NAME : ROFAIDA EZZAT MANSOUR

ID:20221453055

SUBJECT : NP COMPLETENESS ALGORITHM DESIGN

In computer science, the NP-completeness or NP-hardness of a problem is a measure of the difficulty of solving that problem. A problem is NP-complete if it can be solved by a polynomial time algorithm, and if it is also NP-hard. The term NP-complete was first introduced by Stephen Cook in 1971.

NP-complete problems are of interest to researchers in computational complexity theory because they provide a way to show that certain problems are intrinsically difficult to solve. Many important problems, such as the traveling salesman problem, are NP-complete.

- In computational complexity theory, a problem is **NP-complete** when:

1. It is a decision problem, meaning that for any input to the problem, the output is either "yes" or "no".
2. When the answer is "yes", this can be demonstrated through the existence of a short (polynomial length) *solution*.
3. The correctness of each solution can be verified quickly (namely, in polynomial time) and a brute-force search algorithm can find a solution by trying all possible solutions.
4. The problem can be used to simulate every other problem for which we can verify quickly that a solution is correct. In this sense, NP-complete problems are the hardest of the problems to which solutions can be verified quickly. If we could find solutions of some NP-complete problem quickly, we could quickly find the solutions of every other problem to which a given solution can be easily verified.

Decision vs Optimization Problems

NP-completeness applies to the realm of decision problems. It was set up this way because it's easier to compare the difficulty of decision problems than that of optimization problems. In reality, though, being able to solve a decision problem in polynomial time will often permit us to solve the corresponding optimization problem in polynomial time (using a polynomial number of calls to the decision problem). So, discussing the difficulty of decision problems is often really equivalent to discussing the difficulty of optimization problems.

For example, consider the vertex cover problem (Given a graph, find out the minimum sized vertex set that covers all edges). It is an optimization problem. The corresponding decision problem is, given undirected graph G and k, is there a vertex cover of size k?

Why should we care?

These NP-complete problems really come up all the time. Knowing they're hard lets you stop beating your head against a wall trying to solve them, and do something better:

- Use a heuristic. If you can't quickly solve the problem with a good worst case time, maybe you can come up with a method for solving a reasonable fraction of the common cases.
- Solve the problem approximately instead of exactly. A lot of the time it is possible to come up with a provably fast algorithm, that doesn't solve the problem exactly but comes up with a solution you can prove is close to right.
- Use an exponential time solution anyway. If you really have to solve the problem exactly, you can settle down to writing an exponential time algorithm and stop worrying about finding a better solution.
- Choose a better abstraction. The NP-complete abstract problem you're trying to solve presumably comes from ignoring some of the seemingly unimportant details of a more complicated real world problem. Perhaps some of those details shouldn't have been ignored, and make the difference between what you can and can't solve.

NP-Complete problems and their proof for NP-Completeness:

- Boolean satisfiability problem (SAT)
- Knapsack problem
- Hamiltonian path problem
- Travelling salesman problem (decision version)
- Subgraph isomorphism problem
- Subset sum problem
- Clique problem
- Vertex cover problem
- Independent set problem
- Dominating set problem
- Graph coloring problem

❖ **SAT Problem:** SAT(Boolean Satisfiability Problem) is the problem of determining if there exists an interpretation that satisfies a given boolean formula. It asks whether the variables of a given boolean formula can be consistently replaced by the values **TRUE or FALSE** in such a way that the formula evaluates to **TRUE**. If this is the case, the formula is called *satisfiable*. On the other hand, if no such assignment exists, the function expressed by the formula is **FALSE** for all possible variable assignments and the formula is *unsatisfiable*.

❖ **Problem:** Given a boolean formula ***f***, the problem is to identify if the formula ***f*** has a satisfying assignment or not.

❖ **Explanation:** An instance of the problem is an input specified to the problem. An instance of the problem is a boolean formula ***f***. Since an NP-complete problem is a problem which is both **NP** and **NP-Hard**, the proof or statement that a problem is NP-Complete consists of two parts:

1. *The problem itself is in NP class.*
2. *All other problems in NP class can be polynomial-time reducible to that.*
   (B is polynomial-time reducible to C is denoted as ≤ PC)

If the 2nd condition is only satisfied then the problem is called **NP-Hard**.
But it is not possible to reduce every NP problem into another NP problem to show its NP-Completeness all the time i.e., to show a problem is NP-complete then prove that the problem is in NP and any NP-Complete problem is reducible to that i.e. if B is NP-Complete and B ≤ PC For C in NP, then C is NP-Complete. Thus, it can be verified that the **SAT Problem** is NP-Complete using the following propositions:

➢ **SAT is in NP:**
It any problem is in NP, then given a 'certificate', which is a solution to the problem and an instance of the problem(a boolean formula ***f***) we will be able to check (identify if the solution is correct or not) certificate in polynomial time. This can be done by checking if the given assignment of variables satisfies the boolean formula.

➢ **SAT is NP-Hard:**
In order to prove that this problem is NP-Hard then reduce a known problem, Circuit-SAT in this case to our problem. The boolean circuit **C** can be corrected into a boolean formula as:

Ex :KNAPSACK solution in details and pseudocode

The **knapsack problem** is the following problem in combinatorial optimization:

Given a set of items, each with a weight and a value, determine which items to include in the collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items. The problem often arises in resource allocation where the decision-makers have to choose from a set of non-divisible projects or tasks under a fixed budget or time constraint, respectively.

The knapsack problem has been studied for more than a century, with early works dating as far back as 1897.[1]

0-1 Knapsack Problem

In this variant, the items are defined as:

Here, there is only one of each item   available. So if   is , that means the item is not added to the knapsack. If   is , that means the item is added to the knapsack.

The decision version of the 0-1 knapsack problem is an NP-Complete problem.

Let's see why.

Given weights and values of items and respectively, can a subset of items be picked that satisfy the following constraints:

A 'Yes' or 'No' solution to the above decision problem is NP-Complete. Solving the above inequalities is the same as solving the Subset-Sum Problem, which is proven to be NP-Complete. Therefore, the knapsack problem can be reduced to the Subset-Sum problem in polynomial time.

Further, the complexity of this problem depends on the size of the input values   ,   . That is, if there is a way of rounding off the values making them more restricted, then we'd have a polynomial-time algorithm.

This is to say that the non-deterministic part of the algorithm lies in the size of the input. **When the inputs are binary, it's complexity becomes exponential, hence making it an NP-Complete problem**