Assignment 4 - P, NP, NP-Hard & NP-Complete Problems

Cathal Lawlor - 21325456

1. Polynomial Problems:

Definition

A polynomial problem is an algorithm that we can solve in polynomial time (a quantity consisting of variables and coefficients), such as something in linear, logarithmic or quadratic time. It is a number that grows at a reasonable rate, unlike exponential or factorial time complexities which are not polynomial-time and is not desired/used in algorithms as they grow quite fast becoming practically impossible to compute.

This means that there is an algorithm that can solve it, with the algorithm performing a number of steps, limited by a polynomial function of *n,* with *n* being the input length used in the algorithm. [i]

There can be multiple inputs such as *n*'s, but once that every one of the inputs is a polynomial, then the result of their multiplication of is still a polynomial.

If we have a polynomial problem, the formal definition for its time complexity is: [ii]

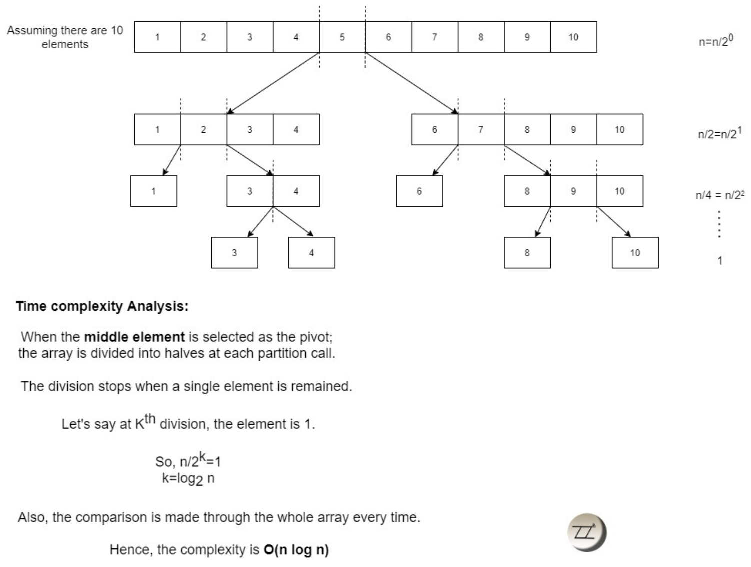
$$T(n) = o(C * n^k)$$

Where C > 0 and k > 0. C and k are both constants, with *n* being the input size. It is expected that in general k is less than *n* in a polynomial-time problem.

Example

An example of a polynomial time-based problem is sorting an unsorted list of n numbers, where they are ordered into a descending or ascending order.

One common and well-known solution is the Quicksort algorithm, which recursively partitions the input list into two smaller sub-lists, with all elements smaller than a selected pivot in one, and the other sub-list holding all elements greater than or equal to the pivot. The pivot element is placed in its final position and the algorithm is applied recursively to the two sub-lists on either side, and so on, until the sub-lists are either 0 or 1.



Time complexity Analysis:

When the **middle element** is selected as the pivot;
the array is divided into halves at each partition call.

The division stops when a single element is remained.

Let's say at K^{th} division, the element is 1.

So, $n/2^k=1$
$k=\log_2 n$

Also, the comparison is made through the whole array every time.

Hence, the complexity is **O(n log n)**

The time complexity is on average O(n log n) with a worst case of O(n^2), meaning that the running time of Quicksort only grows at most as a polynomial number of n, with n being the size of the input list of numbers. It is bound then by either of the time complexities. This makes QuickSort a polynomial-time algorithm.

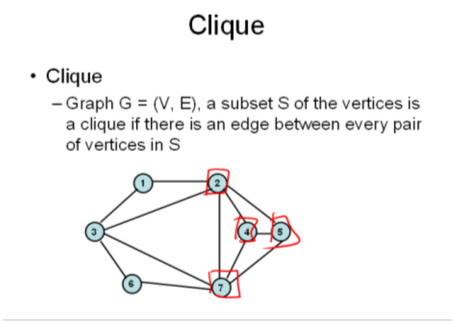
2. Non-deterministic Polynomial Problems:

Definition

A non-deterministic polynomial problem is where a proposed solution can be verified or checked in polynomial-time, with the clause that there is no known algorithm for a deterministic Turing machine to solve the problem in polynomial-time.[iii]

A deterministic Turing machine (DTM) is a theoretical simplified computer, with a clear set of instructions that it follows. This means that it will know exactly what the next action is given an input and past state. Example: a symbol is read in as an input, and will know exactly the next action to perform based on the symbol and the current state of the machine. A non-deterministic machine can have multiple possible next actions to choose from given the same inputs. (we generally would associate quantum computing with this).

A NP problem is a problem that can have its solution checked in polynomial time, but finding a solution cannot be done in polynomial-time by a known algorithm. (Non-deterministic Turing machines might be able to find a solution in polynomial time though[iv])



Clique

- Clique
  - Graph G = (V, E), a subset S of the vertices is a clique if there is an edge between every pair of vertices in S

Example

Clique problem – we are given a graph G(V, E), check for clique(s) of size x in G. A subset S of vertices is a clique, where every pair of vertices must be connected, by having an edge between each other in S.

We can check whether S is a clique by checking that every pair of vertices in the subset is connected by an edge in E. This verification can be done in $O(x^2)$ time, which has an input that is polynomial in size. However, there is no known algorithm to solve all instances of the clique problem in polynomial-time.

Therefore, the Clique problem is in NP. The Clique problem was published as part of one of the 21 NP-complete problems published by Karp[v], and is NP-Hard, as it is reducible from the Boolean satisfiability problem.

3. NP-Hard Problems:

Definition

A problem is in the NP-Hard problem class, if there exists another problem that is reducible to the problem in question. Suppose problem ‘A’, if a problem ‘B’ exists, that is reducible to ‘A’, where it takes P time to reduce ‘B’ to ‘A’, then ‘A’ is NP-Hard.
This is regardless of the matter if the NP-Hard problem is a nondeterministic polynomial problem (NP), as NP-Hard problems aren’t required to / don’t always have their solution verifiable in polynomial-time. Outside of that, a NP-Hard problem is as hard as NP-Complete problems to solve.[vi]

Example

The Travelling Salesman problem (TSP) is a NP-Complete problem, again proven by Richard Karp. As it is NP-Complete, it must be at least as hard to solve as any other NP-complete problem in the 21 NP-Complete problems.

It is finding the shortest possible path between that visits every vertex in a graph, each representing a city, and returning to the starting point. The problem is, given an integer L, determine whether there exists a Hamiltonian cycle of length at most L.

TSP is in NP, because given a proposed Hamiltonian cycle, it is easy to verify whether it is a valid solution in polynomial-time. Every instance of the Traveling Salesman Problem can be transformed into an instance of the Hamiltonian Cycle Problem through reduction[vii]. Therefore, the Travelling Salesman problem is NP-Hard.

4. NP-Complete Problems:

Definition

NP-Complete problems must be in the NP problem class, taking only polynomial-time to be verified, but being unable to be solved in a polynomial-time manner by a DTM, where the class represents all problems ‘A’ in NP as a set, with any other NP problem ‘B’ must be able to be reducible to ‘A’, taking only polynomial-time to achieve this. [viii]

This is where we know how to solve ‘B’ in polynomial-time if we know how to solve ‘A’ in polynomial-time. Reduction is where algorithm_A solves ‘A’, and a transformation from ‘B’ to ‘A’, permits algorithm_A to be incorporated into algorithm_B to help solve ‘B’.

To be in the NP-complete problem class, the problem 'A' must be part of the NP problems class, with every problem in NP able to be reducible to 'A' in polynomial time. If there is an algorithm that is in polynomial-time for any of the NP-Complete, it means there is a polynomial-time algorithm that can solve every problem in NP.

Example

The Graph k-Colouring problem is a NP-Complete problem[ix]. With an undirected graph and positive integer k, the problem is determining whether there exists a possible arrangement, to colour the vertices of the graph using k or fewer colours where no two neighbouring / adjacent vertices have the same colour.

The Graph Colouring problem is in NP because given a colouring of the vertices, we can easily verify whether it is a valid colouring in polynomial time, but as the Boolean satisfiability problem (SAT) – a NP-Hard problem is reducible to it by creating a graph where vertices represent variables and edges represent clauses. We attempt to colour the graph using two colours, where true variables are one colour and false variables are the other colour. If it's possible to colour the graph this way, then the SAT formula is satisfiable, otherwise, it's not. This makes graph k-colouring NP-Hard, making it NP-Complete.

5. History of the *‘P versus NP’* problem

The argument of ‘P versus NP’ in layman’s terms, is the question, is every polynomial problem in NP? It is hotly debated and there have been numerous papers arguing for and against the statement ‘P = NP’, still to today with no general consensus on an answer.

It has been proven, any problems that a non-deterministic Turing machines can solve, can be solved by deterministic Turing machines, with the only catch of not knowing how long a deterministic Turing machine will take.

The ‘P = NP’ question was explicitly stated using the terms P and NP back in 1971 in a paper published by Stephen Cook, titled “The complexity of theorem proving procedures”[x], though the first questions around the topic of problem were mentioned as far back in as in the 1950’s where a mathematician named John Nash had contacted the NSA about encryption systems[xi].

Nash had sent letters on encryption and proposed remarkably modern ideas for encryption systems, like ones that we see being used in the current day. Based on his ideas he had come to a belief that “it is quite feasible to design ciphers that are effectively unbreakable”. He then continues, to explain how he “cannot prove it” and neither does he “expect it to be proven”. Something that is a common opinion today.

If this conjecture of his had been proven, it would equate to implying P ≠ NP for us. In Cooks paper, it is detailed: "Another important open question is whether P = NP. If we can find a method for solving problems in NP in polynomial time, then P = NP. Since any NP problem can be reduced to the SAT problem in polynomial time, it suffices to find a polynomial time algorithm for SAT in order to establish P = NP."

These lines suggest that if we find an algorithm runnable in polynomial-time for the Boolean satisfiability problem (SAT), any problem in NP could now be solved in polynomial time. This would prove that P = NP. Even in the decades of research since, we have been unable to find such an algorithm.

There have been countless efforts to prove P isn’t equal to NP. Recently an effort tried to prove this theorem by trying to show some currently believed NP-Complete problems cannot be solved efficiently by any algorithm. These efforts like earlier have been unable to prove this.

If P isn’t equals to NP, then these problems are going to remain fundamentally difficult to solve, where we might never get the best solution even with development of best techniques and algorithms in these circumstances. Researchers believe it is very difficult to prove, with some believing it is fundamentally unsolvable.[xii]

References

1. [i]https://www.britannica.com/science/P-versus-NP-problem
2. [ii] https://www.baeldung.com/cs/p-np-np-complete-np-hard
3. [iii] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms. MIT press. P.1053
4. [iv] *Kleinberg, Jon; Tardos, Éva (2006). Algorithm Design (2nd ed.). Addison-Wesley. p. 464. ISBN 0-321-37291-3.*
5. [v] *Karp, Richard M. (1972). "Reducibility among combinatorial problems". In Miller, Raymond E.; Thatcher, James W. (eds.). Complexity of Computer Computations. Plenum. pp. 85–103.*
6. [vi] https://www.youtube.com/watch?v=moPtwq_cVH8
7. [vii] https://www.geeksforgeeks.org/proof-that-traveling-salesman-problem-is-np-hard/
8. [viii] Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest p.1067
9. [ix] *Karp, Richard M. (1972). "Reducibility among combinatorial problems". In Miller, Raymond E.; Thatcher, James W. (eds.). Complexity of Computer Computations. Plenum. pp. 85–103.*
10. [x] *Cook, Stephen (1971). "The complexity of theorem proving procedures". Proceedings of the Third Annual ACM Symposium on Theory of Computing. pp. 151–158. doi:10.1145/800157.805047. ISBN 9781450374644. S2CID 7573663.*
11. [xi] *NSA (2012). "Letters from John Nash" (PDF). Archived (PDF) from the original on 9 November 2018.*
12. [xii] Is P Versus NP Formally Independent? Scott Aaronson∗ University of California, Berkeley p.16