CHAPTER 4

INTRODUCTION TO NP-COMPLETENESS

4.1 Polynomial and exponential algorithms

Given an undirected graph G, we can ask whether G admits an Euler cycle. An Eulerian cycle is a cycle that traverses each edge exactly once. A path is labelled below- the edges are numbered.

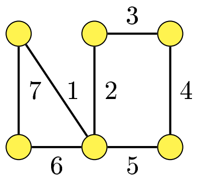


Figure 4.1: A traversal of the graph that results in an Eulerian cycle.

We take an edge precisely once (but have visited two vertices twice). There is a theorem that states that a connected undirected graph has an Euler cycle if and only if each vertex has an even degree. Therefore, we can test whether a graph has an Eulerian cycle by checking the in-degree of each vertex. So, if we represent the graph as an adjacency matrix, the algorithm is $O(|V|^2)$, while if we use an adjacency list, it is $O(|V| + |E|)$.

Given an undirected graph G, we can also ask whether G admits a Hamiltonian cycle. A Hamiltonian cycle is a cycle that visits each vertex precisely once. For example, the graph below has a Hamiltonian cycle shown in red.

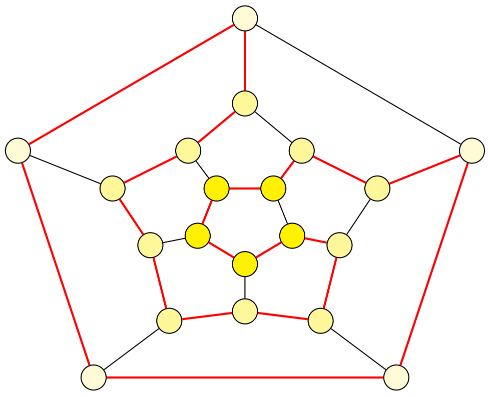


Figure 4.2: A traversal of the graph that results in a Hamiltonian cycle. The traversal is shown in red.

Superficially, the problem seems similar to the Euler cycle problem. However, in an algorithmic sense, they are very different. In fact, nobody has found a polynomial-time algorithm for Hamiltonian cycle. Also, nobody has shown that there does not exist a polynomial-time algorithm to find a Hamiltonian cycle.

One way of finding a Hamiltonian cycle is to generate all the permutations of the vertices. We then check to see if it is a cycle, i.e. the corresponding edges are present. If we have n vertices, then there are $n!$ permutations. For each permutation σ, it takes $O(n^2)$ to check whether σ is a Hamiltonian cycle (assuming G is represented by adjacency lists). In the worst case, to check an edge is present, we have to traverse the adjacency list of length $n-1$, and there are n edges to check. So, the worst-case number of operations in the brute force approach to find a Hamiltonian cycle using adjacency lists is $O(n^2 n!)$. This is an example of an exponential algorithm.

The table below shows the running time of algorithms with various complexities.

	40	50	60	70
n	.00003 secs	.0004 secs	.0005 secs	.0006 secs
n^2	.0009 secs	.0016 secs	.0025 secs	.0036 secs
n^3	.027 secs	.064 secs	.125 secs	.216 secs
n^5	24.3 secs	1.7 mins	5.2 mins	13.0 mins
2^n	17.9 mins	12.7 days	35.7 years	366 cents
3^n	6.5 years	2855 cents	2×10^8 cents	1.3×10^{13} cents
$n!$	8.4×10^{16} cents	2.6×10^{32} cents	9.6×10^{48} cents	2.6×10^{66} cents

Table 4.1: The running time of algorithms with various complexities (assuming 10^9 operations per second).

Clearly, as n grows, the distinction between polynomial and exponential time algorithms becomes dramatic.

This behaviour still applies even with increases in computing power. Assume that for each time complexity, we have a size N of the largest instance solvable in an hour on a normal computer. The table below shows how the complexity changes as the computer gets faster.

	current	100 times faster	1000 times faster
n	N_1	$100 \cdot N_1$	$1000 \cdot N_1$
n^2	N_2	$10 \cdot N_2$	$31.6 \cdot N_2$
n^3	N_3	$4.64 \cdot N_3$	$10 \cdot N_3$
n^5	N_4	$2.5 \cdot N_4$	$3.98 \cdot N_4$
2^n	N_5	$N_5 + 6.64$	$N_5 + 9.97$
3^n	N_6	$N_6 + 4.19$	$N_6 + 6.29$
$n!$	N_7	$\leq N_7 + 1$	$\leq N_7 + 1$

Table 4.2: Maximum improvement in computation with increasing computing power.

Again, we can see that for polynomial-time algorithms, there is a significant improvement. However, for the exponential algorithms, there is little to no increase.

In summary, exponential-time algorithms are ‘bad’ in general. If we increase the processor speed, there is no significant change in this slow behaviour when the input size is large. On the other hand, polynomial-time algorithms are ‘good’ in general. Often, polynomial-time algorithms require some insight, while exponential-time algorithms are variations on exhaustive search. A problem is polynomial-time solvable if it admits a polynomial-time algorithm.

4.2 NP-Complete Problems

There is a set of problems, called NP-complete, for which no polynomial-time algorithm is known. However, if one of them is solvable in polynomial time, then they all are. On the other hand, we also do not know whether they cannot be solved in polynomial time. But, if one of them is not solvable in polynomial time, then they all aren't solvable.

There are two reasons for a problem to be NP-complete. Either the polynomial time is not sufficient in order to discover a solution, or the solution itself is so large that exponential time is needed to output it. We will only focus on the first case.

There are many problems that are known to be NP-complete. Moreover, there are some problems have been shown to be undecidable, i.e. there cannot be an algorithm that could solve the problem.

The roadblock problem is a decidable problem that is not solvable in polynomial time. There are two players- A and B, along with a network of roads, comprising intersections connected by roads. Each road is coloured either black, blue or green. Some intersections are marked either 'A wins' or 'B wins'. A player has a fleet of cars at intersections, with at most one per intersection. Player A begins, and subsequently players make moves in turn. They move one of their cars on one or more roads of the same colour. A car may not stop at or pass over an intersection which already has a car. The problem is to decide, for a starting configuration, whether A can win, regardless of what moves B makes.

In summary, there are two classes of problems- polynomial-time solvable problems and intractable problems. We do not know where NP complete problems belong.

A problem is characterised by parameters. Typically, there are infinitely many instances for a given problem. A problem instance is created by giving these parameters values. For example, the Hamiltonian cycle is NP-complete. The problem is:

- **Name**: Hamiltonian Cycle (HC)
- **Instance**: a graph G
- **Question**: does G contain a cycle that visits each vertex exactly once?

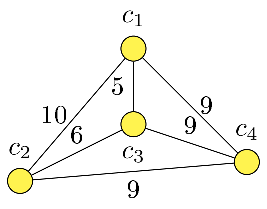
This is an example of a decision problem- for the question, the answer we want is either `true` or `false`. So, every instance is either a 'yes-instance' or a 'no-instance'.

Another NP-complete problem is the Travelling Salesman Problem, shown below:

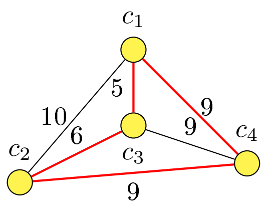
- **Name**: Travelling Salesman Decision Problem (TSDP)
- **Instance**: a set of n cities and integer distance $d(i, j)$ between each pair of cities i, j, and a target integer K
- **Question**: is there a permutation $p_1, p_2, \ldots, p_{n-1}, p_n$ of the cities such that
$$d(p_1, p_2) + d(p_2, p_3) + \cdots + d(p_{n-1}, p_n), d(p_n, p_1) \leq K.$$

> That is, can we find a path to travel from one city back to itself (going through all the cities) such that the total distance travelled is less than K- such a cycle is called a travelling salesman tour.

For example, consider the following graph representation of the cities.



There is a travelling salesman tour of length 29 when we traverse through the following edges.

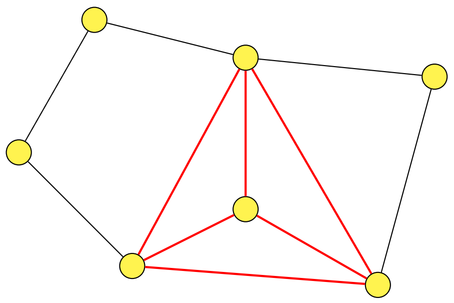


However, there is no tour of length less than 29. It has been shown that the TSDP is NP-complete.

Another NP-complete problem is the clique problem:

- **Name**: Clique Problem (CP)
- **Instance**: a graph G and a target integer K
- **Question**: does G contain a clique of size K? That is, does G have a set of K vertices for which there is an edge between all the pairs.

For example, the graph below has a clique of size 4, as shown in red.



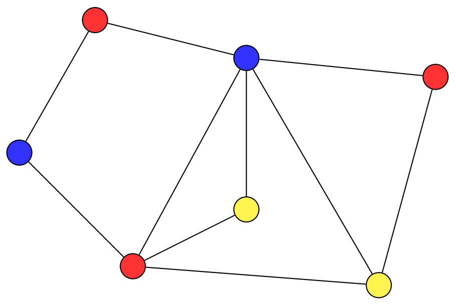
However, the graph does not have a clique of size 5.

The Graph Colouring Problem is also NP-complete. The problem is:

- **Name**: Graph Colouring Problem (GCP)

- **Instance**: a graph G and a target integer K
- **Question**: can one of the K colours be attached to each vertex of G such that the adjacent vertices always have different colours?

For example, the graph below can be coloured using 3 colours, as shown below.



However, there is no colouring using 2 colours.

Finally, we look at the satisfiability problem:

- **Name**: Satisfiability (SAT)
- **Instance**: Boolean expression B in conjunctive normal form (CNF). A CNF is of the form $C_1 \wedge C_2 \wedge \cdots \wedge C_n$, where each C_i is a clause. A clause is of the form $l_1 \vee l_2 \vee \cdots \vee l_m$, where each l_j is a literal. Finally, a literal is a variable x (which can either be `true` or `false`) and its negation $\neg x$.
- **Question**: Is B satisfiable? That is, can we assign the variables `true` or `false` so that the expression B is true.

For example, the following boolean expression is in CNF:

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee x_4).$$

Also, the expression above is satisfiable if we set $x_1 =$ `true`, $x_2 =$ `false`, $x_3 =$ `true`, $x_4 =$ `true`. The satisfiability problem is NP-complete.

The problems we have looked at have been decision problems- they return `true` or `false`. However, we are typically interested in optimisation problem, where we actually have the maximum or the minimum value. For example, the travelling salesman optimisation problem (TSOP) finds the minimum length of a tour. We also have search problems, where we find some appropriate optimal structure. For example, the travelling salesman search problem (TSSP) finds the minimum length tour.

NP-completeness primarily deals with decision problems, but corresponding to each instance of an optimisation or a search problem is a family of instances of a decision problem by setting the 'target' values. Almost invariably, an optimisation or a search problem can be solved in polynomial time if and only if the corresponding decision problem can.

4.3 P and NP

The class P is the set of all decision problems that can be solved in polynomial time. There are many problems in P, e.g.

- Is there a path of length $\leq K$ from vertex u to vertex v in a graph G?
- Is there a spanning tree of weight $\leq K$ in a graph G?
- Is a graph G bipartile?
- Is a graph G connected?
- Does a directed graph D contain a cycle?
- Does a text t contain an occurrence of a string s?
- For two strings s and t, is their distance $d(s, t) \leq K$?

P is often extended to include search and optimisation problems.

The class NP is the set of decision problems that can be solved in non-deterministic polynomial time. A non-deterministic algorithm can make non-deterministic (random) choices, so it can give different answers depending on when we run it. Hence, it is apparently more powerful than a normal deterministic algorithm.

Clearly, P is contained within NP- a deterministic algorithm is just a special case of a non-deterministic one. But, the containment may not be strict. Currently, there is no problem known to be in NP and not in P.

A non-deterministic algorithm (NDA) have an extra operation- they can make a non-deterministic choice. A NDA has many possible executions depending on which values were chosen during the process. We say that NDA 'solves' a decision problem Π if for a yes-instance I of Π, there is some execution that returns `true`, and for any no-instance I of Π, there is no execution that returns `true`. Moreover, the NDA 'solves' a decision problem Π in polynomial time if for a yes-instance I of Π, there is some execution that returns `true` which uses a number of steps bounded by a polynomial in the input, and for a no-instance, there is no execution that returns `true`.

Clearly, NDAs are not useful in practice since they do not always return the right answer. However, they are a useful mathematical concept for defining the classes of NP and NP-complete problems.

We will now provide a NDA that 'solves' the graph colouring problem.

```
 1 bool graphColouring(Graph graph, int k):
 2     // choose one of the k colours for each vertex
 3     for Vertex v in graph:
 4         v.colour = Colour.random(k)
 5
 6     // check whether vertices with an edge are coloured different
 7     for Edge edge in graph.edges:
 8         if edge.to.colour == edge.from.colour:
 9             return false
10     return true
```

In line 4, we make a non-deterministic choice for the colour of a vertex from the set of k colours. We then check whether the coloured graph satisfies the graph colouring problem. The operations take polynomial time for every execution of the algorithm.

In general, a NDA can be viewed as having a guessing stage (which is non-deterministic), and a checking stage (which is determinstic). The guessing stage is usually $O(n)$. So, for a problem to be in NP, we need the checking stage to run in polynomial time.

4.4 Polynomial-time reductions

A polynomial-time reduction (PTR) is a mapping f from a decision problem Π_1 to a decision problem Π_2 such that for every instance I_1 of Π_1, the instance $f(I_1)$ of Π_2 can be constructed in polynomial time. Moreover, $f(I_1)$ is a yes-instance of Π_2 if and only if I_1 is a yes-instance of Π_1. We denote this as $\Pi_1 \propto \Pi_2$.

Polynomial-time reductions are transitive. That is, if we have decision problems Π_1, Π_2, Π_3 such that $\Pi_1 \propto \Pi_2$ and $\Pi_2 \propto \Pi_3$, then $\Pi_1 \propto \Pi_3$. By definition, there exist PTRs f from Π_1 to Π_2 and g from Π_2 to Π_3. Now, for any instance I_1 of Π_1, we can create an instance $f(I_1)$ of Π_2 in polynomial time. By definition, $f(I_1)$ has the same answer as I_1. Moreover, g is a PTR as well, so $g(f(I_1))$ is an instance of Π_3 that we can construct from $f(I_1)$ in polynomial time. By definition, $g(f(I_1))$ has the same answer as $f(I_1)$. Therefore, the composition $g \circ f$ is a polynomial time reduction from Π_1 to Π_3 since $(g \circ f)(I_1)$ has the same answer as I_1.

If we have $\Pi_1 \propto \Pi_2$ and $\Pi_2 \in P$, then $\Pi_1 \in P$ as well. To solve an instance of Π_1, we can reduce it to an instance of Π_2 in polynomial-time. Then, we can solve Π_2 in polynomial-time and return the result. The entire process takes polynomial time, so $\Pi_1 \in P$.

Roughly speaking, $\Pi_1 \propto \Pi_2$ means that Π_1 is 'no harder' than Π_2. That is, if we can solve Π_2, then we can solve Π_1 without much more effort- we just need to perform a polynomial time reduction. It is also possible that Π_2 is harder to solve than Π_1. We may only map the instances of Π_1 to those in Π_2 that are 'easy' to solve.

We can reduce the Hamiltonian cycle problem to the travelling salesman problem. For instance, assume that we are given the following instance of the Hamiltonian cycle problem.

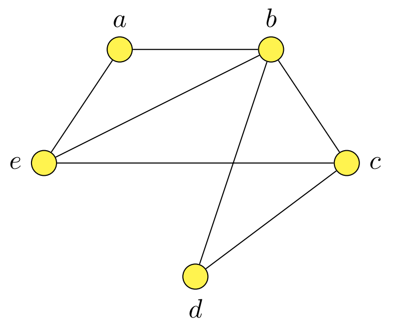


Figure 4.3: An instance of the Hamiltonian Cycle problem.

We will construct a TSDP corresponding to this graph. So, let the vertices be the cities in the problem. Moreover, for two cities x, y, we set $d(x, y) = 1$ if there is an edge $\{x, y\}$ in the graph above, and $d(x, y) = 2$ otherwise. Then, we can set K to be the number of edges. This gives us the following graph.

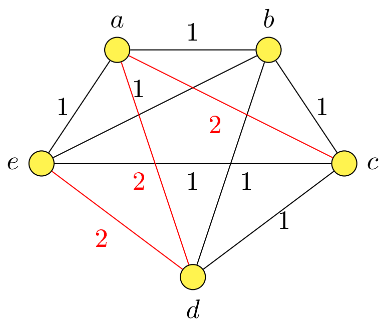


Figure 4.4: An instance of the Travelling Salesman Problem

All these operations can be performed in polynomial-time. Moreover, the graph has a travelling salesman tour of length $\leq K$ if and only if the original graph has a Hamiltonian cycle. The tour includes $|E|$ edges, so we cannot take any edges with weight 2. So, if the travelling salesman problem is in P, then the Hamiltonian cycle problem is in P as well.

We now define NP-completeness. We say that a decision problem Π is NP-complete if Π is in NP, and for every problem Π' in NP, the problem Π' reduces to Π in polynomial-time. If Π is NP-complete and Π is in P as well, then we have P = NP. Every problem in NP can be solved in polynomial time by reduction to Π. Supposing P $\neq$ NP, if Π is NP-complete, then Π is not in P.

To prove that a problem is NP-complete, it is not feasible to describe a reduction from every problem in NP. However, if we know that a problem Π_1 is NP-complete, then to show that another problem Π_2 is NP-complete, we need to show that Π_2 is in NP and that there exists a polynomial-time reduction from Π_1 to Π_2. Since Π_1 is NP-complete, we know that for any problem Π in NP, we have $\Pi \propto \Pi_1$. We also know that $\Pi_1 \propto \Pi_2$. Since $\propto$ is transitive, it follows that $\Pi \propto \Pi_2$. Therefore, Π_2 is NP-complete.

We know that the satisfiability problem (SAT) is NP-complete. The proof consists of a generic polynomial-time reduction to SAT from an abstract definition of a general problem in the class NP. The generic reduction could be instantiated to give an actual reduction for each individual NP problem. So, to prove that a decision problem Π is NP-complete, we need to show that Π is in NP and that there is some polynomial-time that reduces SAT to Π.

We will use this technique to show that the Clique problem (CP) is NP-complete. To prove CP is NP-complete, we show that CP is in NP and that there exists a polynomial-time reduction from SAT to CP. The following is an NDA that solves the CP.

```
bool clique(Graph graph, int k):
    List<Vertex> vertices = []
    // choose k vertices from the graph
    for int i in range(0, k):
        vertices.add(graph.vertices.random())

    // check every distinct pair of vertices has an edge
    for Vertex v1 in vertices:
        for Vertex v2 in vertices:
            if v1 != v2 and not graph.hasEdge(v1, v2):
                return false
    return true
```

We can guess the solution in $O(K)$ time, and check in $O(K^2)$. Therefore, the problem is in NP.

Next, we show that there is a polynomial-time reduction from SAT to CP. So, assume that we are given an instance B of SAT. We construct an instance (G, K) of CP. We set K to be the number of clauses of B. The vertices of G are (l, C), where l is a literal in clause C. We add an edge $\{(l, C), (m, D)\}$ if $l \neq \neg m$ and $C \neq D$. That is, we add an edge if distinct literals from different clauses can be satisfied simultaneously. This is a polynomial-time algorithm since the construction takes $O(n^2)$- in the worst case, we need to compare every literal with every other literal when constructing the edges.

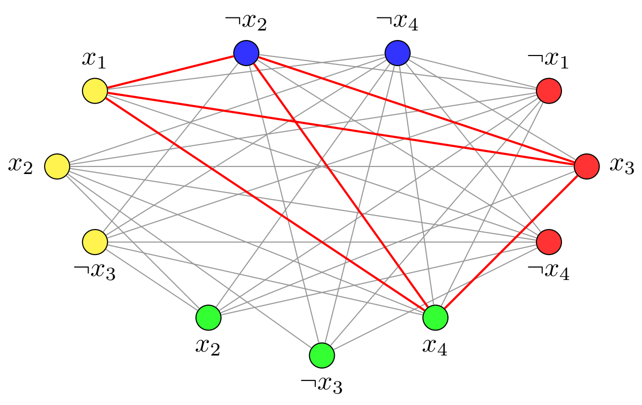
Now, if B has a satisfying assignment, then if we choose a `true` literal in each clause, the corresponding vertices form a clique of size K in G. On the other hand, if G has a clique of size K, then assigning each literal associated with a vertex in the clique to be `true` yields a satisfying assignment for B.

We only have edges between literals in distinct clauses, so if we have a clique of size K, then it must include precisely one literal from each clause. Moreover, we can satisfy all the literals in the clique simultaneously, since a clause is a disjunction of the literals and we can satisfy one of the literals. Moreover, this satisfies B since B is the conjunction of the clauses, and we satisfy all the clauses.

We will illustrate this with an example. Assume that we are given the following CNF.

$$(x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_3 \lor \neg x_4) \land (\neg x_2 \lor x_4) \land (x_2 \lor \neg x_3 \lor x_4)$$

There are 4 vertices, so $K = 4$. The graph G consists of (l, C), where l is a literal in clause C. We can join them if they are from different clauses and can be satisfied together. So, this gives us the following graph.



Highlighted in red is the clique of size 4. So, setting $x_1 = \texttt{true}, \neg x_2 = \texttt{true}, x_3 = \texttt{true}, x_4 = \texttt{true}$ satisfies B.

Problem Restrictions

A restriction of a problem consists of a subset of the instances of the original problem. If a restriction of a given decision problem Π is NP-complete, then so is Π. However, given an NP-complete problem Π, a restriction of Π might be NP-complete.

For example, a clique restricted to cubic graphs is in P. By definition, a cubic graph is a graph in which every vertex belongs to 3 edges. A largest clique has size at most 4, so exhaustive search is $O(n^4)$. On the other hand, if we restrict graph colouring to cubic graphs, the problem remains NP-complete.

The K-colouring problem is a restriction of GCP for a fixed number K of colours. A 2-colouring problem is in P, since it reduces to checking the graph is bipartile. On the other hand, 3-colouring is NP-complete.

Similarly, K-SAT is a restriction of SAT in which every clause contains exactly K literals. 2-SAT is in P, but 3-SAT is NP-complete. We can prove this by showing that SAT can be reduced to 3-SAT. So, assume we are given an instance B of SAT. We will construct an instance B' of 3-SAT. For each clause C of B, we construct a number of clauses of B', as shown below:

- If $C = l_1$, we introduce 2 addition variables x_1 and x_2 and add the clauses: $(l_1 \vee x_1 \vee x_2), (l_1 \vee x_1 \vee \neg x_2), (l_1 \vee \neg x_1 \vee x_2)$ and $(l_1 \vee \neg x_1 \vee \neg x_2)$ to B'. The conjunction of these clauses can be satisfied if and only if l_1 is `true`.
- If $C = (l_1 \vee l_2)$, then we introduce a variable y and add the clauses $(l_1 \vee l_2 \vee \neg y)$ and $(l_1 \vee l_2 \vee \neg y)$. The conjunction of these clauses can be satisfied if and only if $l_1 \vee l_2 = $ `true`.
- If $C = (l_1 \vee l_2 \vee l_3)$, then we can just add the clause to B'.
- If $C = (l_1 \vee \cdots \vee l_k)$ (for $k > 3$), then we introduce $k-3$ additional variables $z_1, \ldots, z_{k-3}$ and add the clauses: $(l_1 \vee l_2 \vee z_1), (\neg z_1 \vee l_3 \vee z_2), (\neg z_2 \vee l_4 \vee z_3), \ldots, (\neg z_{k-4} \vee l_{k-2} \vee z_{k-3}), (\neg z_{k-3} \vee l_{k-1} \vee l_k)$. Again, all the clauses hold if and only if C holds.

If we have a problem that is NP-complete, we can try restricting a problem if only a particular class of problems are of interest, given that the restriction is in P. We can seek an exponential-time algorithm that improves on exhaustive search. Moreover, for an optimisation problem, we could settle for an approximation algorithm that runs in polynomial time, especially if it gives a provably good result (within some factor of the optimal solution). For a decision problem, we can settle for a probabilistic algorithm that gives the correct answer with high probability.