

Lecture 2: Algorithm Complexity

Tsung-Wei (TW) Huang

Department of Electrical and Computer Engineering

University of Utah, Salt Lake City, UT



Recap: We Need Algorithms!

- To optimize design among different objectives, area, power, performance, and etc.
- **Fundamental questions: How to do it smartly?**
- Definition of algorithm in a board sense: A step-by-step procedure for solving a problem. Examples:
 - Cooking a dish
 - Making a phone call
 - Sorting a hand of cards
- Definition for computational problem
 - A well-defined computational procedure that takes some value as input and produces some value as output

Recap: Big Mod Problem

- **Computational problem description**

- Input: a, b, c ($3 < a, c < 50000000, 1 < b < 2147483647$)
- Output: $x = a^b \pmod{c}$

- **A simple solution**

```
for(i=1, x=1; i<=b; i++)  
    x = (x*a)%c;  
std::cout << "a^b % c = " << x << std::endl;
```

- **Is the solution smart enough?**

- We need a total of “b” iterations
- Very slow execution for large b

Recap: Recursive Solution

- Input: $a=2$, $b=16$, c ; Output: $x = a^b \pmod{c}$
- **Simple solution needs a total of $b=16$ iterations**
- Let's refine the iteration count recursively

$$2^{16} = (2^8)^2 \quad \text{total 1 calculation}$$

$$2^8 = (2^4)^2 \quad \text{total 1 calculation}$$

$$2^4 = (2^2)^2 \quad \text{total 1 calculation}$$

$$2^2 = (2^1)^2 \quad \text{total 1 calculation}$$

$$2^1 = (2^0) \quad \text{total 1 calculation}$$

Recap: Recursive Solution (cont'd)

```
#define SQUARE(x) (x*x)
int bigmod(int a, int b, int c) {
    if(b==0) return 1;
    else if(b%2==0)
        return SQUARE(bigmod(a,b/2,c)%c)%c;
    else return ((a%c)*bigmod(a,b-1,c))%c;
}

int main() {
    int a,b,c;
    while(std::cin >> a >> b >> c) std::cout<<bigmod(a,b,c)<<endl;
    return 0;
}
```

$2^{16} = (2^8)^2$	total 1 calculation
$2^8 = (2^4)^2$	total 1 calculation
$2^4 = (2^2)^2$	total 1 calculation
$2^2 = (2^1)^2$	total 1 calculation
$2^1 = (2^0)$	total 1 calculation

Recap: Iterative Version

```
int bigmod(int a, int b, int c) {  
    int result = 1;  
    while(b > 0) {  
        if(b & 1) {  
            result = (result * a) % c;  
        }  
        b = b >> 1;  
        a = a*a%c;  
    }  
    return result;  
}
```

Recap: Runtime Comparison

- <https://www.onlinegdb.com/>

Runtime (with -O2) – Enabled Optimization			
(a, b, c)	Naive	Refined (recursive)	Refined (iterative)
(2, 1000000, 3)	13720 us	24 us	22 us
(2, 10000000, 3)	136148 us	9 us	4 us
(2, 100000000, 3)	1320049 us	8 us	4 us

Recap: Code Analysis

- <https://godbolt.org/z/6zT4cK3Wz>

	Naive	Refined (recursive)	Refined (iterative)
Lines of C++ Code	7	11	11
Lines of Assembly (-O0)	55	99	59
Lines of Assembly (-O2)	27	57	32

Computational Complexity

- **Computational complexity** is an abstract measure of the time and space necessary to execute an algorithm as function of its input size
 - The input is the graph $G(V,E)$
 - input size = $|V|$ and $|E|$
 - The input is the truth table of an n -variable Boolean function
 - input size = 2^n
- What's the growth of algorithm complexity in terms of input size?
 - Best case?
 - Worst case?

Time and Space Complexity

- **Time complexity** describes elementary computational steps
 - Ex: addition (or multiplication, or value assignment etc.) is one step
 - Normally, by "most time-efficient" algorithm we mean the fastest runtime to reach the goal
- **Space complexity** describes memory usage
 - Ex: in bits, bytes, words
 - Normally, by "most space-efficient" algorithm we mean the lowest memory usage

Analysis of an Algorithm

- **We need some way to compare different algorithms**
 - Usually the run time is the criteria used
 - However, difficult to compare since algorithms may be implemented in different machines, use different languages, etc.
 - Also, run time is input-dependent.
 - Which input to use?
 - Which case do we consider? Worst or best?
 - ...



Big-O Notation

- **Consider runtime for the **worst-case** input**
 - upper bound on runtime
- **Express runtime as a function of input size n**
 - Interested in the run time for large inputs
 - Therefore, interested in the growth rate
- **Ignore arithmetic operations**
 - $+, -, \times, /$, ... operators treated as constant-time computations
- **Ignore lower order terms**
 - Eventually, the highest order dominates the growth

Big-O Notation (cont'd)

- $g = O(f)$, if two constants n_0 and K can be found such that for all $n \geq n_0$:

$$g(n) \leq K \cdot f(n)$$

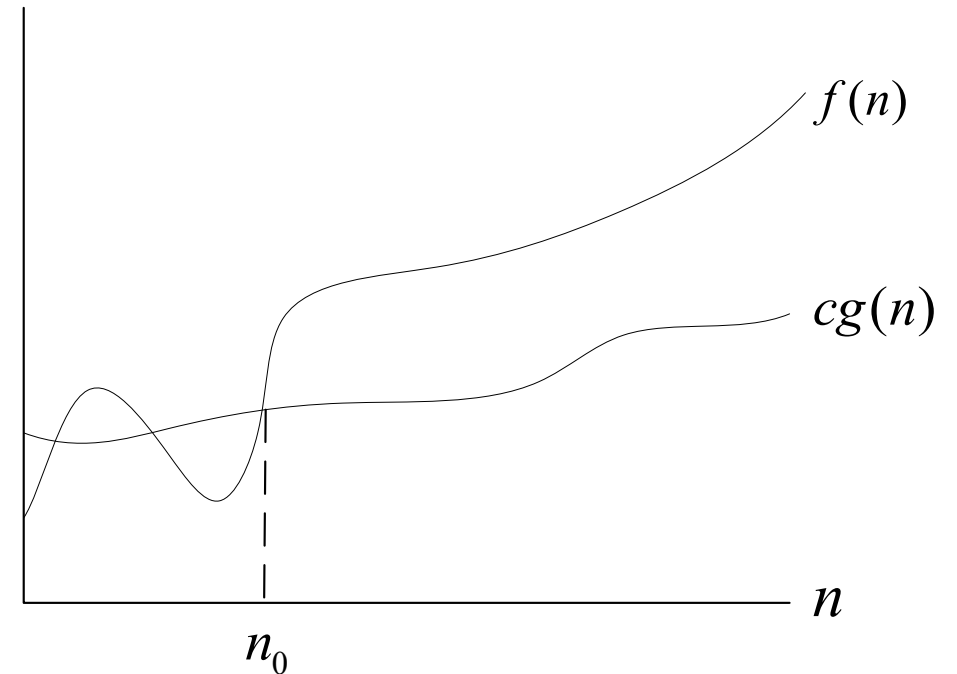
- Examples:

$$2n^2 = O(n^2)$$

$$2n^2 + 3n + 1 = O(n^2)$$

$$n^{1.1} + 100000000000n \text{ is } O(n^{1.1})$$

$$n^{1.1} = O(n^2)$$



Example: Big-Mod Complexity

- Compute $a^b \bmod c$ (b is as large as 2147483647)

- Naïve method

for($i=1, j=1; i \leq b; i++$)

$j = (j * a) \% c;$

- $2^{16} = 2 * 2 * 2 * 2 * 2 * 2 * \dots * 2$ total 16 iterations

- Refined solution

• $2^{16} = (2^8)^2$ total 1 iteration

• $2^8 = (2^4)^2$ total 1 iteration

• $2^4 = (2^2)^2$ total 1 iteration

• $2^2 = (2^1)^2$ total 1 iteration

• $2^1 = (2^0)$ total 1 iteration

What is the space and time complexities for each method in terms of big-O notation?

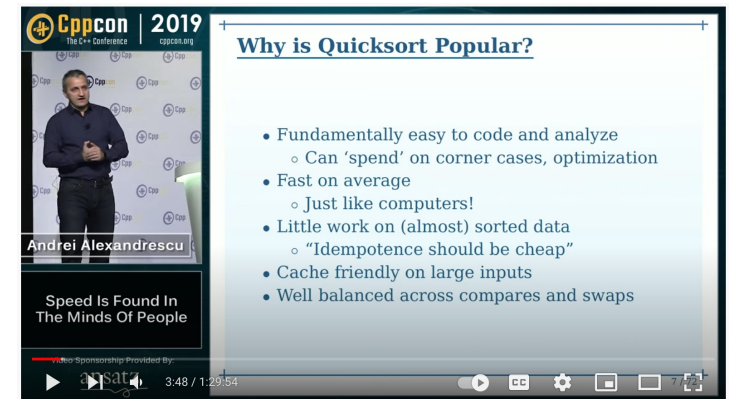
Example: Selection Sort

- Input: An array of n numbers $D[1] \dots D[n]$
- Output: An array of n numbers $E[1] \dots E[n]$ such that $E[1] \geq E[2] \geq \dots \geq E[n]$
- Algorithm:
 - For i from 1 to n do
 - Select the largest remaining number from $D[1..n]$
 - Put that number into $E[i]$

What is the space and time complexities for each method in terms of big-O notation?

Time Complexity of Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
<u>Quicksort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(\log(n))$
<u>Mergesort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
<u>Timsort</u>	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
<u>Heapsort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$
<u>Bubble Sort</u>	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Insertion Sort</u>	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Selection Sort</u>	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Tree Sort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(n)$
<u>Shell Sort</u>	$\Omega(n \log(n))$	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	$O(1)$
<u>Bucket Sort</u>	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$
<u>Radix Sort</u>	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
<u>Counting Sort</u>	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$
<u>Cubesort</u>	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$



Andrei Alexandrescu, "Speed is Found in Minds of People,"
 CppCon 2019
<https://www.youtube.com/watch?v=FJJTYQYB1JQ>

Empirical Analysis

- **On modern computers (e.g., Intel i5/i7 2.5-3.5GHz)**
 - $O(N) = 1000000 \sim 8000000$ takes about 1s to finish
 - *Generally true* on modern PCs
- **Big-O complexity for an input size N**
 - $O(N)$
 - `for(int i=1; i<=N; i++) some_constant_work();`
 - $O(N^2)$
 - `for(int i=1; i<=N; i++)`
 `for(int j=1; j<=N; j++)`
 `some_constant_time_work();`
 - $O(N^3), O(N^4) \dots$

How long does
these loops take?

Empirical Analysis (cont'd)

- **A Problem with $N = 1000$, Time Limit = 1s**
 - Could an $O(N)$ algorithm pass the time limit constraint?
 - What about the $O(N \log N)$?
 - What about the $O(N^2)$?
 - What about the $O(N^2 \log N)$?
 - What about the $O(1)$ time ?
- **A Problem with $N = 1000000$, Time Limit = 1s**
 - Could an $O(N)$ algorithm pass the time limit constraint?
 - What about the $O(\log N)$?
 - What about the $O(\log N \log N)$?
 - What about the $O(N^2)$?

Applicable to Various Algorithm Designs

- Greedy
- Divide and Conquer
- Dynamic programming
- Graph algorithms
- Analytical algorithms
 - Linear programming
 - Integer linear programming
 - Quadratic programming
 - ...

Exponential Time Complexity

- An algorithm has an **exponential time** complexity if its execution time is given by the formula

$$\text{execution time} = k_1 \cdot (k_2)^n$$

where n is the size of the input data and k_1, k_2 are constants

Ex: $O(2^n)$, $O(n^2 3^n)$, ...

Problem of Exponential Function

- Consider $O(2^n)$, value doubled when n is increased by 1

n	2^n	$1\mu\text{s} \times 2^n$
10	10^3	0.001 s
20	10^6	1 s
30	10^9	16.7 mins
40	10^{12}	11.6 days
50	10^{15}	31.7 years
60	10^{18}	31710 years

- If you borrow \$10 from a credit card with APR 18%, after 40 yrs, you will owe \$12700!

Intractable Problems

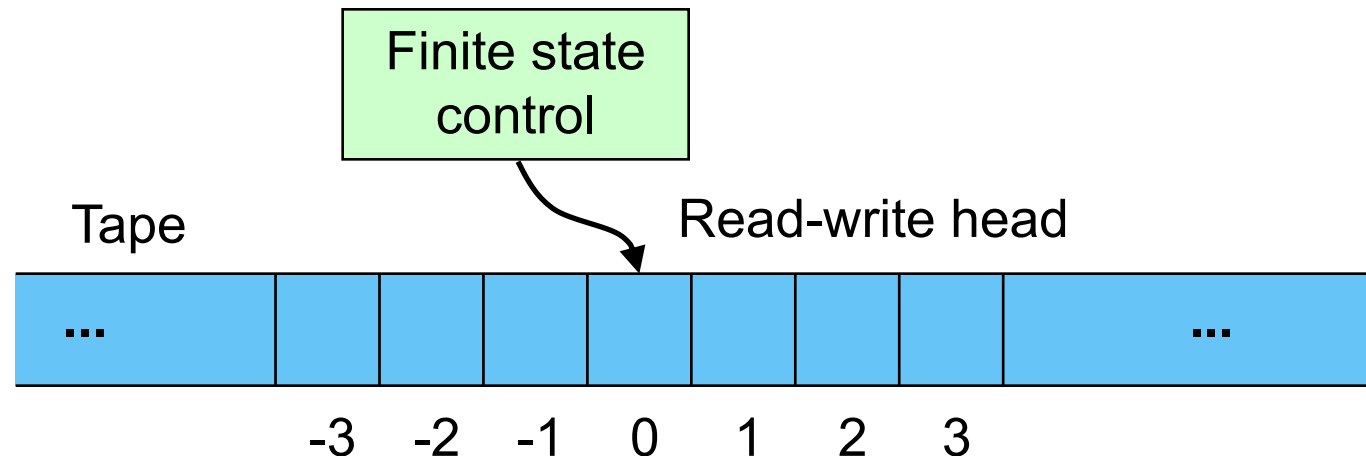
- With exponential complexity, the execution time grows so fast that even the fastest computers cannot solve problems of practical sizes in a reasonable time
- The problem is called **intractable** if the best algorithm known to solve this problem requires exponential time
- Many physical design problems are intractable, unfortunately ☹
 - Partition a circuit of billions of components with various constraints
 - Place millions of cells in a rectangular area with various constraints

Complexity Classes: P Class

- **Class P** contains those problems that can be solved in polynomial time (the number of computation steps necessary can be expressed as a polynomial of the input size n)
 - Ex: $O(n)$, $O(n^2)$, $O(n \log n)$
- The computer defined here is a deterministic Turing machine

Deterministic Turing Machine

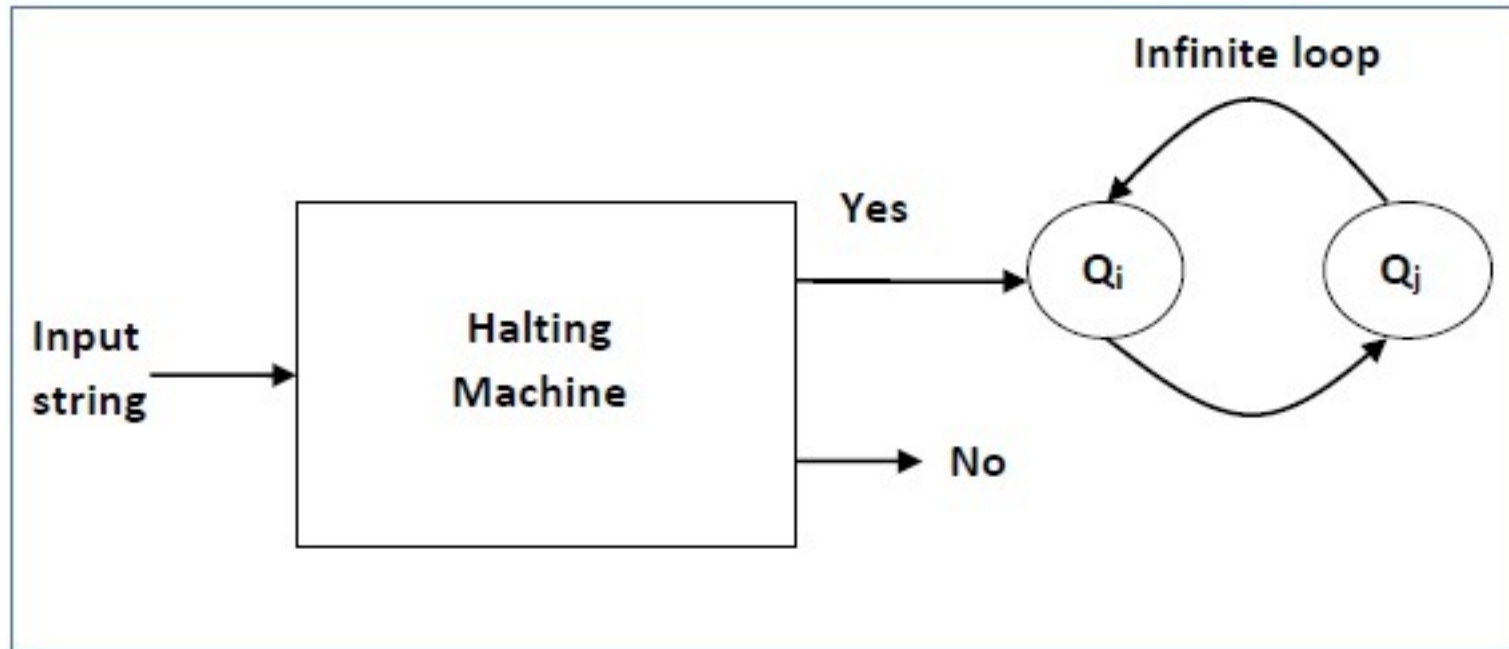
- A mathematical model of a universal computer
- Any computation that needs polynomial time on a Turing machine can also be performed in polynomial time on any other machine
- Deterministic means each step in a computation is predictable



Undecidable Decision Problem

- **The Halting Problem**

- Determine from a description of an arbitrary computer program and an input *whether the program will finish running* (i.e., halt) or continue to run forever

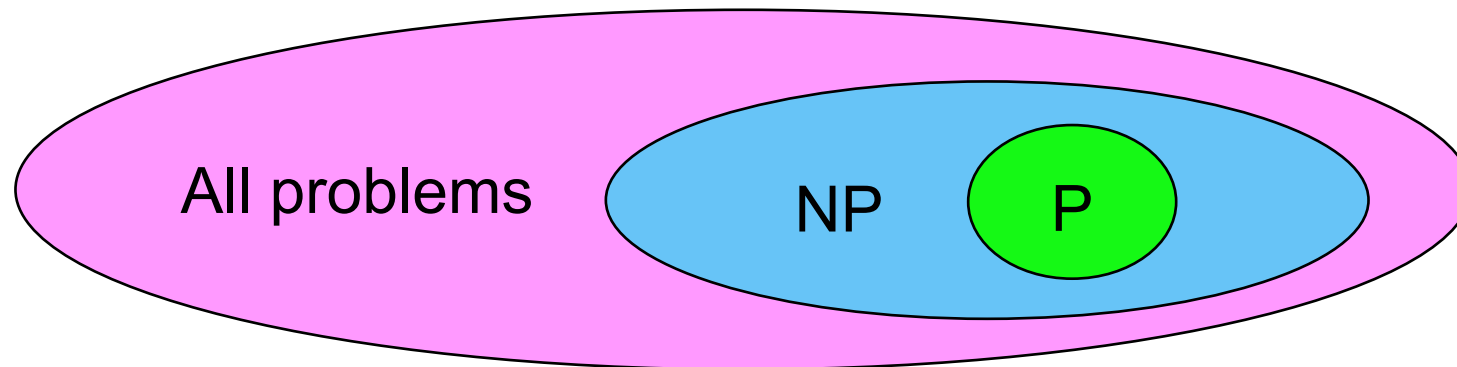


Non-deterministic Turing Machine

- **An imaginary, very powerful machine that**
 - Can spawn many parallel search paths each solving the problem deterministically
 - NOT a formal definition but an easy way to understand what a non-deterministic Turing machine is!
 - If **solution checking** for some problems can be done in polynomial time on a deterministic machine, then the problem can be **solved** in polynomial time on a non-deterministic Turing machine
- **Non-deterministic two stages:**
 - make a guess what the solution is
 - check whether the guess is correct

Complexity Classes: NP Class

- **Class NP** contains those problems that can be solved in polynomial time on a *non-deterministic* Turing machine



- Alternatively, NP is the set of decision problems for which the problem instances, where the answer is "yes", have proofs verifiable in polynomial time by a deterministic Turing machine

[https://en.wikipedia.org/wiki/NP_\(complexity\)](https://en.wikipedia.org/wiki/NP_(complexity))

NP-complete Problem Class

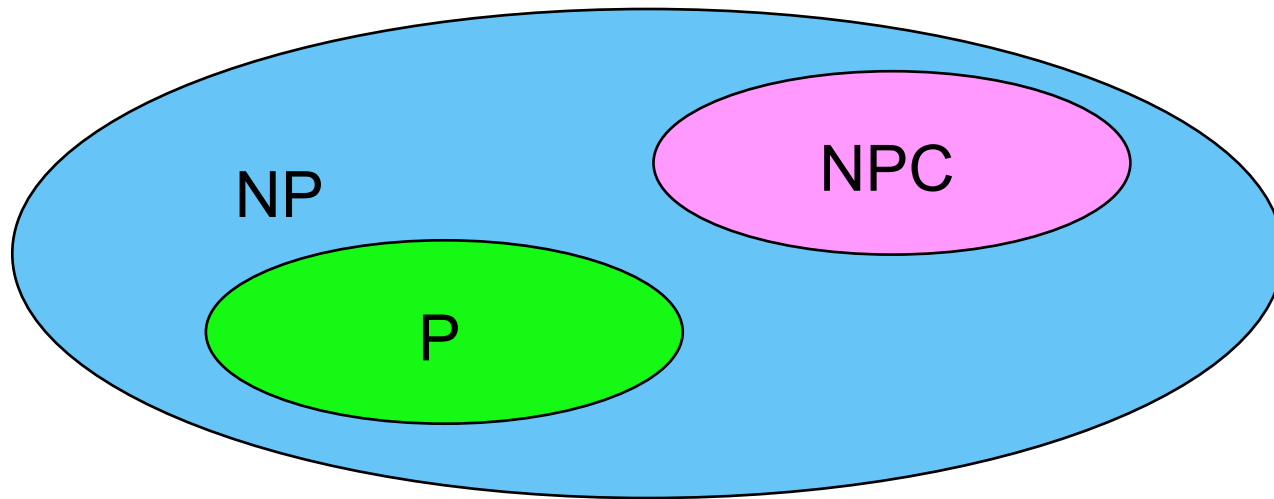
- A question which is still not answered:

$$P = NP \text{ or } P \neq NP$$

- There is a strong belief that $P \neq NP$, due to the existence of NP-complete problems (NPC)
 - All NPC problems have the same degree of difficulty
 - If one of them could be solved in polynomial time, all of them would have a polynomial time solution

NP-complete Problem Definition

- A problem is **NP-complete** if and only if
 - It is in NP
 - Every problem in NP is reducible to the problem
 - This condition alone make the problem NP-hard (discussed later)



There is a strong belief that $P \neq NP$, due to the existence of NPC problems.

What is “Reducible”?

- If we can solve problem A, and if problem B can be transformed into an instance of problem A **with polynomial time**, then we can solve problem B by **reducing** problem B to problem A and then solve the corresponding problem A

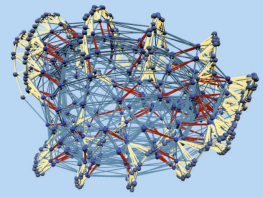
$$B \leq_p A$$

(the symbol “ \leq ” hints that A is at least or more difficult than B)

- Example:
 - Problem A: Sorting
 - Problem B: Given n numbers, find the i-th largest numbers.
 - Polynomial-time Reducible

First NP-complete Problem: Satisfiability

- **Cook's theorem**
 - SATISFIABILITY (SAT) is NPC
 - The first NPC problem
 - Every NP problem reduces to SAT



SAT Competition 2022

Affiliated with the 25th International Conference on Theory and Applications of Satisfiability Testing taking place on the 2nd - 5th of August 2022 in Haifa, Israel.

SAT Competition 2022 is a competitive event for solvers of the Boolean Satisfiability (SAT) problem. It is organized as a satellite event to the SAT Conference 2022 and stands in the tradition of the annual SAT Competitions and SAT-Races / Challenges.

Objective

The area of SAT Solving has seen tremendous progress over the last years. Many problems (e.g. in hardware and software verification) that seemed to be completely out of reach a decade ago can now be handled routinely. Besides new algorithms and better heuristics, refined implementation techniques turned out to be vital for this success.

To keep up the driving force in improving SAT solvers, we want to motivate implementers to present their work to a broader audience and to compare it with that of others.

Researchers from both academia and industry are invited to submit their solvers and benchmarks to SAT Competition 2022.

Overview

Competition Tracks

Solver Submission

- UNSAT Certificates
- StarExec Cluster
- AWS Cloud

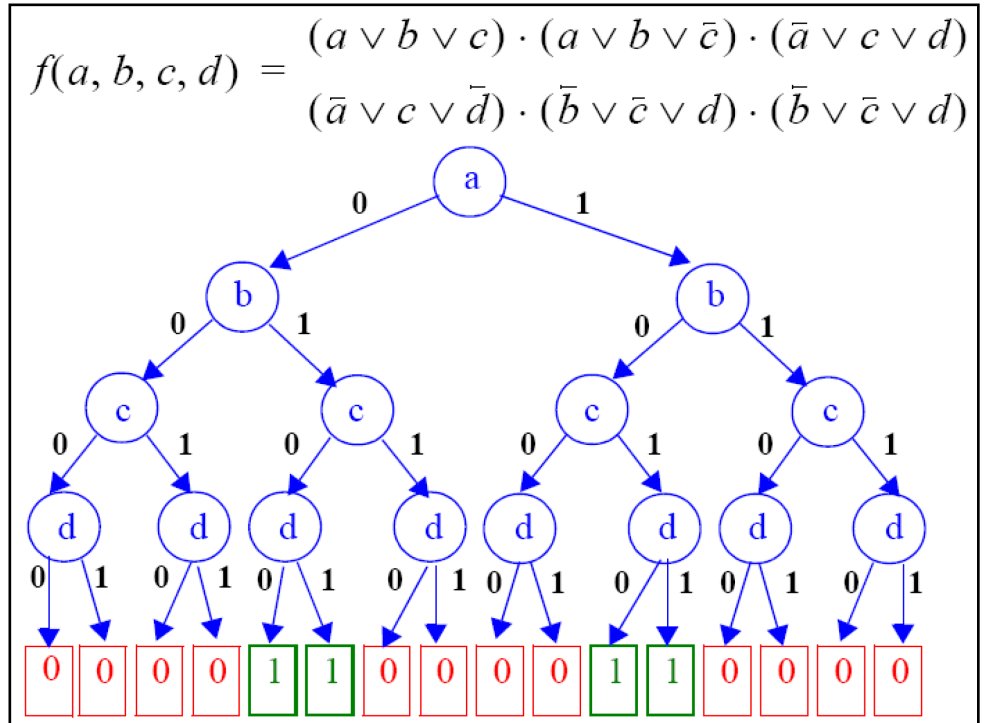
Benchmark Submission

Downloads

Results

Organizers

<https://satcompetition.github.io/2022/>



SAT Problem: Can we find an input for $f(a, b, c, d)$ to be 1?

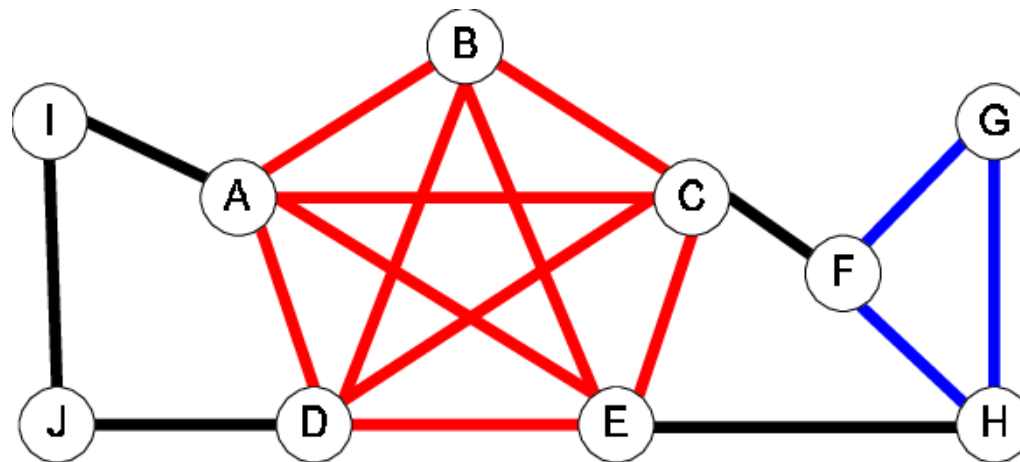
NP-complete Problem Examples

- **Clique finding problem**

- Input: graph $G = (V, E)$, positive integer $K \leq |V|$
- Question: does G contain a clique of size K or more?

- **Minimum cover problem**

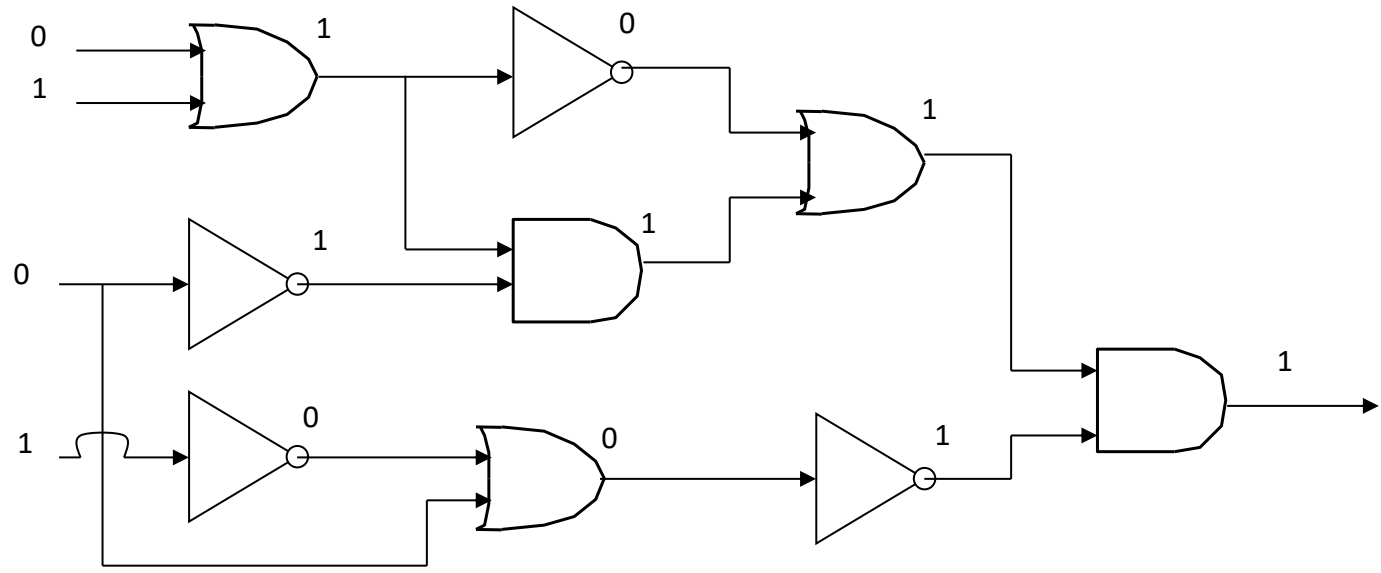
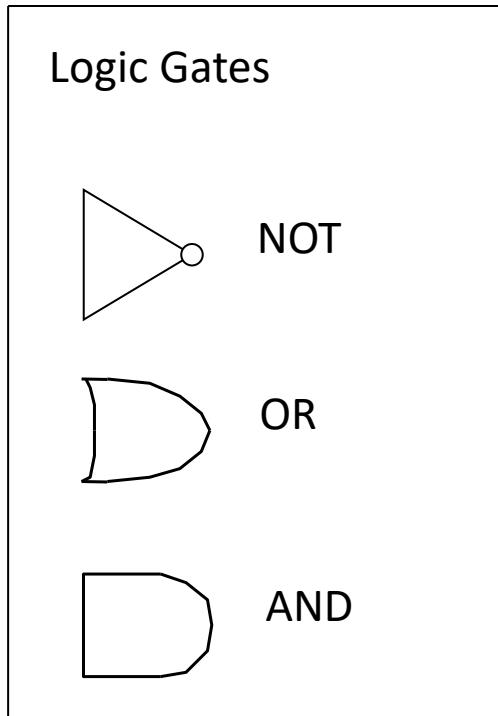
- Input: collection C of subsets of a finite set S , positive integer $K \leq |C|$
- Question: does G contain a cover for S of size K or less?



NP-complete Problem Examples (cont'd)

- **Circuit set problem**

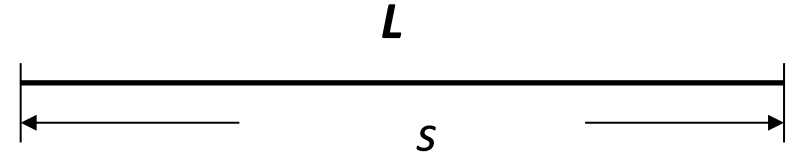
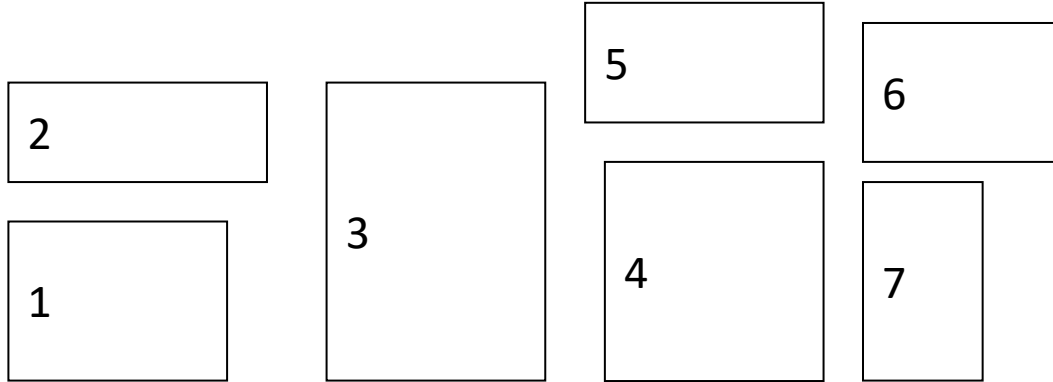
- Take a Boolean circuit with a single output node and ask whether there is an assignment of values to the circuit's inputs so that the output is "1"



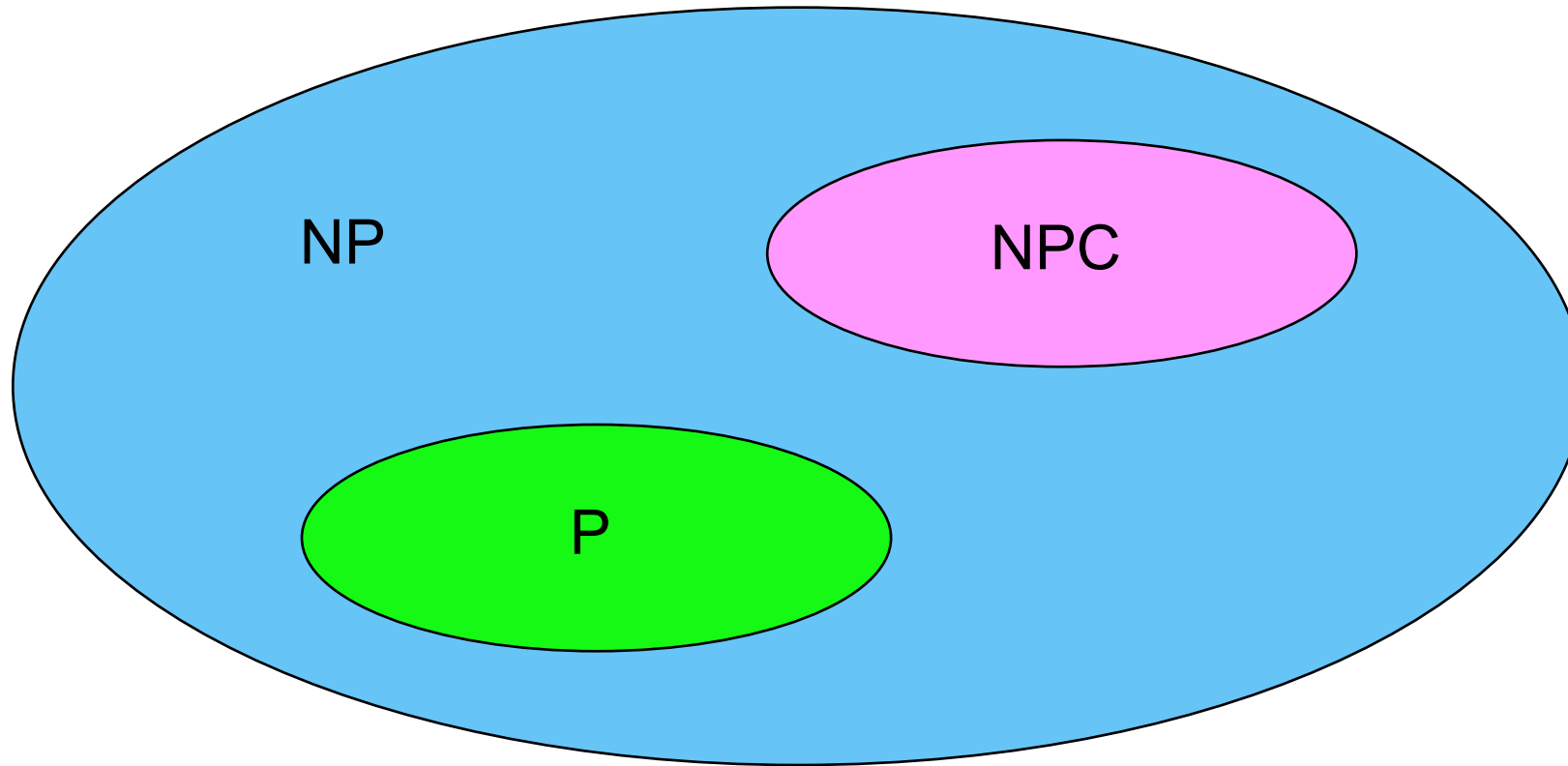
NP-complete Problem Examples (cont'd)

- **Knapsack problem**

- Can we translate a subset of rectangles to have their bottom edges on L so that the total area of the rectangles touching L is at least s ?



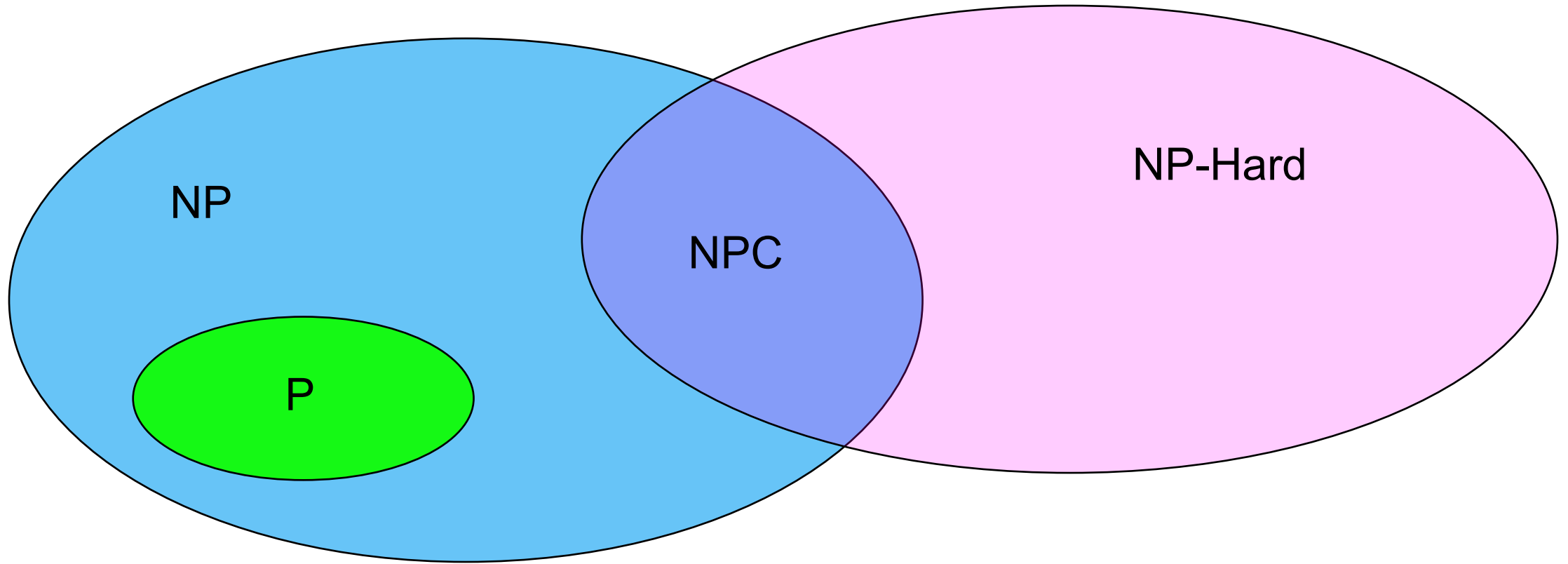
World of NP: Assuming $P \neq NP$



NP-Hard Problems

- Any decision problem (inside or outside of NP) to which we can transform an NP-complete problem to it in polynomial time will have a property that it cannot be solved in polynomial time, unless $P = NP$
- Such problems are called NP-hard
 - “as hard as the NP-complete problems”

NP-Hard, NP, and NPC

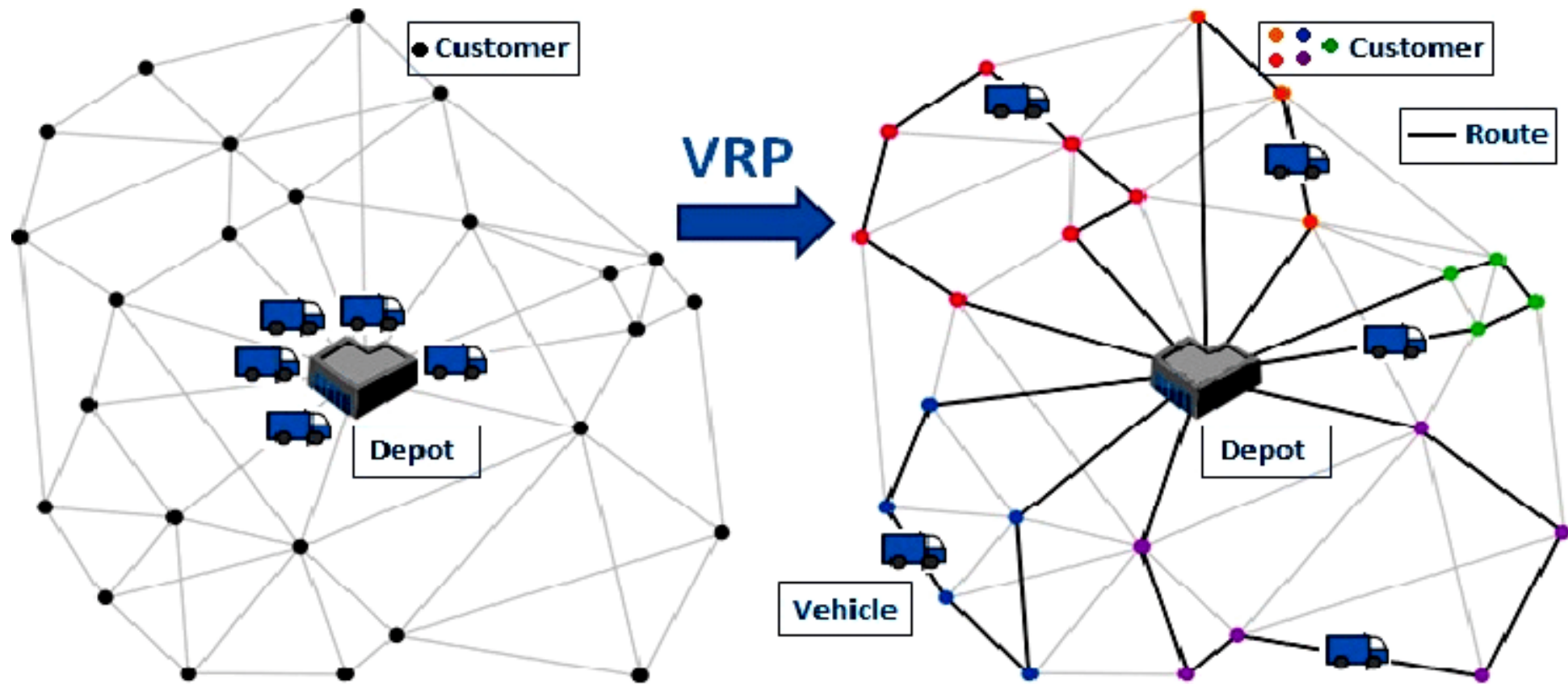


- **Traveling salesman problem (TSP)**


-

Applications of TSP


- Vehicle routing problem (VRP)






Applications of TSP (cont'd)




Center for Discrete Mathematics and Theoretical Computer Science
Founded as a National Science Foundation Science and Technology Center





Click [here](#) for event search




[HOME](#) [ABOUT](#) [NEWS](#) [PROGRAMS](#) [EVENTS](#) [GIVING](#)

HOME > PROGRAMS > IMPLEMENTATION CHALLENGES > IMPLEMENTATION CHALLENGE: VEHICLE ROUTING

Implementation Challenge: Vehicle Routing

12th Implementation Challenge

The 12th Implementation Challenge is dedicated to the study of Vehicle Routing problems (broadly defined), bringing together research in both theory and practice. The over-arching purpose of a Challenge is to assess the practical performance of algorithms for a particular problem class, while fostering interactions that transfer ideas between researchers in areas that span algorithms, data structures, implementation, and applications. This rendition of the Challenge is part of the [DIMACS Special Focus on Bridging Continuous and Discrete Optimization](#) and will be capped by a [workshop](#) hosted by DIMACS at Rutgers University. The event has been postponed because of COVID-19 and is now planned for April 6-8, 2022. This Challenge is being held in honor of David S. Johnson, and the workshop will include activities dedicated to him and his many contributions to the study of algorithms.



DIMACS Implementation Challenge

NEWS

RESU

SUBM

PAPE

PUBLI

TIMEL

PARTI

CAPACITATED VRP

VRP WITH TIME

The VRP variants currently included in the Challenge are:

- 1) [Capacitated VRP](#)
- 2) [Capacitated VRP with Time Windows](#)
- 3) [Inventory Routing Problem](#)
- 4) [VRP with Split Deliveries](#)
- 5) [Electric Vehicle Routing](#)
- 6) [Capacitated Arc Routing](#)
- 7) [Time-dependant Capacitated Arc Routing](#)
- 8) [Dynamic Ride Hailing](#)

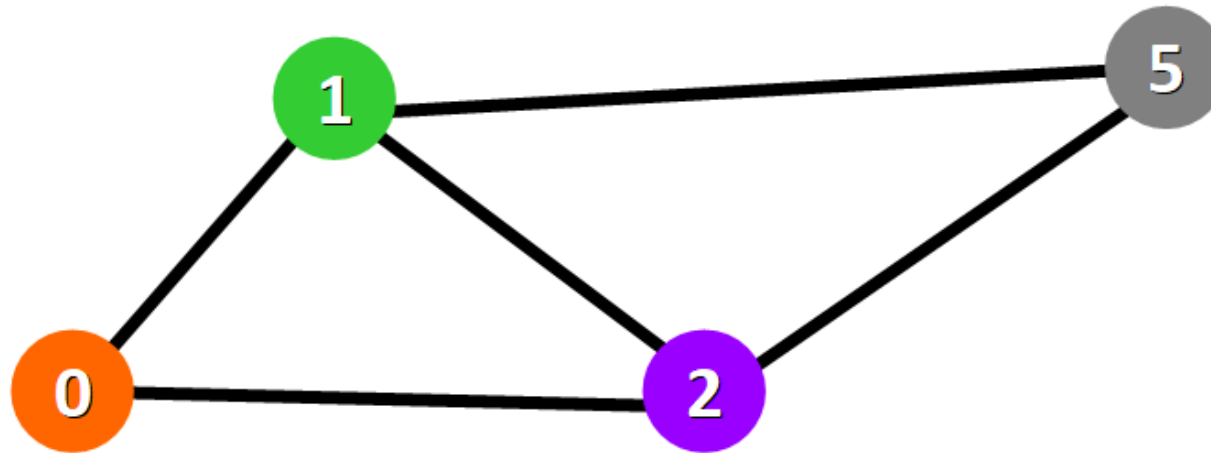
2020-2022 DIMACS Implementation Challenge: <http://dimacs.rutgers.edu/programs/challenge/vrp/>

Practical Consequences

- **Many physical design problems are NPC or NP-hard**
 - Exact solutions to such problems are often of exponential complexity
 - Exact solutions to such problems can only be found when the problem size is small (e.g., $N \leq 20$)
- **One should instead focus on sub-optimal solutions by**
 - **Approximation algorithms**: they can guarantee a solution within e.g. 20% of the optimum
 - **Heuristics**: nothing can be said a priori about the quality of the solution (experience-based)

Example

- Tractable and intractable problems can be very similar:
 - the SHORTEST-PATH problem for undirected graphs is in **P**
 - the LONGEST-PATH problem for undirected graphs is **NP-complete**



Summary

- The class NP-complete is a set of problems which we believe there is no polynomial time algorithms
 - Therefore, it is a class of hard problems
- NP-hard is another class of problems containing the class NP-complete
 - However, we don't know if it belongs to NP
- If we know a problem is in NP-complete or NP-hard, you have little hope to solve it efficiently
 - Just keep this in mind if you still don't understand it ...