NP-Completeness

Noman Ahmad

Hunter College Department of Computer Science

December 19, 2020

Abstract

During this course we learned various techniques to solve algorithms as well as various famous algorithms in the field of computing. We learned about various sub-topics in the field such as Divide and Conquer Algorithms, Randomized Algorithms, Hashing, Red-Black Trees, Heaps, Sorting, and Graph Theory. We learned how to analyze different problems in order to optimize the time complexity of it's solution. All the algorithms we analyzed and solved in this class can be classified as polynomial-time algorithms. Not all problems have polynomial-time solutions, some problem's have no solution at all. This paper analyzes the different classes of problems and their relevance in the field of computing.

Polynomial-Time

All the problems we discussed in class can be solved in O(n^k) time, for some constant k. These algorithms are known as polynomial-time algorithms, often referred to as the class of P. A proper distinction for this class of problems, are problems whose running time is given as a function that is polynomial on the size of the input or it can be bounded by a polynomial function with respect to it's input. These problems are also referred to sometimes, as tractable or easy. Examples of these polynomial running times are O(1), O(lgn), O($\log_k$n), O(n), O(n$\log_k$n), O(n^2), O(n^3), O(n^{100}), O($n^{log_k c}$). Linear Search, Binary Search, Insertion Sort, Merge Sort, Heap Sort, Randomized Quick Sort, Median-Finding, Order Statistics, Hashing, Krusgal's Algorithm, Prim's Algorithm, Breadth-First Search, Depth-First Search, Bellman-Ford Algorithm, Dijkstra's Algorithm, Topological Sorting, Floyd-Warshall Algorithm, and Strassen's Algorithm are all examples of Polynomial-Time Solutions to problems. The problems that these solve are known as easy or tractable. There exists problems that can't be solved in polynomial time, and these form some interesting classifications on their own[3][1].

Non-Deterministic Polynomial Time

The class of problems known as NP (Non-Deterministic Polynomial) consists of problems that are verifiable in polynomial time, meaning we can verify that their solution is correct in polynomial time however, the solution itself is not a polynomial-time solution. An example of this is finding the Hamiltonian Cycle of a directed graph. Given a directed graph $G = (V,E)$, we would like to determine a cycle that contains each vertex in V. While this happens to be a NP Problem, verifying that a solution contains all the vertices can be done in polynomial time. By implication, all problems in P are also in NP because all problems in P can be solved in polynomial time without need for verification. An interesting aspect of NP problems, is the entire subset of NP-Complete problems. These problems are all NP, however they are known to be the "hardest" problems in the entire set of NP[1].

NP-Complete Problems

As mentioned before, NP Complete problems are problems in the set of NP. The distinguishing factor here is that all of these problems are related in a form of "reduction". This basically means that any problem that is in the set of NP Complete, there exists a polynomial-time algorithm that can transform the problem into any other NP Complete Problem. So theoretically, if we were to find a polynomial-time solution to any one of these problems, that would mean that we can solve all of these NP Complete Problems in polynomial time[1].

The Knapsack Problem

An example of an NP Complete Problem is the Knapsack Problem. The problem statement is that we are given a knapsack with a weight limit of W. a collection of n items with values v_1,v_2,v_3,...,v_n and weights w_1,w_2,w_3,...,w_n, and we would like to optimize the problem and produce the following:

$$\max(\sum_{i=1}^{n} v_i)$$

such that

$$\sum_{i=1}^{n} w_i \leq W$$

This is basically asking, to find the maximum value of the items that can be added to the knapsack such that the total weight does not exceed the weight limit W. For the brute force solution to this problem, we can obviously find all possible subsets, which would give us a complexity of $O(2^n)$. In order to optimize, we can take a look at a dynamic programming solution. In order to represent the problem in terms of smaller problems, we will use an 2D array

or matrix, V[0...n,0...W]. In the end, V[n,W] will contain the maximum and optimal value of the items that can be added to the knapsack. Next, we need to define the value of the optimal solution in terms of solutions to smaller problems. Initially we will say that:

$$V[0, w] = 0 \quad \text{for} \quad 0 \leq w \leq W$$

$$V[i, w] = -\infty \quad \text{for} \quad w < 0$$

Then, we have to define the recursive solution to the sub problems, storing them in V to be used by the later sub problems:

$$V[i, j] = \max(V[i-1, j], v_i + V[i-1, j-w_i]) \quad \text{for} \quad 1 \leq i \leq n, 0 \leq j \leq W.$$

This method compares the item that is right above the current problem in the solutions matrix V. It either selects that to be the optimal solution for $V[i, w]$ or it adds the value of $V[i, w]$ to that solution. The pseudo code for this algorithm is the following:

Algorithm 1 Knapsack Algorithm Implementation

```
1:  function KNAPSACK(V,W,N,W):
2:      for i ← 0 to W:
3:          V[0, i] ← 0
4:      for i ← 1 to n:
5:          for j ← 0 to W:
6:              if (w[i] less than or equal to j):
7:                  V[i, j] ← max(V[i − 1, j], v[i] + V[i − 1, j − w[i]])
8:              else:
9:                  V[i, j] ← V[i − 1, j]
10:     return V[n, W]
```

The complexity of this algorithm is clearly $O(nW)$. The initialization are done in O(1) time, and the max is computed also in O(1) time because it just accesses elements from the matrix, the nested loop iterates up to W for every n therefore it is $O(n * W)$. Now this might seem like a polynomial-time algorithm with respect to n, however it is W that makes it exponential. This is because the complexity is really exponential to the bit length of W, even though we take n * W steps in our algorithm, we see that as we increase the bit-length of W, like for example from 1000 to 10000000 increasing the binary representation from 4-bits to 8-bits, the complexity grows exponentially with W, not n. Therefore, this is an NP problem. These kind of problems which seem linear because they grow linear with n, but are actually exponential, are known as pseudo-polynomial algorithms[2].

NP-Hard

The final set of problems is classified under the NP-Hard subset. This set is comprised of problems that are the hardest and most complex problems in

computer science, not just to solve but also to verify. To understand NP Hard problems first we need to make a proper distinction between Decision Problems and Optimization Problems. Decision problems are those that ask for simple "yes" or "no" answers. Optimization problems are the kind of problems in which the answer is the solution with the best answer in comparison to other feasible solutions. The interesting idea is that most optimization problems can be given a related decision problem. For example consider the problem of Shortest-Path, where we are given an undirected graph $G = (V,E)$, and vertices u and v, and we would like to find a path such that the number of edges is minimized. We can translate this to a decision problem. Given the same graph G and vertices u and v and a parameter k, does a path exist from u to v, consisting of at most k edges? Now this relationship is important when trying to classify problems as easy or hard. If an optimization problem is easy, then a related decision problem is also easy. However, if we can show that a decision problem is hard, then its related optimization problem must be just as hard. For NP Hard problems, not only are they hard to compute and verify, some of these problems don't have decision problems to relate to. Unlike NP and NP-Complete which can be verified in polynomial time, some NP Hard problems have been proven to not be verifiable in polynomial time. Another interesting property is that if we know that the decision problem is within NP-Complete, then the optimization of such a problem has to be in NP-Hard, which is the case with the popular Traveling Salesman Problem[1].

Conclusion

In this report, we looked at some basic ideas of the set of problems in computer science as well as how they can be classified due to their complexity. We examined the differences between P, NP, NP Complete and NP Hard Problems. We also looked at the Knapsack problem which we were able to show is NP Complete despite seeming like a polynomial time algorithm. There are obviously a lot more of these problems being researched and discussed in the field of computer science, but these were some basic high-level ideas.

References

[1] Thomas H. Cormen and Charles E. Leiserson. *Introduction to algorithms, 3rd edition.* 2009.

[2] Subham Datta. *0-1 Knapsack: A Problem With NP-Completeness and Solvable in Pseudo-Polynomial Time.* Aug 2020.

[3] David Matuszek. *Polynomial-Time Algorithms.* University of Pennsylvania, Apr 1996.