Unit 8:

NP completeness

8.1. Introduction: Tractable and Intractable Problems, Concept of Polynomial Time and Super Polynomial Time Complexity

8.2. Complexity Classes: P, NP, NP-Hard and NP-Complete

8.3. NP Complete Problems, NP Completeness and Reducibility, Cooks Theorem, Proofs of NP Completeness (CNF-SAT, Vertex Cover and Subset Sum)

8.4. Approximation Algorithms: Concept, Vertex Cover Problem, Subset Sum Problem

Prepared by: Shiv Raj Pant

Introduction:

- So far, we studied quite a number of problems and their solutions(algorithms).
- During the course of the study, we had a chance to learn major algorithm design techniques.
- So Now the question is : *Is that all? What next* ? Do we need to know something more about solving problems ?
- The answer would be : "yes". There is still one important thing that we must know about designing algorithms to solve problems – ***The NP completeness***.

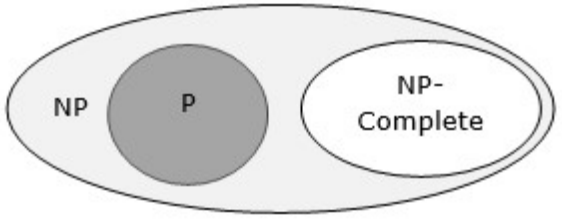
- The main question we deal here are:
  - *Are all problems in the world solvable ? Are there any problems that can not be solved by existing computing systems? By "solvable problem" we mean "the problem for which we can design an algorithm"*
  - *Can all problems be solved efficiently (in polynomial time)? Are there any problems which are not solvable in polynomial time.*
  - *How to solve hard(complex) problems efficiently?*

Tractable and Intractable Problems

- We call problems as tractable or easy, if the problem can be solved using polynomial time algorithms.
- The problems that cannot be solved in polynomial time but requires superpolynomial time algorithm are called intractable or hard problems.
- There are many problems for which no algorithm with running time better than exponential time is known some of them are, traveling salesman problem, circuit satisfiability problem, etc.

P and NP classes and NP completeness

- The set of problems that can be solved using polynomial time algorithm is regarded as class P.
- The problems that are verifiable in polynomial time constitute the class NP.
- The class of NP complete problems consists of those problems that are NP as well as they are as hard as any problem in NP
- The main concern of studying NP completeness is to understand how hard the problem is.
- So if we can find some problem as NP complete then we try to solve the problem using methods like approximation, rather than searching for the faster algorithm for solving the problem exactly.



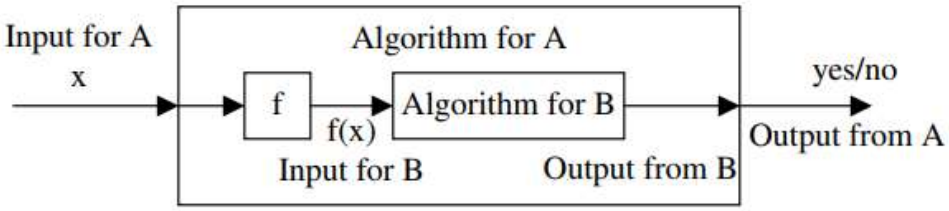
- Class NP is the set of decision problems solvable by nondeterministic algorithms in polynomial time.
- When we have a problem, it is generally much easier to verify that a given value is solution to the problem rather than calculating the solution of the problem.
- Using the above idea we say the problem is in class NP (nondeterministic polynomial time) if there is an algorithm for the problem that verifies the problem in polynomial time.

  For e.g. Circuit satisfiability problem (SAT) is the question "Given a Boolean combinational circuit, is it satisfiable?

  i.e. does the circuit has assignment sequence of truth values that produces the output of the circuit as 1?"
- Given the circuit SAT problem we can take a circuit x and with the set of input values that produce output 1, and we can verify that whether the given input satisfies the circuit in polynomial time.
- So we can say that circuit satisfiability problem is NP.
- We can always say $P \subseteq NP$

Polynomial time reduction

- Given two decision problems A and B, a polynomial time reduction from A to B is a polynomial time function f that transforms the instances of A into instances of B such that the output of algorithm for the problem A on input instance x must be same as the output of the algorithm for the problem B on input instance f(x) as shown in the figure below.



- If there is polynomial time computable function f such that it is possible to reduce A to B, then it is denoted as $A \leq_p B$.
- The function f described above is called reduction function and the algorithm for computing f is called reduction algorithm.

NP-Completeness

We define some problem say A, is NP-complete if

1. $A \in NP$, and
2. $B \leq_p A$, for every $B \in NP$. (this property is called A is NP-hard)

Theorem

If any NP-complete problem is polynomial time solvable, then P = NP. Equivalently, if any problem in NP is not polynomial time solvable, then no NP-complete problem is polynomial time solvable.

Steps for proving a problem A to be NP-complete

1. Prove A ∈ NP.
2. Select a known NP-complete problem B.
3. Describe an algorithm that computes a function f mapping every instance x of B to an instance f(x) of A.
4. Prove that the function satisfies x ∈ B iff f(x) ∈ A for all x.
5. Prove the algorithm for computing f to be polynomial time algorithm.

Why study NP-completeness ?

- Before designing algorithm for any problem, it very important to check if the algorithm is NP-complete.
- If we find any algorithm is NP complete then we should not try to find polynomial algorithm using existing algorithm design techniques.
- In such situation, we should try to solve by approximation algorithm.

Which is the first NP complete problem?

- From the definition of NP completeness, we see that to prove a problem A to be NP-complete, we need another NP problem to be reduced into A.
- This gives rise to a question similar to *"Which came first, egg or chicken?"*
- To solve this egg-chicken problem, we need the first NP complete problem so that later we can prove Np-completeness of other problems
- The first NP-complete problem is the Circuit satisfiability problem (C-SAT) or simply SAT.

Circuit satisfiability problem(C-SAT)

"Given a Boolean combinational circuit, is it satisfiable?

i.e. does the circuit has assignment sequence of truth values that produces the output of the circuit as 1?"

Cook's Theorem

C-SAT is NP-complete

Tentative idea for the proof.

- The following is the basic idea behind the proof of Cook's theorem. The actual proof is very lengthy involving complex explanations.
- Consider a SAT problem S. To prove above theorem, we need to show two things:
    1. *SAT ∈ NP*
    2. *SAT is NP-hard. i.e. for all X ∈ NP, X ≤p S*

1. SAT ∈ NP

Given the circuit SAT problem we can take a circuit x and with the set of input values that produce output 1, and we can verify that whether the given input satisfies the circuit in polynomial time. So we can say that circuit satisfiability problem is NP.

2. SAT is NP-hard

Take a problem V ∈ NP, let A be the algorithm that verifies V in polynomial time (this must be true since V ∈ NP). We can program A on a computer and therefore there exists a (huge) logical circuit whose input wires correspond to bits of the inputs x and y of A and which outputs 1 precisely when A(x,y) returns yes.

For any instance x of V let Ax be the circuit obtained from A by setting the x-input wire values according to the specific string x. The construction of Ax from x is our reduction function. If x is a yes instance of V, then the values y for x gives satisfying assignments for Ax. Conversely, if Ax outputs 1 for some assignments to its input wires, that assignment translates into values for x.

Thus any NP problem can be reduced to a logical circuit. Hence S is NP-hard

Some NP-complete problems

- *Travelling Salesperson problem*
- *Vertex cover problem*
- *Subset sum problem*

Note: The syllabus includes the NP-completeness proofs of CNF-SAT, subset-sum and vertex-cover problems. However, due to limited time, we are skipping this part for now.

Approximation Algorithms

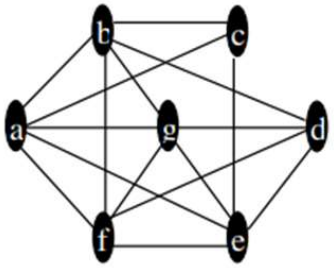
- An approximate algorithm is a way of dealing with NP-completeness for optimization problem.
- This technique does not guarantee the best solution.
- The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at most polynomial time.
- If we are dealing with optimization problem (maximization or minimization) with feasible solution having positive cost then it is worthy to look at approximate algorithm for near optimal solution.
- Approximation algorithms are a useful way to solve NP-complete optimization problems efficiently at the cost of compromising some optimality.

Vertex Cover Problem

A **vertex cover** of an undirected graph G =(V,E) is a subset V' $\subseteq$ V such that for all edges (u,v) $\in$E either u$\in$V' or v$\in$V' or u and v $\in$ V'. The problem here is to find the vertex cover of minimum size in a given graph G.

In simple words, the VCP is to find minimum subset of vertices such that they touch every edges.

Example: Consider following graph.



The possible vertex covers are: {a,b,e,f,g}, {a,b,c,d,e,f,g}, {b,c,d,e,f,g} etc.

The optimal vertex cover is the one with minimum number of vertices among all vertex covers.
e.g. {a,b,e,f,g} is optimal vertex cover.

How to solve?

- Finding optimal vertex-cover is the NP-complete problem.
- But we can efficiently find a vertexcover that is near optimal by using approximation algorithm

Algorithm:

ApproxVertexCover (G){

$C \leftarrow \{\}$; $E' = E$

while E` is not empty

do Let (u, v) be an arbitrary edge of E`

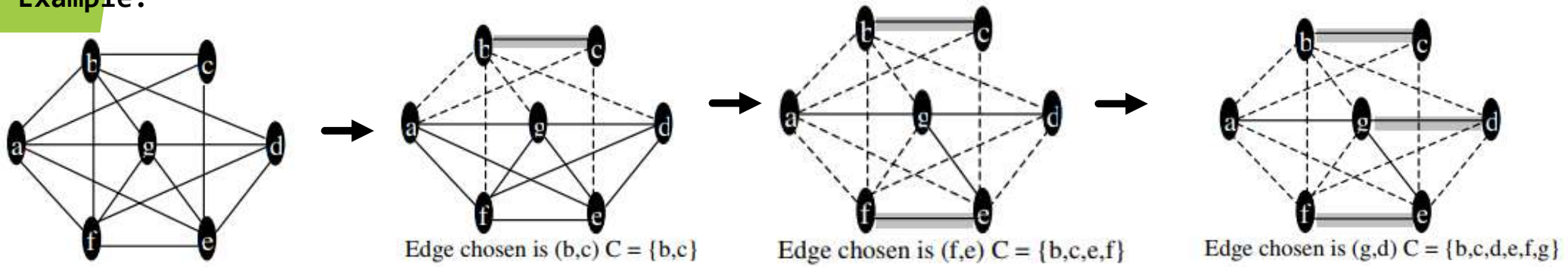
$C = C \cup \{u, v\}$

Remove from E` every edge incident on either u or v

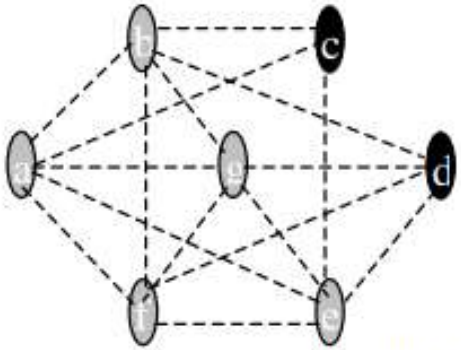
return C

}

Example:



Is this solution optimal ?



Optimal vertex cover as lightly shaded vertices

Analysis:

If E' is represented using the adjacency lists, the above algorithm takes O(V+E) since each edge is processed only once and every vertex is processed only once throughout the whole operation.

Subset sum problem

- Here, we will solve the optimization version of subset sum problem.

"Given a list A= {x_1, x_2, x_3, .. x_n} of numbers and a number m, find the subset of A whose elements sum to the largest number less than or equal to m."

Subset sum problem arise in many real-life applications. For example:
Suppose we have truck that can carry a maximum weight W. The truck has to transport some items from among n items where each item i has weight w_i. We must select items in such a way that the truck is not overloaded i.e., the sum of weights of items must not exceed the capacity of truck(W). Also, we don't want the truck to be underloaded.
The truck should be loaded maximally within it's capacity

- Subset sum problem is NP-complete.
- It means the running time of any algorithm to find exact(optimal) solution takes exponential time.

Example:
A = {1,5,7,9}, m = 11

Here, the largest value (less than 11) that can obtained from subsets of A is 10 (Here, the corresponding subset is {1,9})

- Let us first try the exact solution. Then we will solve the problem using approximation algorithm that finds near-optimal solution efficiently.
- Let us define a set addition operation as follows:

  if A = $\{x_1, x_2, x_3, .. x_k\}$ *and x is a number, then* A+ x = $\{x_1+x, x_2+x, .. x_k+x\}$

An exponential-time algorithm for exact solution

```
ExactSubsetSum(A,m)
{
 L0 = {0}
 for i=1 to n
  {
   Li = merge(Li-1, Li-1 + xi)
   Remove every element greater than m from Li
  }
 Return the largest element in Ln
}
```

This is the same merge operation as in the merge sort

Example:

A = {1,5,7,9}, m = 11

Solution:

L_0 = {0}

L_1 = merge({0},{0}+1} = {0,1}

L_2 = merge({0,1},{0,1}+5} = {0,1,6}

L_3 = merge({0,1,5,6},{0,1,5,6}+7} = {0,1,5,6,7,8,12,13}

Remove 12,13 because 12,13>m

L_4 = merge({0,1,5,6,7,8},{0,1,5,6,7,8}+9} = {0,1,5,6,7,8,9,10,14,15,16,17}

Remove 14,15,16,17. Now the final list L_4 is:

L_4 = {0,1,5,6,7,8,9,10}

The largest element 10 is the answer. It means we can get a maximum value of 10 (less than 11) from given set A.

Analysis:
Clearly, the set L_i grows exponentially while merging the subsets of A.

Approximation algorithm for Subset sum problem

- Approximation algorithm is a modification to the optimal algorithm.
- The idea behind approximation algorithm is to reduce the size of L_i to prevent it from growing exponentially.
- After each merge process, we remove the redundant (almost equal) elements based on some criteria of closeness (a condition which determines if two elements are very close).

```
ApproxSubsetSum(A,m)
{
 L0 = {0}
 for i=1 to n
  {
   Li = merge(Li-1, Li-1 + xi)
   for each group of close values in Li keep
   only one value
   Remove every element greater than m from Li
   }
  Return the largest element in Ln
}
```


Example: A = {104,102,201,101}, m = 308

Solution:

L_0 = {0}
L_1 = merge({0},{0}+104} = {0,104}
L_2 = merge({0,104},{0,104}+102} = {0,102,104,206}

Here 102 and 104 are close values. Keep 102.

L_3 = merge({0,102,206},{0,102,206}+201} = {0,102,201,206,303,407}

201 and 206 are close. Keep 201. Remove 407 because 407>m

L_4 = merge({0,102,201,303},{0,102,201,303}+101} = {0,101,102,201,203,302,303,404}

Here, {101,102}, {201,203},and {302,303} are close groups. Keep first elements only. Remove 404.

Now the final list is: L_4 = {0,101,201,302}

The largest element 302 is the answer.

This is very near to the optimal value 307!

End of unit 8

Thank You!