Theory of NP Completeness

Computing times of algorithms can be clustered into two groups-

1. Solutions are bounded by the polynomial

2. Solutions are bounded by a non-polynomial

The theory of the NP-Completeness does not provide any method of obtaining polynomial time algorithms for the problems of the second group.

Note 1: "***Many of the problems for which there is no polynomial time algorithm available are computationally related***". Try to find relationship between problems that can't be solved in polynomial time.

Note 2: ***When we are not able to solve a problem in a deterministic manner, let us write the algorithm in a non-deterministic manner.***

Deterministic algorithms

Algorithm with the property that the result of every operation is uniquely defined is called as deterministic algorithm.

Non-Deterministic algorithms

In a theoretical framework, we can allow algorithms to contain operations whose outcome are not uniquely defined but are limited to a specified set of possibilities.

When the outcome is not uniquely defined but is limited to a specific set of possibilities, we call it **non deterministic algorithm**.

Statements used to specify such algorithms are:

1. choice (S) arbitrarily choose one of the elements of set S

2. failure signals an unsuccessful completion

3. success signals a successful completion

The assignment X= choice(1:n) could result in X being assigned any value from the integer range[1..n]. There is no rule specifying how this value is chosen.

The computing time for failure and success is taken to be O(1).

A machine capable of executing a nondeterministic algorithm is known as nondeterministic machine (does not exist in practice).

Example: Un-deterministic Searching Algorithm

Problem Statement: Searching an element **x** in a given set of elements A(1:n). We are required to determine an index **j** such that A(j) = x or j = 0 if x is not present.

Algorithm:

```
j = choice(1:n)
if A(j) = x {
     print(j);
     success
}
print('0');
     failure
```

"A nondeterministic machine does not make any copies of an algorithm every time a choice is to be made. Instead it has the ability to correctly choose an element from the given set".

Decision Problem vs Optimization Problem

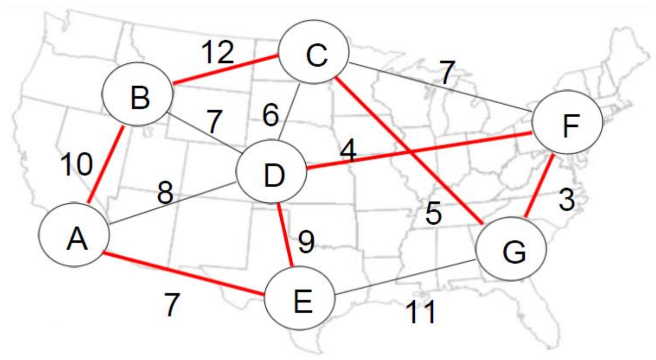
Decision Problem: computational problem with intended output of “yes” or “no”, 1 or 0

Example

Traveling Salesperson Problem:

Given: a weighted graph of nodes representing cities and edges representing flight paths (weights represent cost)

Is there a route that takes the salesperson through every city and back to the starting city with cost no more than k? The salesperson can visit a city only once (except for the start and end of the trip).



Is there a route with cost at most 56?

YES (Route above costs 50.) (10 + 12 +5 + 3 + 4 + 9 + 7=50)

Is there a route with cost at most 40? (YES or NO)?

Optimization Problem:

computational problem where we try to get optimal solution (maximize or minimize)

An algorithm describes an **optimization problem** if the answer is in maximization or minimization.

Example:

If there are n cities, what is the maximum number of routes? That we might need to compute?

How to build a route?

Compute cost of every possible route.

- Pick a starting city
- Pick the next city (n-1 choices remaining)
- Pick the next city (n-2 choices remaining)
- ….

Maximum number of routes: (n-1)!   i.e. O(n!)

Computability Classes

P Class *(Polynomial time solvable)*
P is a set of all decision problems solvable by a deterministic algorithm in polynomial time.

Intuitively, P is the set of problems that can be solved in time $O(n^k)$, for some constant "**k**" where "**n**" is the size of the input to the problem.

NP Class

NP is the set of all decision problems verifiable by a nondeterministic algorithm in polynomial time.

The class **NP** consists of all those decision problems whose positive solutions can be verified in polynomial time given the right information.

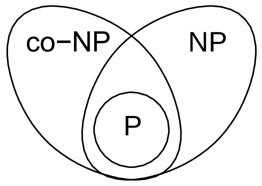
NP is the set of decision problems with the following property:

> If the answer is YES, then there is a proof of this fact that can be checked in polynomial time.

NP does NOT stand for non-polynomial, it stands for "Non-deterministic polynomial time"

co-NP Class

co-NP is the opposite of NP. If the answer to a problem in co-NP is NO, then there is a proof of this fact that can be checked in polynomial time.



NP-Hard

A problem M is NP-hard if every other problem L in NP/NP Hard is polynomial-time **reducible** to M.

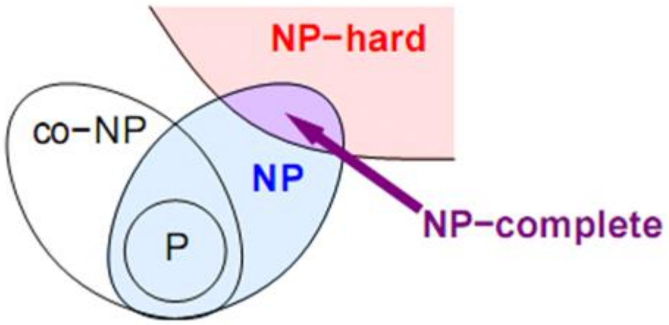
NP-complete (NPC)

A problem is NP-complete if

(1) It is in NP, and

(2) It is NP-hard.

That is NPC=NP ∩ NP-hard



Example: Standard NP-complete problems are

CIRCUIT-SAT

Clique Decision Problem

Node Cover Decision

Chromatic Number Decision Problem

Traveling salesperson decision problem

Property:

$$P \subseteq NP$$

NP-Complete (NPC) - have the property that it can be solved in polynomial time if all other NP-Complete problems can be solved in polynomial time.

Every problem in NPC can be transformed to another problem in NPC.

"All NP-Complete problems are NP-Hard but not all NP-Hard problems are NP-Complete."

NP-Complete problems are subclass of NP-Hard.

If there were some way to solve one of these problems in polynomial time, we should be able to solve all of these problems in polynomial time.

The most famous unsolved problem in Computer Science is

Whether $P=NP$ or $P \neq NP$

http://www.claymath.org/millennium-problems/p-vs-np-problem

Cook's Theorem

Satisfiability problem is in P if $P = NP$

Or

A known NP Complete problem is in P if P=NP.

Satisfiability problem

Let $x_1,x_2,...,x_n$ denotes boolean variables. Let $\overline{x_i}$ denotes the negation of x_i. A literal is either a variable or its negation.

A formula in propositional calculus is an expression that can be constructed using literals and ***and*** or ***or***.

Formula is in ***conjugate normal form*** (CNF) iff it is represented as $\bigwedge_{i=1}^{k} c_i$ where the c_i are clauses each represented as $\bigvee 1_{ij}$

It is in ***disjunctive normal form*** (DNF) iff it is represented as $\bigvee_{i=1}^{k} c_i$ and each clause is represented as $\bigwedge 1_{ij}$

Example:

$(x1 \wedge x2) \vee (x3 \wedge \overline{x4})$ is in DNF

$(x3 \vee \overline{x4}) \wedge (x1 \vee \overline{x2})$ is in CNF.

The satisfiability problem is to determine if a formula is true for some assignment of truth values to the variables.

```
procedure EVAL(E, n)
//determines if the propositional formula E is satisfiable

var boolean: x[1..n];
for i = 1 to n do //choose a truth value assignment//
          $x_i$ = choice(true, false);
if E($x_1$,...,$x_n$) is true then
     success //satisfiable
else
     failure
```

Review of NPC and NP Hard

Definition. Let *L1* and *L2* be problems.

L1 ***reduces*** to *L2* iff there is a way to solve *L2* by deterministic polynomial time algorithm that solve *L1* in polynomial time.

If we have a polynomial time algorithm for *L2* then we can solve *L1* in polynomial time.

Definition. A problem L is ***NP-Hard*** if and only if satisfiability reduces to L. (Satisfiability α L)

Definition. A problem L is ***NP-Complete*** if and only if L is NP-Hard and $L \in NP$.

- **Steps to prove X is NP-complete**
  - Prove X∈ NP.
    - Given a certificate, the certificate can be verified in poly time.
  - Prove X is NP-hard.
    - Select a known NP-complete P'.
    - Describe a transformation function *f* that maps every instance *x* of P' into an instance *f*(*x*) of X.
    - Prove *f* satisfies that the answer to *x*∈P' is YES if and only if the answer to *f*(*x*)∈X is YES for all instance *x* ∈P'.
    - Prove that the algorithm computing f runs in poly-time.

Non-computable Problems

Non-computable Problems: Problems that have no algorithms at all to solve them

Noncomputability and Undecidability

- An algorithmic problem that has no algorithm is called **noncomputable**.
- If the noncomputable algorithm requires a yes/no answer, the problem is called undecidable.

Program Termination/Halting Problem

● Can we determine if a program will terminate given a valid input?

● Example:

Does this program terminates when x = 4001?
Does this program terminates when x = 2008?

```
#include<stdio.h>
int main()
{
   int x=2;
   while(x!=1)
   {
      printf("%d ",x);
      x=x-2000;
   }
   printf("Done");
   return 0;
}
```

Can we write a general program Q that takes as its input any program P and an input I and determines if program P will terminate (halt) when run with input I?

- It will answer YES if P terminates successfully on input I.
- It will answer NO if P never terminates on input I.

This computational problem is undecidable!

- No such general program Q can exist!
- It doesn't matter how powerful the computer is.
- It doesn't matter how much time we devote to the computation.

Halting problem: An example of NP-Hard decision problem which is not NP-Complete.