LOWER BOUND THEORY

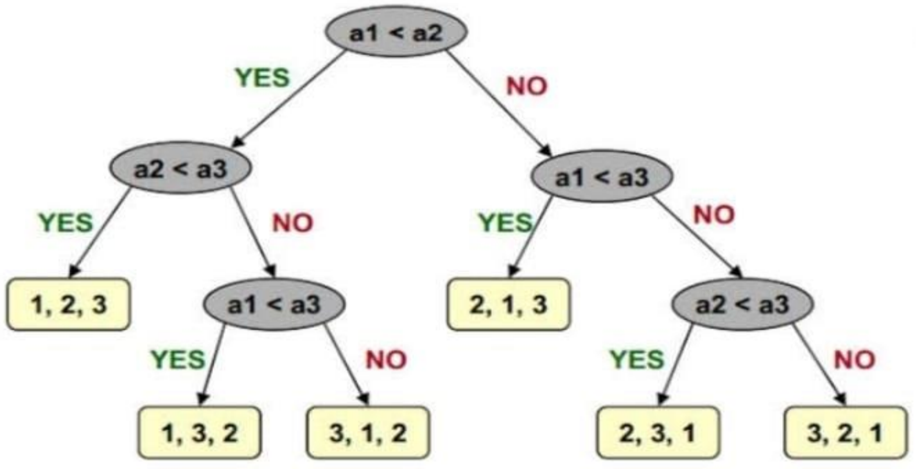
- Lower bound theory concept is based upon the calculation of minimum time that is required to execute an algorithm.
- Lower bound theory is used to calculate a minimum number of comparisons required to execute an algorithm.
- According to lower bound theory, for a lower bound L(n), of an algorithm, it is not possible to have any algorithm whose time complexity is less than L(n) for random input.
- Once Lower bound is calculated, then we compare it with the actual complexity of the algorithm and if their order is same then we can decide our algorithm as optimal

Decision Tree: A decision tree is full binary tree that shows the comparison between elements that are executed by an appropriate sorting algorithm operating on an input of a given size.

- ✓ Control, Data movement and other conditions of the algorithm are ignored.
- ✓ In a decision tree there will be an array of length n, so total leaves will be n!

Example: Decision tree for Comparison of three elements a1, a2, a3.

- ✓ Left subtree will be true condition i.e. $a_i \leq a_j$
- ✓ Right subtree will be false condition i.e. $a_i > a_j$



Decision Tree

APPROXIMATION ALGORITHMS

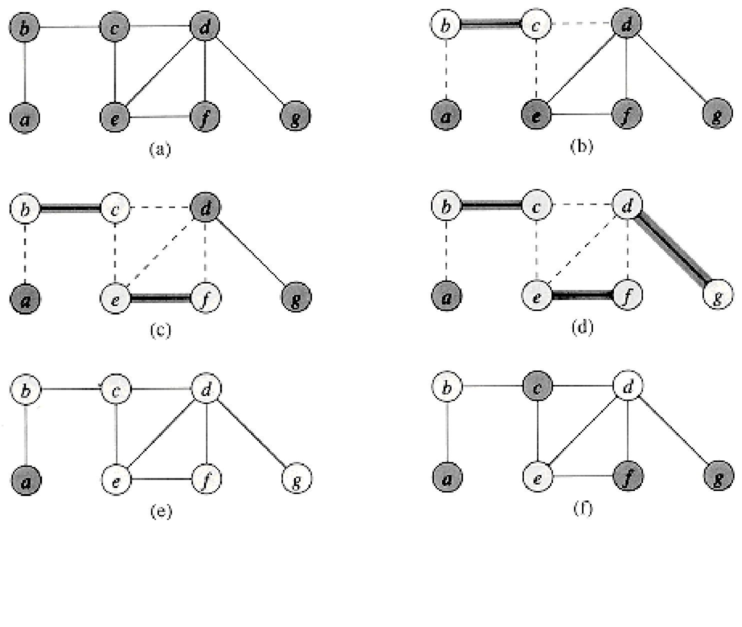
- An algorithm that returns near-optimal solutions is called an ***approximation algorithm***.
- An Approximation algorithm is a way of dealing with NP Completeness for optimization problem.
- Goal of approximation is to come close as much as possible to optimal solution in polynomial time.
- We say that an approximation algorithm for the problem has a ***ratio bound*** of $\rho(n)$ if for any input of size n, the cost C of the solution produced by the approximation algorithm is within a factor of $\rho(n)$ of the cost C^* corresponding to an optimal solution:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le \rho(n) .$$

Vertex-Cover problem

- A ***vertex cover*** of an undirected graph $G = (V,E)$ is a subset $V' \subseteq V$ such that if (u,v) is an edge of G, then either $u \in V'$ or $v \in V'$ (or both). The size of a vertex cover is the number of vertices in it.
- The ***vertex-cover problem*** is to find a vertex cover of minimum size in a given undirected graph. We call such a vertex cover an ***optimal vertex cover***

```
APPROX-VERTEX-COVER($G$)
1  $C \leftarrow \emptyset$
2  $E' \leftarrow E[G]$
3  while $E' \neq \emptyset$
4      do let $(u, v)$ be an arbitrary edge of $E'$
5          $C \leftarrow C \cup \{u, v\}$
6          remove from $E'$ every edge incident on either $u$ or $v$
7  return $C$
```

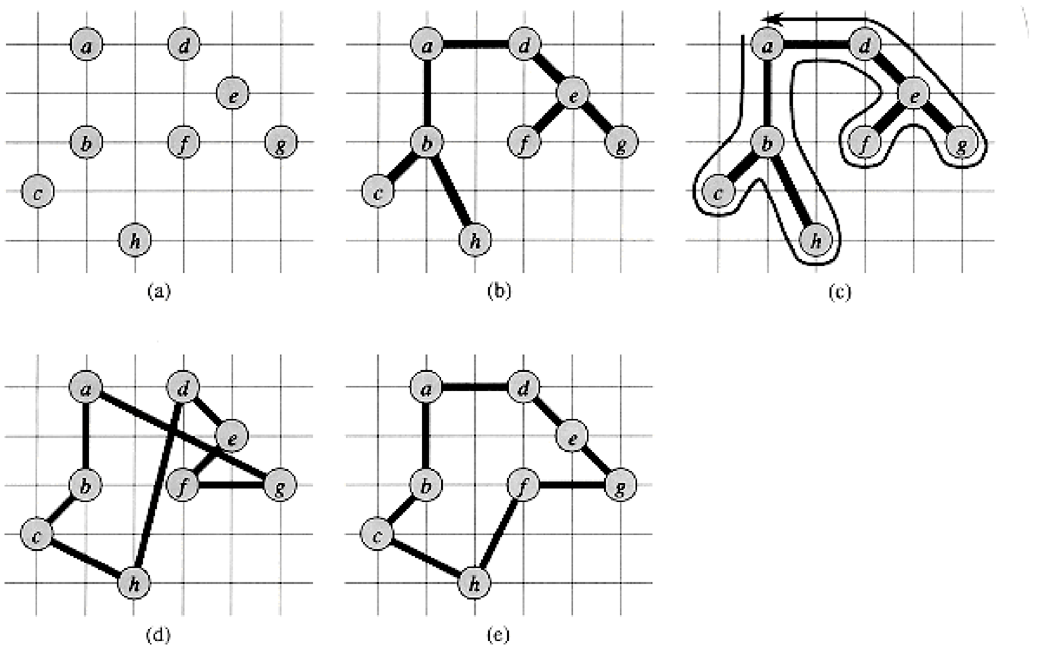


Traveling-Salesman problem

In the traveling-salesman problem, we are given a complete undirected graph $G = (V, E)$ that has a non-negative integer cost $c(u, v)$ associated with each edge $(u, v) \in E$, and we must find a Hamiltonian cycle (a tour) of G with minimum cost.

APPROX-TSP-TOUR(*G*, *c*)

```
1 select a vertex $r \in V[G]$ to be a "root" vertex
2 grow a minimum spanning tree $T$ for $G$ from root $r$ using MST-PRIM($G, c, r$)
3 let $L$ be the list of vertices visited in a preorder tree walk of $T$
4 return the Hamiltonian cycle $H$ that visits the vertices in the order $L$
```



(a) (b) (c)

(d) (e)

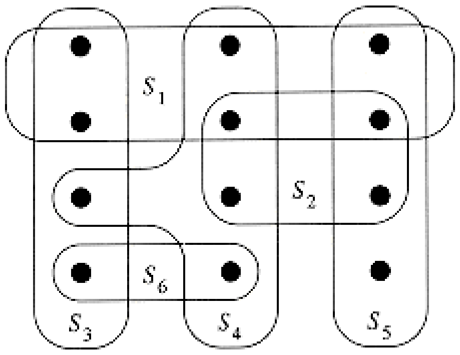
Set-Covering problem

An instance (X, F) of the ***set-covering problem*** consists of a finite set *X* and a family F of subsets of *X*, such that every element of *X* belongs to at least one subset in F:

$$X = \bigcup_{S \in \mathcal{F}} S .$$

We say that a subset S € F ***covers*** its elements. The problem is to find a minimum-size subset $\mathcal{C} \subseteq \mathcal{F}$ whose members cover all of *X*:

```
GREEDY-SET-COVER($X, \mathcal{F}$)
1  $U \leftarrow X$
2  $\mathcal{C} \leftarrow \emptyset$
3  while $U \neq \emptyset$
4        do select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$
5           $U \leftarrow U - S$
6           $\mathcal{C} \leftarrow \mathcal{C} \cup \{S\}$
7  return $\mathcal{C}$
```



Amortized Analysis

- In an amortized analysis, the time required to perform a sequence of data structure operation is averaged over all the operations performed.
- Amortized analysis differs from average case analysis because Amortized analysis does not involve probability.
- An amortized analysis guarantees the average performance of each operation in the worst case.
- Amortized analysis requires knowledge of which series of operations are possible.
- There are generally three methods for performing amortized analysis as follows-
    1. Aggregate Method
    2. Accounting Method
    3. Potential Method

1. Aggregate Method

- ✓ In this method, we determine an upper bound T(n) on the total cost of a sequence of n-operations.
- ✓ In the worst case, the average cost or amortized cost per operation will be T(n)/n

  Example- Let we have a stack allowing following three operations:
    1. Push
    2. Pop
    3. Multipop
- ✓ Let us analyze a sequence of n operations which would be any possible permutation of Push, Pop and Multipop operations, on initially empty stack.
- ✓ Worst case cost of a Multipop operation in the sequence is O(n).
- ✓ The worst-case time of any stack operation is therefore O(n) and hence, a sequence of n operation costs O(n^2) because O(n) Multipop operations will be performed where each Multipop operation costs O(n).
- ✓ Using the aggregate method of amortized analysis, we can obtain a better upper bound that consider the entire sequence of n operations.
- ✓ Any sequence of n operations involving permutation of Push, Pop and Multipop operations has at most worst-case complexity of order O(n) because
    - Each element can be popped at most one time for each time it is pushed
    - Number of Push operations can be O(n) at most.
    - The number of Pops (due to either Pop or Multipop operation) is at most O(n).
    - It means over all complexity of sequence of n operations involving permutation of Push, Pop and Multipop operations will be O(n).
    - The amortized cost = O(n)/n = O(1).

2. <u>Accounting Method</u>

- In this method different charges are assigned to different operation.
- Overcharges result in a credit.
- Credits can be used later to help perform other operations whose amortized cost is less than actual.
- The total amortized cost for any sequence of operations must be an upper bound on the total actual cost (sum of all individual operation's actual cost)
- The total credit assignment must be nonnegative.

Example: Let us analyze sequence of n operations involving permutation of Push, Pop and Multipop operations on an initially empty stack.

Suppose amortized cost for Push, Pop and Multipop operation is 2,0,0 respectively.

Operation	Actual Cost	Amortized Cost
Push	1	2
Pop	1	0
Multipop	Min(s,k)	0

Total amortized cost and credits can be calculated by following way.

Operation	Amortized Cost	Actual Cost	Credit
Push(s,a)	2	1	1
Push(s,b)	2	1	2
Pop(s)	0	1	1
Push(s,c)	2	1	2
Pop(s)	0	1	1
Pop(s)	0	1	0
Push(s,d)	2	1	1
Pop(s)	0	1	0
Total	**8**	**8**	-

For above example total amortized cost will be 8 (for 4 Push operations, i.e. 8 = 2*4). Therefore, total Amortized cost for sequence of n operations involving permutation of Push, Pop and Multipop operations in worst case will be 2*n = O(n) (if n Push operations are applied).

3. Potential Method

- It is similar to accounting method but here, prepaid work is considered as potential or potential energy instead of credit
- The amortized cost $\mathbf{c'_i}$ of the ith operation with respect to potential function Φ is defined by:

  $\mathbf{c'_i = c_i + \Phi(D_i) - \Phi(D_{i-1})}$ (actual cost + potential energy)

  where $\mathbf{c'_i}$ is amortized cost of ith instruction

  $\mathbf{c_i}$ actual cost of ith instruction

  $\mathbf{\Phi(D_i)}$ potential of D_i

- The amortized cost of each operation is therefore its actual cost plus increase in potential due to the operation.
- The total amortized cost of operation is: $\mathbf{\sum_{i=1}^{n} c'_i}$

$$\mathbf{= \sum_{i=1}^{n} [c_i + \Phi(D_i) - \Phi(D_{i-1})]}$$

$$\mathbf{= \sum_{i=1}^{n} [c_i + \Phi(D_n) - \Phi(D_0)]}$$

Example: Potential for a stack represents the number of elements in the stack.

Amortized Cost of Push

Potential Change of Push= $\Phi(D_i) - \Phi(D_{i-1}) = (s+1)-s=1$

Amortized Cost of Push= $c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1+1=2$

Amortized Cost of Multipop(s,k)

Potential Change of Multipop(s,k)= $\Phi(D_i) - \Phi(D_{i-1}) = (s-k)-s=-k$

Amortized Cost of Multipop(s,k)= $c_i + \Phi(D_i) - \Phi(D_{i-1}) = -k+k=0$

Similarly, Amortized Cost of Pop=0

P, NP, NPC and NP Hard

P

- *P is a complexity class that represents the set of all decision problems that can be solved in polynomial time*.
- That is, given an instance of the problem, the answer yes or no can be decided in polynomial time.

Example

Given a connected graph G, can its vertices be coloured using two colours so that no edge is monochromatic?

Algorithm: start with an arbitrary vertex, color it red and all of its neighbour's blue and continue. Stop when you run out of vertices or you are forced to make an edge have both of its endpoints be the same color.

NP

- *NP is a complexity class that represents the set of all decision problems for which the instances where the answer is "yes" have proofs that can be verified in polynomial time.*
- This means that if someone gives us an instance of the problem and a certificate (sometimes called a witness) to the answer being yes, we can check that it is correct in polynomial time.

Example

Integer factorisation is in NP. This is the problem that given integers n and m, is there an integer f with 1 < f < m, such that f divides n (f is a small factor of n)?

This is a decision problem because the answers are yes or no. If someone hands us an instance of the problem (so they hand us integers n and m) and an integer f with 1 < f < m, and claim that f is a factor of n (the certificate), we can check the answer in *polynomial time* by performing the division n / f.

NP-Complete

NP-Complete is a complexity class which represents the set of all problems X in NP for which it is possible to reduce any other NP problem Y to X in polynomial time.

Intuitively this means that we can solve Y quickly if we know how to solve X quickly. Precisely, Y is reducible to X, if there is a polynomial time algorithm f to transform instances y of Y to instances x = f(y) of X in polynomial time, with the property that the answer to y is yes, if and only if the answer to f(y) is yes.

Example

3-SAT. This is the problem where we are given a conjunction (ANDs) of 3-clause disjunctions (ORs), statements of the form

(x_v11 OR x_v21 OR x_v31) AND (x_v12 OR x_v22 OR x_v32) AND ...AND(x_v1n OR x_v2n OR x_v3n) where each x_vij is a boolean variable or the negation of a variable from a finite predefined list (x_1, x_2, ... x_n).

It can be shown that *every NP problem can be reduced to 3-SAT*.

What makes NP-complete problems important is that if a deterministic polynomial time algorithm can be found to solve one of them, every NP problem is solvable in polynomial time (one problem to rule them all).

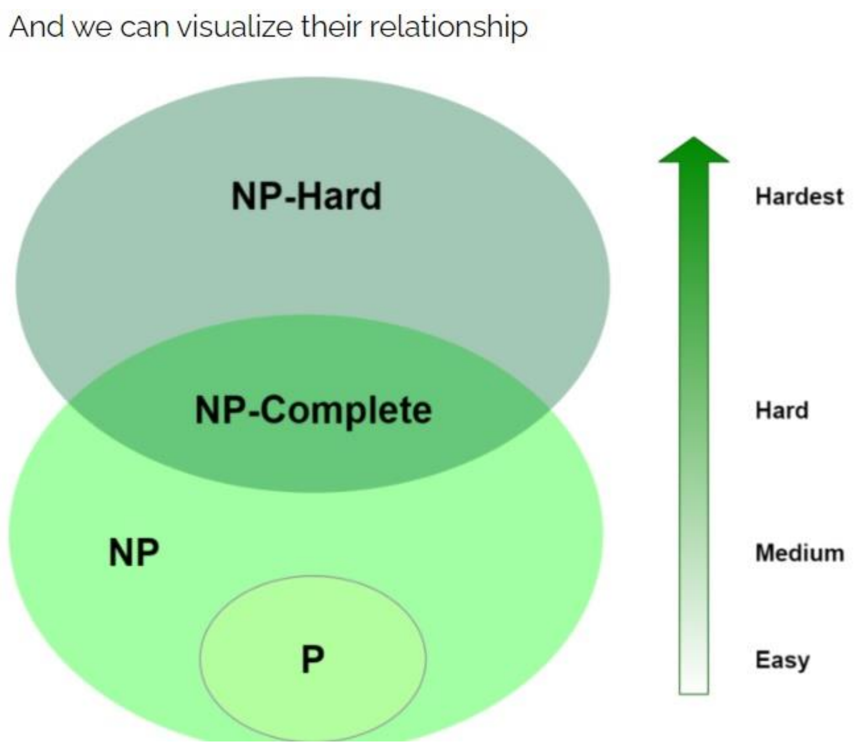
NP-hard

A problem X is NP-hard, if there is an NP-complete problem Y, such that Y is reducible to X in polynomial time.

But since any NP-complete problem can be reduced to any other NP-complete problem in polynomial time, all NP-complete problems can be reduced to any NP-hard problem in polynomial time. Then, if there is a solution to one NP-hard problem in polynomial time, there is a solution to all NP problems in polynomial time.

Example

The **halting problem** is an NP-hard problem. This is the problem that given a program P and input I, will it halt? This is a decision problem but it is not in NP. It is clear that any NP-complete problem can be reduced to this one. As another example, any NP-complete problem is NP-hard.


And we can visualize their relationship
NP-Hard
NP-Complete
NP
P
Hardest
Hard
Medium
Easy