
- P is included in NP — generate a solution and check it!



This is because if I can generate solution in polynomial implies I can check the solution in polynomial time thus P is included in NP

P and NP

- P is the class of problems with regular polynomial time algorithms (worst-case complexity)
- P is included in NP — generate a solution and check it!
- Is P = NP?

So does every algorithm which has efficient checking solution also have an efficient generation?

P and NP

- P is the class of problems with regular polynomial time algorithms (worst-case complexity)
- P is included in NP — generate a solution and check it!
- Is P = NP?
- Is efficient checking same as efficient generation?

P and NP

- P is the class of problems with regular polynomial time algorithms (worst-case complexity)
- P is included in NP — generate a solution and check it!
- Is P = NP?
- Is efficient checking same as efficient generation?
- Intuitively this should not be the case

Example:
Teacher: Given product of two very large primes, find primes. Easy to check solution
Student: Hard to find solution

P and NP

- P is the class of problems with regular polynomial time algorithms (worst-case complexity)
- P is included in NP — generate a solution and check it!
- Is P $=$ NP?
- Is efficient checking same as efficient generation?
- Intuitively this should not be the case

P $\neq$ NP?

intuitively checking is easy and generation is hard

P and NP

- P is the class of problems with regular polynomial time algorithms (worst-case complexity)
- P is included in NP — generate a solution and check it!
- Is P $=$ NP?
- Is efficient checking same as efficient generation?
- Intuitively this should not be the case

P $\neq$ NP?

- A more formal reason to believe this?

P and NP

- P is the class of problems with regular polynomial time algorithms (worst-case complexity)
- P is included in NP — generate a solution and check it!
- Is P $=$ NP?
- Is efficient checking same as efficient generation?
- Intuitively this should not be the case

$P \neq NP$?

- A more formal reason to believe this?
- Many "natural" problems are in NP
  - Factorization, satisfiability, travelling salesman, vertex cover, independent set, ...

P and NP

- P is the class of problems with regular polynomial time algorithms (worst-case complexity)
- P is included in NP — generate a solution and check it!
- Is P = NP?
- Is efficient checking same as efficient generation?
- Intuitively this should not be the case

P $\neq$ NP?

- A more formal reason to believe this?
- Many "natural" problems are in NP
  - Factorization, satisfiability, travelling salesman, vertex cover, independent set, . . .
- These are all inter-reducible,
  - Like vertex cover, independent set

One problem can be reduced to the other (inter-reducible)

Example: Finding min vertex cover can be reduced to finding max independent set

P and NP

- P is the class of problems with regular polynomial time algorithms (worst-case complexity)
- P is included in NP — generate a solution and check it!
- Is P = NP?
- Is efficient checking same as efficient generation?
- Intuitively this should not be the case

P $\neq$ NP?

- A more formal reason to believe this?
- Many "natural" problems are in NP
  - Factorization, satisfiability, travelling salesman, vertex cover, independent set, ...
- These are all inter-reducible,
  - Like vertex cover, independent set
- If we can solve one efficiently, we can solve them all!

Boolean satisfiability

- Boolean variables $x_1, x_2, x_3, \ldots$

Boolean satisfiability

- Boolean variables $x_1, x_2, x_3, \ldots$
- Clause — disjunction of literals, $(x_1 \vee \neg x_2 \vee x_3 \vee \cdots \vee x_k)$

Disjunction - Logical "OR" operator
Conjunction - Logical "AND" operator

Boolean satisfiability

- Boolean variables $x_1, x_2, x_3, \ldots$
- Clause — disjunction of literals, $(x_1 \lor \neg x_2 \lor x_3 \lor \cdots \lor x_k)$
- Formula — conjunction of clauses, $C_1 \land C_2 \land \cdots \land C_m$

Boolean satisfiability

- Boolean variables $x_1, x_2, x_3, \ldots$
- Clause — disjunction of literals, $(x_1 \vee \neg x_2 \vee x_3 \vee \cdots \vee x_k)$
- Formula — conjunction of clauses, $C_1 \wedge C_2 \wedge \cdots \wedge C_m$
- 3-SAT — each clause has at most 3 literals

3-SAT
3: Maximum number of literals per clause
SAT: Satisfiability

So if we constraint the size of clause, in other words if we set a limit on maximum number of literals per clause then it is called n-SAT

Boolean satisfiability

- Boolean variables $x_1, x_2, x_3, \ldots$
- Clause — disjunction of literals, $(x_1 \vee \neg x_2 \vee x_3 \vee \cdots \vee x_k)$
- Formula — conjunction of clauses, $C_1 \wedge C_2 \wedge \cdots \wedge C_m$
- 3-SAT — each clause has at most 3 literals

Reducing SAT to 3-SAT

1. So what if reduce the SAT problem to 3-SAT

2. We want to reduce it in such a way that satisfiability doesn't change

3. But we also know that if SAT is difficult to solve then reducing it to 3-SAT means 3-SAT will also be difficult to solve

Boolean satisfiability

- Boolean variables $x_1, x_2, x_3, \ldots$
- Clause — disjunction of literals, $(x_1 \lor \neg x_2 \lor x_3 \lor \cdots \lor x_k)$
- Formula — conjunction of clauses, $C_1 \land C_2 \land \cdots \land C_m$
- 3-SAT — each clause has at most 3 literals

Reducing SAT to 3-SAT

- Consider a 5 literal clause $(v \lor \neg w \lor x \lor \neg y \lor z)$

1. We want to convert this to 3-SAT

2. But also ensure that satisfiability remains unchanged

Boolean satisfiability

- Boolean variables $x_1, x_2, x_3, \ldots$
- Clause — disjunction of literals, $(x_1 \vee \neg x_2 \vee x_3 \vee \cdots \vee x_k)$
- Formula — conjunction of clauses, $C_1 \wedge C_2 \wedge \cdots \wedge C_m$
- 3-SAT — each clause has at most 3 literals

Thus overall satisfiability remains the same

Reducing SAT to 3-SAT

- Consider a 5 literal clause
  $(v \vee \neg w \vee x \vee \neg y \vee z)$
- Introduce a new literal and split the clause
  $(v \vee \neg w \vee a) \wedge (\neg a \vee x \vee \neg y \vee z)$

This is equivalent to the original

If a= True then left part can be anything (T/F) but the right part must be True

If a= False then right part can be anything (T/F) but the left part must be True

Boolean satisfiability

- Boolean variables $x_1, x_2, x_3, \ldots$
- Clause — disjunction of literals, $(x_1 \lor \neg x_2 \lor x_3 \lor \cdots \lor x_k)$
- Formula — conjunction of clauses, $C_1 \land C_2 \land \cdots \land C_m$
- 3-SAT — each clause has at most 3 literals

Reducing SAT to 3-SAT

- Consider a 5 literal clause $(v \lor \neg w \lor x \lor \neg y \lor z)$
- Introduce a new literal and split the clause $(v \lor \neg w \lor a) \land (\neg a \lor x \lor \neg y \lor z)$
- This formula is satisfiable iff original clause is

Boolean satisfiability

- Boolean variables $x_1, x_2, x_3, \ldots$
- Clause — disjunction of literals, $(x_1 \vee \neg x_2 \vee x_3 \vee \cdots \vee x_k)$
- Formula — conjunction of clauses, $C_1 \wedge C_2 \wedge \cdots \wedge C_m$
- 3-SAT — each clause has at most 3 literals

Reducing SAT to 3-SAT

- Consider a 5 literal clause $(v \vee \neg w \vee x \vee \neg y \vee z)$
- Introduce a new literal and split the clause $(v \vee \neg w \vee a) \wedge (\neg a \vee x \vee \neg y \vee z)$
- This formula is satisfiable iff original clause is
- Repeat till all clauses are of size 3 or less

$(v \vee \neg w \vee a) \wedge (\neg a \vee x \vee b) \wedge (\neg b \vee \neg y \vee z)$

Boolean satisfiability

- Boolean variables $x_1, x_2, x_3, \ldots$
- Clause — disjunction of literals, $(x_1 \vee \neg x_2 \vee x_3 \vee \cdots \vee x_k)$
- Formula — conjunction of clauses, $C_1 \wedge C_2 \wedge \cdots \wedge C_m$
- 3-SAT — each clause has at most 3 literals

Reducing SAT to 3-SAT

- Consider a 5 literal clause
  $(v \vee \neg w \vee x \vee \neg y \vee z)$
- Introduce a new literal and split the clause
  $(v \vee \neg w \vee a) \wedge (\neg a \vee x \vee \neg y \vee z)$
- This formula is satisfiable iff original clause is
- Repeat till all clauses are of size 3 or less

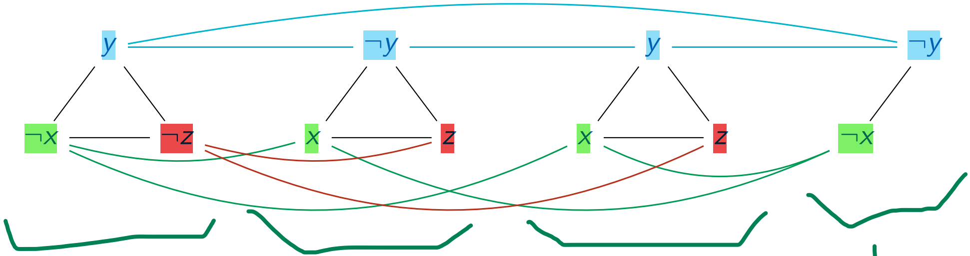
  $(v \vee \neg w \vee a) \wedge (\neg a \vee x \vee b) \wedge (\neg b \vee \neg y \vee z)$
- If SAT is hard, so is 3-SAT

3-SAT to independent set

Claim: We can reduce a 3-SAT problem to independent set problem

- Construct a graph from a 3-SAT formula

$$(\neg x \lor y \lor \neg z) \land (x \lor \neg y \lor z) \land (x \lor y \lor z) \land (\neg x \lor \neg y)$$



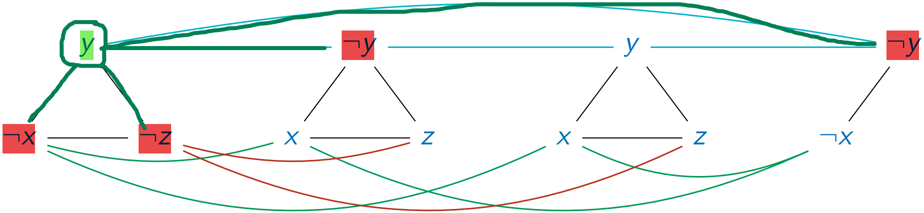
Clause: 1 2 3 4

4 Clauses

Connect each literal with its negation

- Construct a graph from a 3-SAT formula

$$(\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (x \vee y \vee z) \wedge (\neg x \vee \neg y)$$



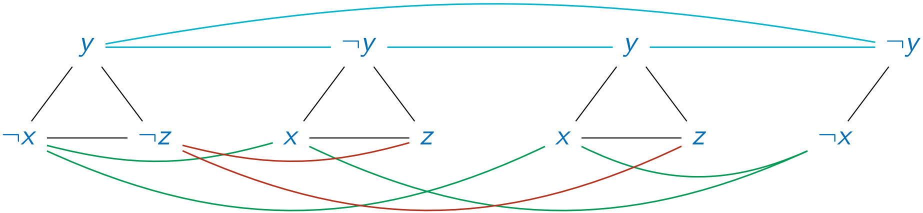
- Independent set picks one literal per clause to satisfy

1. This means in total we can pick at most 4 literals.
2. Consider picking a literal as setting it to True
3. Example: See what all literals you cannot pick (or set True) when you pick y of first clause

3-SAT to independent set

- Construct a graph from a 3-SAT formula

$$(\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (x \vee y \vee z) \wedge (\neg x \vee \neg y)$$



- Independent set picks one literal per clause to satisfy

Remember picking is equivalent to setting that literal to True

- Edges enforce consistency across clauses

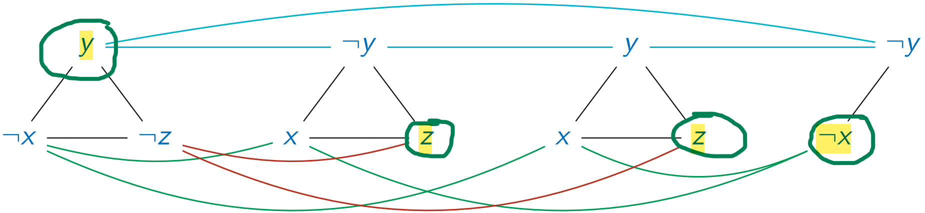
And this is how we can solve the problem of 3-SAT by solving independent set

3-SAT to independent set

- Construct a graph from a 3-SAT formula

$$(\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (x \vee y \vee z) \wedge (\neg x \vee \neg y)$$ = True

y = True
z = True
x = False



Independent set

- Independent set picks one literal per clause to satisfy
- Edges enforce consistency across clauses

Reductions within NP

- SAT $\rightarrow$ 3-SAT, 3-SAT $\rightarrow$ independent set, independent set $\leftrightarrow$ vertex cover
- Reduction is transitive, so SAT $\rightarrow$ vertex cover, ...
- Other inter-reducible NP problems
  - Travelling salesman, integer linear programming ... All these problems are "equally" hard

NP-Completeness

Cook-Levin Theorem

Every problem in NP can be reduced to SAT

As SAT is hard to solve this implies that a NP problem is also hard to solve

- Original proof is by encoding computations of Turing machines

NP-Completeness

Cook-Levin Theorem

Every problem in NP can be reduced to SAT

- Original proof is by encoding computations of Turing machines
- Can replace by encoding of any "generic" computation model — boolean circuits, register machines ...

NP-Completeness

Cook-Levin Theorem

Every problem in NP can be reduced to SAT

- Original proof is by encoding computations of Turing machines
- Can replace by encoding of any "generic" computation model — boolean circuits, register machines ...

- SAT is said to be complete for NP

NP-Completeness

Cook-Levin Theorem

Every problem in NP can be reduced to SAT

- Original proof is by encoding computations of Turing machines
- Can replace by encoding of any "generic" computation model — boolean circuits, register machines . . .

- SAT is said to be complete for NP
  - It belongs to NP

NP-Completeness

Cook-Levin Theorem

Every problem in NP can be reduced to SAT

- Original proof is by encoding computations of Turing machines
- Can replace by encoding of any "generic" computation model — boolean circuits, register machines ...

- SAT is said to be complete for NP
  - It belongs to NP
  - Every problem in NP reduces to it

NP-Completeness

Cook-Levin Theorem

Every problem in NP can be reduced to SAT

- Original proof is by encoding computations of Turing machines
- Can replace by encoding of any "generic" computation model — boolean circuits, register machines ...

- SAT is said to be complete for NP
  - It belongs to NP
  - Every problem in NP reduces to it
- Since SAT reduces to 3-SAT, 3-SAT is also NP-complete

NP-Completeness

Cook-Levin Theorem

Every problem in NP can be reduced to SAT

- Original proof is by encoding computations of Turing machines
- Can replace by encoding of any "generic" computation model — boolean circuits, register machines ...

- SAT is said to be complete for NP
  - It belongs to NP
  - Every problem in NP reduces to it
- Since SAT reduces to 3-SAT, 3-SAT is also NP-complete
- In general, to show P is NP-complete, reduce some existing NP-complete problem to P

$P \neq NP$?

- A large class of practically useful problems are NP-complete
    - Scheduling, bin-packing, optimal tours ...

$P \neq NP$?

- A large class of practically useful problems are NP-complete
  - Scheduling, bin-packing, optimal tours ...
- If one of them has a solution in P, all of them do

$P \neq NP$?

- A large class of practically useful problems are NP-complete
  - Scheduling, bin-packing, optimal tours ...
- If one of them has a solution in P, all of them do
- Many smart people have been working on these problems for centuries

$P \neq NP$?

- A large class of practically useful problems are NP-complete
  - Scheduling, bin-packing, optimal tours ...
- If one of them has a solution in P, all of them do
- Many smart people have been working on these problems for centuries
- Empirical evidence that NP is different from P

$P \neq NP$?

- A large class of practically useful problems are NP-complete
  - Scheduling, bin-packing, optimal tours ...
- If one of them has a solution in P, all of them do
- Many smart people have been working on these problems for centuries
- Empirical evidence that NP is different from P
- But a formal proof is elusive, and worth $1 million!