Brought to you by:

ASTRA

FUNDAMENTALS OF COMPUTER SCIENCE II PARTIAL

1° BEMACS

Written by
Glulia Pinciroli

2023-2024 Edition

Find out more at:

astrabocconi.it

COMPLEXITY THEORY

We want to classify optimization problems into **complexity classes** based on how difficult it is to solve them.
We want to put some order in the computational challenges that we want to solve.
Depending on how the optimization problems are structured, they may be easy to solve or hard to solve, in which case you have to accept a sub-optimal solution.

Halting problem
Can we decide whether a program P halts on input i by inspection rather than RUNNING $P(i)$?
Is there a program $halt(P, i)$ (which takes as input a code P and the input to that code (i)), such that

$$halt(P,i) = \begin{cases} \text{true if } P(i) \text{ halts} \\ \text{false otherwise} \end{cases}$$

Supposing that the program *halt* exists, we can use it to write a function "trouble" that takes as an input the string "s".

```
function trouble(string s)
    if halt(s,s)        → if halt outputs true, it loops forever
        loop forever
    else                → if halt outputs false, it returns true
        return true
```

What happens if we run trouble(trouble)?

1. **Case 1:** Program trouble halts on input trouble. Since we assume that the program *halt* is correct, *halt* returns true on input trouble,trouble. Hence, program trouble loops forever on input trouble → **contradiction.**
2. **Case 2:** Program trouble loops forever on input trouble. Hence, by the correctness of the *halt program*, *halt* returns false on input trouble,trouble. Hence, program trouble halts at input trouble → **contradiction.**

We are building a function such that, by construction, if it is going to halt, it is also going to loop forever → syntactically it is a perfectly correct program, but it leads to contradiction. The result of this contradiction is that the program *Halt* cannot exist.

The problem of halting cannot be solved → not all problems can be solved.

COMPLEXITY (problem) = AMOUNT of RESOURCES consumed by SOLUTION

- amount → # elementary operations. Asymptotic scaling. Worst case bound.
- resources → Time, space, and energy.
- solution → best algorithm.

Other 2 problems:

- Multiplication → it can be done in a polynomial number of steps $O(n^2)$ → easy to solve. Best known algorithm (FFT)=$O(n \, log \, n \, loglog \, n)$

- Factorization $\rightarrow$ finding the 2 prime factors that give a certain number $\rightarrow$ Naive: $O(n^2 \cdot 2^{2/n}) \rightarrow$ hard to solve. Best known algorithm (GNFS)=$O(exp\ ((\frac{64}{9}n)^{1/3}(log\ n)^{2/3}))$.

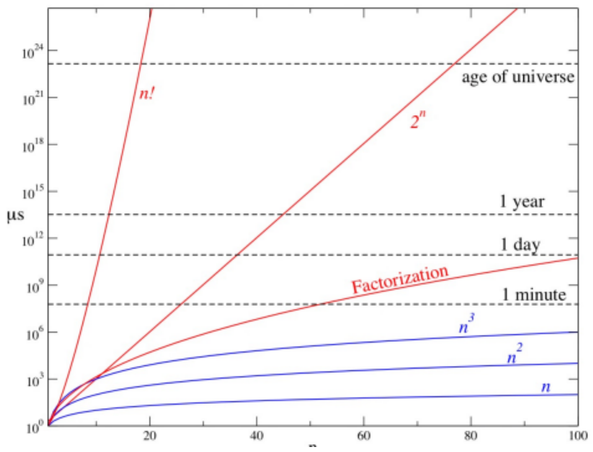
Can we classify these types of behaviors?

Time complexity of an algorithm

Worst case time complexity $T(n) \rightarrow$ running time that the algorithm takes to solve the worst possible instance of a problem.

$$T(n) = \max_{|x|=n} t(x)$$

where $t(x)$ is the algorithm's running time for input data x (in arbitrary units), and the maximum is taken over all problem instances of size n.
The worst-case time is an upper bound for the observable running time.



Example of a NON-TRIVIAL PROBLEM that can be solved in POLYNOMIAL TIME

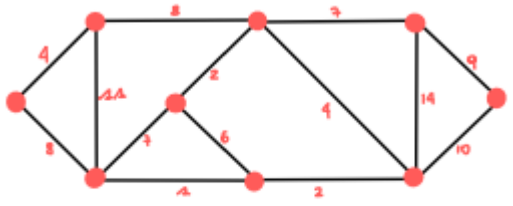
MINIMUM SPANNING TREE (MST)

Def. A spanning tree of G is a subgraph T that is:

- connected
- acyclic
- includes all the vertices $\rightarrow$ spanning graph.

INPUT: connected, undirected graph G with positive edge weights
OUTPUT: a minimum weight spanning tree
Find a subgraph that connects all vertices (spanning) in the graph (for example a spanning subgraph) and whose edges have minimum total weight (to each edge we associate a weight $w_{i,j}$ which is a positive quantity).
The weight is some cost associated to an edge. Given a graph that has all these weights, I want to find a subgraph that connects all the vertices such that the sum of the weights is minimized.

$$\min_{T} \sum_{e \in T} W_e$$



Observations:

- the subgraph is going to be a tree
- the subgraph is not going to have a cycle $\rightarrow$ having loops add cost without adding reachability $\rightarrow$ has to be a tree.

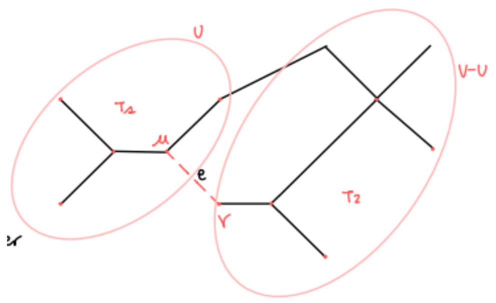
Cut property

Def. A cut in a graph is a partition of its vertices into two (non-empty) sets. Cut = edges that connect the two sets.
Def. A crossing edge connects a vertex in one set with a vertex in the other.
Theorem (the cut property)
Given any cut, the crossing edge of minimal weight is in the MST.

Lemma: Let $U \subset V$ be a subset of the vertices of $G = (V, E)$, and let e be the edge with the smallest weight of all edges connecting U and $V - U$. Then e is part of the MST.
We imagine seperating the graph in two parts (U and V). Among all the edges that connect the two parts, the one that has minimum weight must belong to this minimum spanning tree.
Proof (by contradiction)
Suppose T is an MST not containing e. Let $e = (u, v)$, with $u \in U$ and $v \in V - u$. Then, because T is a spanning tree, it contains a unique path from u to v that together with e forms a cycle in G.
This path must include another edge f connecting U and $V - U$. Now $T + e - f$ is another spanning tree with less total weight than $T \rightarrow T$ was not an MST.

Other way:
Suppose min-weight crossing edge e is not in the MST.

- adding e to the MST creates a cycle
- some other edge f in the cycle must be a crossing edge
- removing f and adding e is also a spanning tree
- since weight of e is less than the weight of f, that spanning tree has lower weight
- contradiction!



This lemma lets an MST grow edge by edge. For all the starting points you obtain the same graph.

Greedy MST algorithm demo

1. Start with all edges colored gray
2. Find cut with no black crossing edges; color its min-weight edge black
3. Repeat until $v - 1$ edges are colored black

Efficient design
Proposition: the greedy algorithm computes the MST.
Efficient implementations: key steps:

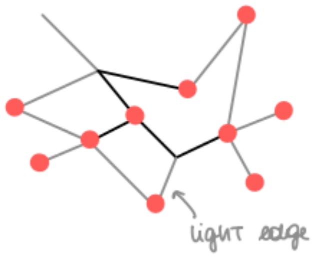
1. How do we choose the cut?
2. How do we find the min-weight edge?

Two examples:

Prim's algorithm

PRIM(G)
Input: weighted graph $G(V, E)$

```
Output: minimum spanning tree
    begin
    Let $T$ be a single vertex from $G$
        while $T$ has less than $n$ vertices → continue until you
    reach all vertices
    find the minimum edge connecting $T$ to $G - T$
    add it to $T$
    end
end
```



Computational cost: $T(n) = O(n^2 \, log \, n)$. Solvable in Polynomial time, class P.

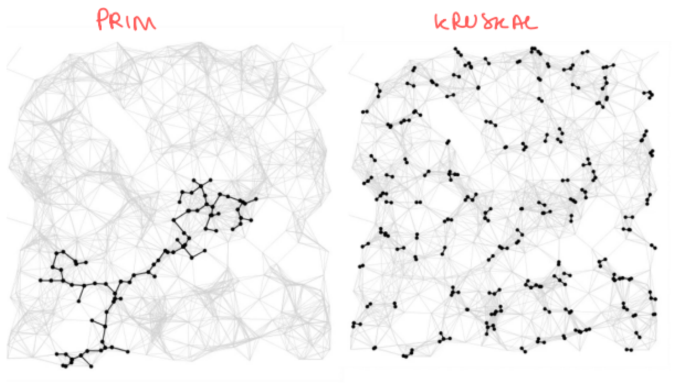
1. Start with vertex 0 and greedly grow tree T
2. Add to T the min weight edge with exactly one endpoint in T
3. Repeat until $v - 1$ edges are left.

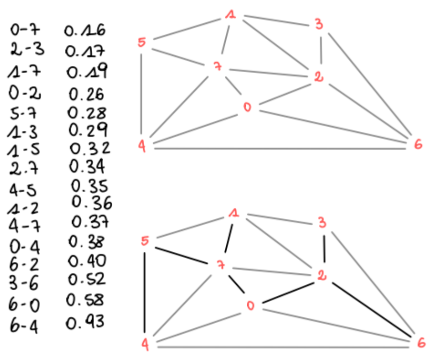
By using just local information we obtain the best possible spanning tree (global optimum). To be computationally efficient: to obtain an optimal result by local moves.

Kruskal's algorithm

Consider edges in ascending order of weight.
Add the next edge to tree T unless doing so would create a cycle.
Suppose you have already chosen some edges. Among the remaining edges, you are going to have an edge of minimum weight (that does not have to be connected to the previous one), but for sure it should not create a cycle. You would eventually end up with a minimum spanning tree.
At the beginning, to put the edges in ascending order of weight, you have to run a sorting algorithm (order $n \, log \, n$ steps)

Prim vs Kruskal: visualization





0-7	0.16
2-3	0.17
1-7	0.19
0-2	0.26
5-7	0.28
1-3	0.29
1-5	0.32
2.7	0.34
4-5	0.35
1-2	0.36
4-7	0.37
0-4	0.38
6-2	0.40
3-6	0.52
6-0	0.58
6-4	0.93

TRAVELING SALESMAN PROBLEM (TSP)

We have a graph that contains only the edges that I will select (it does not have all the possible edges). Each edge has a weight $d_{i,j}$ (positive).
We want to find a cycle of minimum weight that connects all vertices. The cycle can be unique (TSP) or there could be multiple cycles.
For the TSP: the only way to find this path is by exploring all possible alternatives → factorial number , more than exponential. It can be solved, but there is no algorithm for it.
Given an $n \times n$ distance matrix with elements $d_{i,j} \geq 0$, find a cyclic permutation $\pi: [1 \dots n] \to [1 \dots n]$ that minimizes

$$c_n(\pi) = \sum_{i=1}^{n} \quad d_i \pi(i)$$

DECISION PROBLEMS: problems whose solution is either "yes" or "no"
We can turn any optimization problem into a decision problem by adding a bound B to the instance.

MST (decision): Given a weighted graph $G = (V, E)$ with nonnegative weights and a number $B \geq 0$, does G contain a spanning tree T with total weight $\leq B$?
Instead of looking for a minimum weight, we ask ourselves if there exists a tree whose weight is below a given level. The answer is either "yes" or "no".
Running the algorithm multiple times and changing B we find the minimum (equivalent to the optimization problem).
If you can answer " yes" or "no" in polynomial time, we can also find the optimal result in polynomial time.

TSP (decision): Given an $n \times n$ distance matrix with elements $d_{i,j} \geq 0$ and a number $B \geq 0$, is there a tour π with length $\leq B$?

Examples of problems that can be solved easily → called P because they can be solved in polynomial time (solved by an algorithm whose running time is bounded by a polynomial).
Example: KÖNISBERG PROBLEM

THEOREM (Euler 1736)

A connected graph $G(V, E)$ contains an Eulerian circuit if and only if the degree of every vertex is even.
A cycle that traverses each edge of a graph exactly once is called an Eulerian cycle (you have to go back to the initial point).
This is also a polynomial algorithm. You just need to check the degree of the nodes.
It is an $O(n^2)$ algorithm → in the worst case the nodes have degree of $O(n)$.

Intractable itineraries
A cycle that traverses each vertex of a graph exactly once is called a HAMILTONIAN CYCLE.
No insight available. Exhaustive search seems to be unavoidable.
This problem does not belong to the class P

SATISFIABILITY PROBLEM

A boolean variable x can take the value 0 (false) or 1 (true). Boolean variables can be combined in CLAUSES using the boolean operators → by using these 3 operators you can express any boolean function:

- NOT $\cdot$ (negation) → the clause $\bar{x}$ is true (=1) if and only if x is false (=0)
- AND $\wedge$ (conjunction) → the clause $x_1 \wedge x_2$ is true $(x_1 \wedge x_2 = 1)$ if and only if both variables are true: $x_1 = 1$ and $x_2 = 1$.
- OR $\vee$ (disjunction) → the clause $x_1 \vee x_2$ is true $(x_1 \vee x_2 = 1)$ if and only if at least one of the variables is true: $x_1 = 1$ or $x_2 = 1$.

A variable x or its negation is called a LITERAL.

Different clauses can combine to yield complex boolean formulas (consider some set of variables written as conjunctions(and operator) or disjunction (or operator)) such as

$$F_1(x_1, x_2, x_3) = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_3)$$

- $(x_1 \vee \bar{x}_2 \vee x_3)$ → clause: just taking the "or" of some variables.

A formula F is called SATISFIABLE if there is at least one assignment of the variables such that the formula is true (F=1).

The formula evaluates to 1 if and only if all the clauses are satisfied simultaneously (it exists a configuration of the x variables such that all the clauses are satisfied at the same time) → satisfy a list of constraints (clauses) simultaneously.

The existence of an assignment such that F=1 depends typically on the ratio between the number of clauses and the number of variables. It is hard to solve if we have a large number of constraints and a few number of variables .

BOOLEAN FORMULA IN CONJUNCTIVE FORM (CNF)

A set of clauses C_k combined exclusively with the AND operator: $F = C_1 \wedge C_2 \wedge \ldots \wedge C_m$
Literals in each clause are combined exclusively with the OR operator.

Satisfiability (SAT)

Given disjunctive clauses $C_1, C_2, \ldots, C_m$ of literals, where a literal is a variable or negated variable from the set $\{x_1, x_2, \ldots, x_n\}$, is there an assignment of variables that satisfies all clauses simultaneously? (F=1)

K-sat

Given disjunctive clauses $C_1, C_2, \ldots, C_m$ of k literals each (all the clauses contain the same number of variables), where a literal is a variable or negated variable from the set $\{x_1, x_2, \ldots, x_n\}$, is there an assignment of variables that satisfies all clauses simultaneously?

- If $k \leq 2$, there exists an algorithm that can solve this problem in polynomial time.
- If $k \geq 3$, we don't know if such an algorithm exists, but we cannot prove that it exists.

We cannot use this to construct complexity classes, because we cannot prove that an algorithm exists or does not exist. In order to classify problems, we need a mathematical construction.

Idea: given a potential solution, we can check if it satisfies the formula or not by plugging it into the equation. This can be done in polynomial time (linear number of operations).

We first define the classes of problems such that a solution can be verified in polynomial time.

NON-DETERMINISTIC ALGORITHMS (do not exist)

The definition of the second complexity class involves the concept of a NON-DETERMINISTIC ALGORITHM, which is like an ordinary algorithm, except that it might use one additional, very powerful instruction :

goto label 1, label 2

(when it has to make a decision, it takes both decisions)
The computation branches like a tree into several parallel computations that potentially can grow as an exponential function of the time elapsed.
A non-deterministic algorithm can follow all the branches of a search tree by multiplying the number of processors.
It is a mathematical construction, it does not exist, the computational resources (e.g . memory) grow exponentially.
Checking a potential solution is equivalent to computing just one branch of the tree.
Saying that a solution can be checked easily is like saying that the problem can be solved by a non-deterministic algorithm $\rightarrow$ as far as a solution can be checked in polynomial time, the problem can also be solved by a non-deterministic algorithm.

COMPLEXITY CLASSES

DEFINITION 1: A decision problem is an element of the CLASS P if and only if a polynomial time algorithm can solve it.
There exists an algorithm that can solve all instances of a problem (also in the worst case) in polynomial time
Examples : Eulerian circuit , 2- coloring , MST(D)

DEFINITION 2: A decision problem is in the class NP if and only if a nondeterministic algorithm can solve it in polynomial time.
Set of problems for which a solution can be checked in polynomial time.

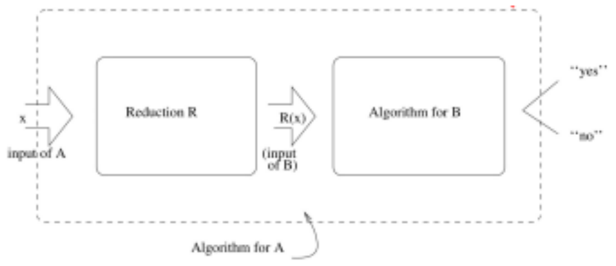
SATISFIABILITY (F) → CLASS NP

INPUT: boolean formula $F(x_1, \dots, x_2)$
OUTPUT: 'yes' if F is satisfiable, 'no' otherwise. Begin

```
        for i=1 to n
        goto both label 1, label 2
                label 1: $x_i$ = true; continue
                label 2: $x_i$ = false; continue and
        if $F(x_1, \dots, x_n)$ = true then return 'yes' else return 'no'
end
```

For each variable it checks the two values and then returns 'yes' and 'no'.
EQUIVALENT DEFINITION 2: : A decision problem is an element of the class NP if and only if for every yes' instance of P there exists a SUCCINCT CERTIFICATE (non-exponential proof) that can be verified in polynomial time. (i.e. we can check that a solution is truly a solution in polynomial time).

POLYNOMIAL REDUCTION OF A AND B



Definition: we say a problem A is polynomially reducible to a problem B and we write A≤B if a polynomial algorithm exists for A provided there is a polynomial algorithm for B.
A≤B means A is included in B → if you can solve B you can solve A.

Suppose we have an algorithm that can solve problem B ; If I can rewrite problem A as a particular instance of problem B, then I can apply the algorithm to it . Transforming problem A in this way should take a time which is polynomially bounded, otherwise it would be useless.
Having a polynomial reduction does not alter the final polynomial nature of the solution of the problem.
Example → Hamiltonian cycle is polynomially reducible to TSP(D) by setting distances to 1 and writing: Hamiltonian cycle ≤ TSP(D).
If you have an algorithm for TSP, you can use it to solve the Hamiltonian cycle just by setting to 1 the weight of all the edges.
By using the idea of reelection we can establish an equivalence class between problems that hide the key of the solvability of all the others. If you solve one of them , you can solve all the others.

NP-COMPLETE PROBLEMS (not all empty set)

Subclass such that all problems in NP and P can be reduced polynomically to problems in the NP-complete class.
Some other polynomial reductions:
- SAT ≤ 3-SAT
- 3-COLORING ≤ 3-SAT
- 3-COLORING ≤ HAMILTONIAN CYCLE

If you solve 1 problem in this class you can solve everything else. The problems in this class are equivalent: they can be reduced into one another.
No polynomial algorithm that can solve NP-complete problems has been found yet.
Polynomial reducibility is transitive: $P_1 \leq P_2$ and $P_2 \leq P_3$. (both mapping takes polynomial time → overall polynomial mapping from P_1 to P_2).
From transitivity it follows that each SAT, 3- SAT ,3- Coloring , and Hamiltonian cycles reduces toTSP(D) → a polynomial algorithm for TSP(D) implies a polynomial algorithm for all these problems.

COOK'S THEOREM (1971): All problems in NP are polynomially reducible to SAT

(SAT is NP complete → not an empty set)

$$\forall P \in NP: P \leq SAT$$

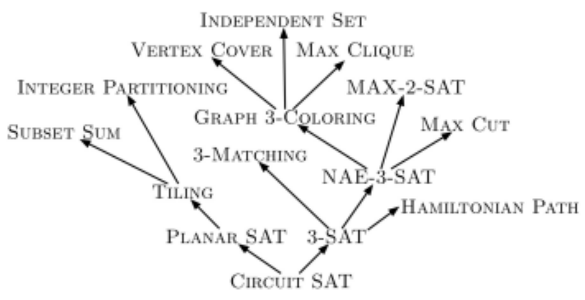
Every problem P in NP can be reduced, and therefore represented, as an instance of SAT. P is always less (in terms of difficulty) than SAT.
IMPORTANT IMPLICATION:
- No problem in NP is harder than SAT, or SAT is among the hardest problems in NP.

- A polynomial algorithm for SAT would imply a polynomial algorithm for every problem in NP → it would imply P=NP.

DEFINITION OF NP-COMPLETENESS (property of belonging to the class NP-complete):

A problem P is called NP complete if $P \in NP$ and $Q \in P$ for all $Q \in NP$.
The class of NP-complete problems collects the hardest problems in NP.
If you show that an NP-complete P problem can be reduced to another problem Q, then Q is also an NP-complete (equivalent class) → you have to reduce a problem which is NP-complete onto another problem to declare that the latter also belongs to the class NP-complete.
The other way around is obvious and useless.
By definition, all problems that are NP-complete can be mapped by all the other problems in NP.
If you are able to map an NP-complete problem to a problem in NP, then the NP problem is moved in the NP-complete class, because you have reduced a problem that is NP-complete onto it, which means that if you can solve the NP problem you can solve the NP-complete problem → you can solve all the other in the class (NP-complete) → the problem should go to the NP-complete class.

NP-COMPLETE FAMILY TREE



Cook was able to show that SAT was NP-complete.
By reducing it to 3-SAT, he was able to show that 3-SAT was also NP-complete and so on so forth → more than 3000 NP-complete problems are known.

COOK-LEVIN THEOREM

Theorem that showed that SAT was an NP-complete problem → they were able to show that any problem that can be checked in polynomial time can be mapped onto SAT.
The Boolean satisfiability problem is NP -complete → any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the problem of determining whether a Boolean formula is satisfiable → for any checkable proof, you can show that there is a SAT formula that does that.

COMPOSING REDUCTIONS

Polynomial time reductions are clearly closed under composition.
By definition: if $L_1 \leq pL_2$ and $L_2 \in P$, then $L_1 \in P$
So, if $L_1 \leq pL_2$ and $L_2 \leq pL_3$, then we also have $L_1 \leq pL_3$.
The usefulness of reduction is that they allow us to establish the relative complexity of problems, even when we cannot prove absolute lower bounds → you can say whether 2

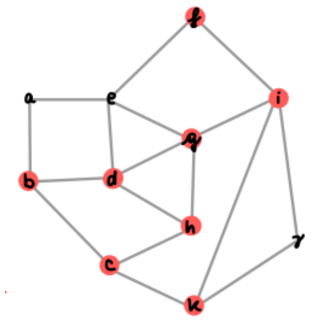
problems are equivalent up to a polynomial reduction (polynomial transformation = transformation in a polynomial number of steps).
If we show, for some problem A in NP that $SAT \leq pA$, it follows that A is also NP-complete (because it means that A is even more general than SAT).

INDEPENDENT SET

Given a graph, a subset $G = (V, E)$ of the vertices is said to be an INDEPENDENT SET, if there are no edges (u, v) for $u, v \in X$ (there are no edges that connect the two nodes).

The natural algorithmic problem is, given a graph, to find the largest independent set.
To turn this optimization problem into a decision problem, we define IND as:

the set of pairs(G, K), where G is a graph, and K is an integer, such that G contains an independent set with K or more vertices .



Given a graph G and an integer number K, you ask whether it exists an independent set with K or more vertices → answer is YES or NO.
To check a solution: take the node and check if there are some edges.
Since the size of the problem is of the order of the total number of edges, then you have to explore something of the order of the size of the system → Polynomial (linear in the number of edges) → the independent set IND belongs to the class NP.
In the example: {a,d,i,c} independent set, {a,c,j,f,g} maximal independent set.

IND is in NP. We want to show it is NP-complete → any instance of 3 SAT can be written as a particular independent set problem on a particular graph → find a reduction from 3 SAT to independent set.

Reduction

We can construct a reduction from 3 SAT to IND.
A Boolean expression ϕ in 3CNF (conjunctive normal form) with m clauses is mapped by the reduction to the pair (G, M), where G is the graph obtained from ϕ as follows:

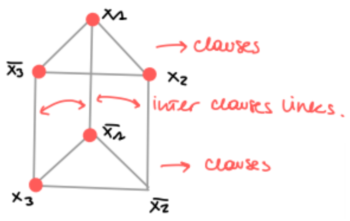
G contains m triangles, one for each clause of ϕ, with each node representing one of the literals (either a variable or its negation) in the clause.

Additionally, there is an edge between two nodes in different triangles, if they represent literals where one is the negation of the other → we connect variables that have the same name but one is negated and the other is not.

Example

$(x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_3 \vee \bar{x}_2 \vee \bar{x}_1)$

- For each clause I build a graph (triangles)
- I connect the literals that are opposite
- If you find an independent set of size m (number of clauses), then from the independent set I can read a solution to the SAT problem.
- Given that we put a link between a variable and its negation, we will never put them in the same independent set → for the

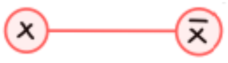


independent set we are only going to choose variables that are different.
- Check if the clauses are satisfied
- The particular constructions that allow you to form the equivalent graph are called GADGETS .

Vertex cover: Given a graph, we want to find a set of nodes such that all the edges are seen by the nodes.
Vertex cover: set of vertices that you choose in order to cover all the edges.
Optimization problem: find the vertex cover with the minimum number of vertices.
This is an NP problem, because given a vertex cover, we just need to check whether all edges have endpoints in this set of vertices.

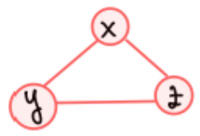
3-SAT → VERTEX COVER

Suppose our boolean formula for 3- SAT , ϕ , contains m variables and l clauses.
For our vertex cover we will use $k = m + 2l \rightarrow$ we can construct a graph such that if on this graph I find a vertex cover of size $m + 2l$, then from the vertex cover I can read out the solution to the original SAT problem.
For each variable in ϕ we will use the following variable gadget:
The variable x in a ϕ for 3-SAT is replaced by 2 nodes in a graph.
One node represents the literal x, while the other represents its negation $\bar{x}$.



1. Create a graph in which, for each variable, I create a pair $x - \bar{x}$ connected by an edge
2. for each clause I will create a triangle
3. connect the literals in the variable gadget.

For each clause in ϕ we will use the clause gadget:

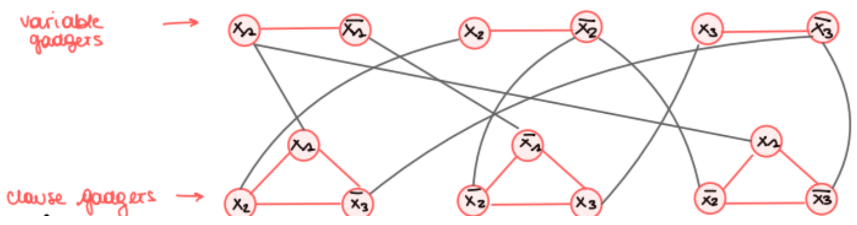


The 3-SAT clause is represented as a clique of 3 nodes, where each node represents a literal in the clause. The clause in the example is: $(x \vee y \vee z)$
Connect the literals in the variable gadgets to the matching (that are alike) literals in the clause gadgets with an edge.

Example

$$\phi = (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$$

$m = 3$
$l = 3$
vertex set $k = 3 + 2 \cdot 3 = 9$
At this point, the reduction is complete. We still have to prove that this reduction is correct, meaning that a vertex cover of size $k = m + 2l$ exists if and only if the given ϕ is satisfiable.



Theorem: if you can find a vertex cover of size $k = m + 2l$, then the 3- SAT formula is satisfiable.
How to check if it is a solution to the SAT problem? Look at the variable gadget and the assignment in the variable gadget.

For sure they have to be covered → there is a link for each variable → at least one of the two nodes has to be covered.
At least two nodes in the triangle need to be covered, otherwise the triangle would not be covered.
If I have l clauses, I must have $2l$ covers that are on the triangles → to reach $m + 2l$ covers I must have just one cover for each variable gadget.
We read the solution from the variable gadget and we check if it is correct.
We conclude that vertex cover is NP complete.

VERTEX-COVER → CLIQUE

Finding a vertex-cover in a graph can be mapped into the problem of finding a maximum clique in another graph → maximum clique is NP-complete.
Problem: find the maximum clique in a graph.
Clique → subset of all the vertices such that every 2 distinct vertices are adjacent.

Proving that P is different from NP is still an open problem
We have 3 classes: P, NP, and NP-complete based on the notion of equivalence between problems: problems are equivalent between the class NP-complete and any problem in NP can be mapped onto them → if you solve anything inside the NP-complete class in polynomial time then everything would become polynomial.
This is a rigorous construction for which only the class P is characterized by the fact that you are able to prove that a problem can be solved in polynomial time.
For the rest the definition does not tell you how long it takes to find a solution, it just tells you how long it takes to check it.
The class NP-complete is just another class, that is within the class P, that contains those problems that are representative of all the others through polynomial reduction.

We cannot prove that it is impossible to prove NP-complete problems → in practice there is no evidence that we can solve them.
Everybody in computer science believes that P ≠ NP → this means that the NP-complete problems, in their worst instances, cannot be solved by any algorithm in polynomial time → WORKING ASSUMPTION → no proof for that from a mathematical point of view.
If P would be equal to NP, then technologies like cryptography would disappear.
Examples → Decrypt: "Does an encrypted message M correspond to clear text T?" → this would be solved in polynomial time.
Short-Proof-Existence: "Does theorem T have a proof with less than n lines?" → NP-complete problem → if P=NP mathematics would disappear because we could prove theorems in polynomial time.

SHORTEST PATH

How to find the shortest path between two nodes in a graph (without having to look at the entire graph)
It differs from the minimum spanning tree because the shortest path problem refers to a starting point and to an endpoint.
Spanning tree does not specify them, it just tells you how to find the tree that connects all vertices with a minimum total weight (distance).

Shortest path is a problem that belongs to the class P (only for P problems we have algorithms).
We have to consider weighted graphs → the distances have to be positive but they don't need to satisfy the triangular inequality, because we may want to incorporate in the weights of the graph things other than the distance, like costs, etc.

Shortest path properties
Property 1: A subpath of a shortest path is itself the shortest path
Property 2: There is a tree of shortest paths from a starting vertex (source) to all other vertices
They are fundamental to finding an algorithm.

DIJKSTRA'S ALGORITHM

The distance of a vertex v from a vertex s, $d(v, s)$, is the length of a shortest path between v and s.
Dijkstra's algorithm computes the distances of all the vertices from a given start vertex s → it computes all the shortest path lengths from a given vertex.

Assumptions:
- the graph is connected → the algorithm can only tell you about the shortest path within one connected component → you should apply the algorithm to each connected component.
- the edges are undirected
- the edge weights are non-negative

We grow a "cloud" of vertices, beginning with s (starting point) and eventually covering all the vertices (reaching all the vertices through shortest paths).
We associate to each vertex v a variable $d(v)$ representing the distance of v from s (the source) in the subgraph consisting of the cloud and its adjacent vertices.
At each step:
- we add to the could (set of explored vertices) the vertex u outside the cloud with the smallest distance label, $d(u)$ → as you grow the cloud, you obtain a new vertex in the cloud and the right distance for the vertex, and then you update the label of all the vertices that are connected, giving them the correct distance.
- we update all the labels of the vertices adjacent to u.

Steps
1. Assign to every node a tentative distance value (distance from the source): set it to 0 for our initial (source) node and to infinity for all other nodes. → we do not know it. The only correct information we have is the distance of the source from itself.
2. Set the initial made as current. Mark all the other nodes unvisited. Create a set of all the unvisited nodes called the UNVISITED SET.
3. For the current node, consider all of its unvisited neighbors and calculate their tentative distances. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one.

We update the distance by taking the minimum value between its current value and the one you would obtain by going through the link that connects the node with the one we are sitting on.
For example , if the current node A is marked with a distance of 6, and the edge connecting it with a neighbor B has length 2, then the distance to B (through A) will be $6 + 2 = 8$. If B was previously marked with distance greater than 8 then change it to 8; otherwise, keep the current value.

4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the unvisited set.
   A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new “current node", and go back to step 3.

Differences between the shortest-path algorithms and Prim’s algorithm
In the shortest path algorithm, if you don’t specify the destination node, the algorithm is going to run until it exhausts the whole graph and then it will give you a tree of shortest path.
Differences between this tree and the minimum spanning tree:

- conceptual difference → the shortest path tree depends on the route, while the Prim’s algorithm does not.
- In Prim’s Algorithm, you start with a vertex, not a specific one, and greedly grow the tree. You add to the tree the minimum weight edge with exactly one endpoint in the tree. Then repeat until $v - 1$ → there is no update of distances, everything depends on the weights.
  The distance of the nodes depends on all possible paths that you could have to reach that node and among all these paths select the one of minimal distance (weight).
- Minimum spanning tree → subgraph of minimum total weight
- Shortest path tree → set of nodes that compose the tree of minimum distance with respect to a given source.

Observations about the running time: Dijkstra’s algorithm runs in $O((n + m)\ log\ n)$ time, provided that the graph is represented by the adjacency list structure.
In the running time:

- n= number of vertices
- m = number of edges → you have to look at all the vertices connected to the cloud
- $log\ n$ → you have to look for the minimum distance across all the candidate nodes to be connected to the cloud, so you have to sort them.

The running time can be expressed as $O(m\ log\ n)$ since the graph is connected $(m > n)$.