1 Summary

- Informal definition/description of classes of problems
- Decision Problems and Optimization problems
  - Abstract problem $\rightarrow$ decision problems
  - NPC problems informal
- Encoding of problem instances
- Formal definition of problems
  - Languages
  - Definition of P and NP
- Polynomial time reducible
- NPC definition
- Method for proving NPC
- Vertex cover is NPC

2 Detailed Summary

2.1 Classes of problems

The set of problems in P are those that can be solved in $O(n^k)$ for some constant C. The class of NP are those problems that can be verified in polynomial time. Meaning, if a certificate of a solution is given, we can verify the validity of that certificate. NP-completeness problems concernes itself with decision problems rather than optimization problems. That is, we concern ourselves with not finding the optimal solution to a problem, but we would rather like to get a yes/no answer solution to a problem.

We can more formally define the a "problem" as an abstract problem Q. This is a binary relation between a set of problem *instances*, I, and a set S of solutions to the the problem instance. S is a set, since the solutions might not in general be unique and there might be more than one solution to a problem instance. A problem instance could for instance in the SHORTEST-PATH problem be a tripple (s, t, G) (source, destination and graph), and a solution would be a sequence of vertices for the shortest path. For NPC this definition of an abstract problem is too big, since decision problems concern themselves with decision problems and thereby expect yes/no solutions. Therefore, in the context of NPC problems, we can relax the definition of an abstract definition as a function (instead of a relation) that maps instances I into the solution set $\{0, 1\}$(yes/no answers). With this definition, SHORTEST-PATH optimization problem can be cast into another problem PATH, which a decision problem, whos instances consists of the 4-tuple (s, t, G, k) that asks whether there is a path from s to t in G of length at most k.

2.2 Encodings

Since the problems actually runs on a machine, the program solving the problem will take an encoding of the problem instance. This encoding will typically be a binary string, and a problem with encoding as a binary string is known as a ***concrete problem***. This means, that if an algorithm finds a solution to an instance i of size $n = |i|$ in $O(T(n))$ then it solves the problem in $O(T(n))$ time. This carries over to the definition of polynomial time solvable, since this means that an algorithm solves a problem in $O(n^k)$ if it given an problem instance with encoded size n is solved in $O(T(n)) = O(n^k)$, and this is the class of problems P - problems that can be solved in polynomial time proportional to their encoding size.

With encodings we can therefore map the abstract problems to concrete problems. There are however problems with the encodings, where specific encodings will determine the efficiency(running time) of the algorithm. We shall use the encoding $\langle k \rangle$ to refer to any "concise" encoding of k. This means that whatever encoding is used, it is polynominally time related to the binary representation. This way, we can work in the concise representation and then always afterwards convert to the binary format. ***Therefore, it is assumed that all problem instances are binary strings encoded in a concise way.***

2.2.1 Why encodings matter

We can have an algorithm that takes a single k as input and runs in $\Theta(k)$ time. With unary encoding, meaning $k$ones, this means that the algorithm runs in $O(n)$ for length $n = k$ inputs. If we however use normal binary encoding, then a number of size k requires $\lfloor \log_2 k \rfloor + 1$ bits to encode., and we instead have the input length $n = \lfloor \log_2 k \rfloor + 1$. However, the running time is still $\Theta(k)$, but since we now have a shorter encoding this means that $k = 2^{\log_2 n}$ which gives running time of $\Theta(2^n)$ and we have exponential running time in the size of the input. Since the running time of an abstract problem depends so heavily on the encoding, we can in reality not talk about running time of an abstract problem without specifying the encoding.

2.3 Languages

From language theory, an alphabet Σ is a finite set of symbols. In this setting, $\Sigma = \{0, 1\}$. A language L is any string made from an alphabet. The language over all strings is denoted by Σ^*. Decision problems, Q, are a mapping, or function, from problem instances x to a $\{0, 1\}$ solution. Since we encode problem instances as binary strings $x \in \Sigma^*$, and decision problems are characterized by the yes-problem instances(those where $Q(x) = 1$ with all other input being 0), we can view decision problems as languages over Σ^*

$$L = \{x \in \Sigma^* : Q(x) = 1\}$$

With the definition of decision problems as languages, we can view the decision problem PATH as the language

$$\begin{aligned} \text{PATH} = \{\langle s,t,G,k\rangle \in \Sigma^* : & G = (V,E) \text{is an undirected graph and} && (1)\\ & s,t \in V && (2)\\ & k \text{is and integer} && (3)\\ & \text{there exists a path from } s \rightsquigarrow t \text{ of at most } k \text{ edges}\} && (4) \end{aligned}$$

The language accepted by an algorithm $A$is the set of binary strings $x \in \Sigma^*$for which

$$L = \{x \in \Sigma^* : A(x) = 1\}$$

An algorithm rejcts a string if $A(x) = 0$.

A language L is decided by A if every string in L is accepted by A and all other strings are rejected. This leads to the definition of accepted and decided in polynomial time. What is important to note is perhaps that if an algorithm only accepts a language, then it can just halt for invalid strings, which is not allowed if it decides a language.

By this we can formally define the complexity class P as

$$\begin{aligned} P = \{L \subseteq \{0,1\}^* : & \text{ there exists an algorithm } A\text{that} && (5)\\ & \text{decides } L\text{in polynomial time} && (6) \end{aligned}$$

meaning all the languages that can be decided in polynomial time. From this follows a theorem saying that the class of problems P are the languages that are accepted by polynomial time algorithms.

Theorem:

$$P = \{L : L \text{ is accepted by a polynomial time algorithm}\}$$

Proof:

We can prove this by showing that P is both an inclusive subset, $\subseteq$, and inclusive superset, $\supseteq$, of the above set. It is trivial to show that P is a subset of this, since by the definition of P being those languages that are decided, and since the languages that are decided in polynomial time are a subset of those that are accepted in polynomial time. This is true, since the definition of deciding meant that it accepts L and rejects all other inputs.

Therefore, we need to show that the above set is a subset as well, meaning that if a language is accepted by a polynomial time algorithm A then it also decided by another polynomial time algorithm, say A'. We can prove this by simulation, and we let A' by an algorithm that runs A. Since we know that A accepts a language in polynomial time, it accepts L in at most cn^k steps. Therfore, we can let A' be the algorithm that simulates A for cn^ksteps. Thereafter, it can check if A accepted the string, in which case A' returns 1, otherwise A' will return 0. Simulating A induces an overhead that is no more than a polynomial factor, and therefore A' is still a polynomial time algorithm that can decide L, and therefore deciding a language is a superset of accepting a language

We have now shown the set is a subset and superset of the definition, meaning it also belongs to the complexity class P.

2.4 Polynomial time verification

A ***verification*** algorithm is a two-argument algorithm $A(x, y)$, where x is the problem instance y is a certificate. We say that A verifies x if there exists a certificate y such that $A(x, y) = 1$. The language verified by $A$is

$$L = \{x = \{0,1\}^* : \text{there exists } y \in \{0,1\}^* \text{such that } A(x, y) = 1$$

An aglorithm verifies a language if for any $x \in L$ there exists a y that can be used to prove that $x \in L$. There must however be no y that can be used that x is in L when it is not the case.

2.5 The class NP

We can use the notion of a verification algorithm to define the complexity class of NP.

Definition

A language $L \in NP$ if and only if there exists a polynomial time verification algorithm A and constant c such that

$$L = \{x \in \{0,1\}^* : \begin{array}{c} \text{there is a } y \in \{0,1\}^* \text{with} \\ |y| = O(|x|^c) \text{ such that } A(x, y) = 1 \end{array}$$

By this definition, then $P \subseteq NP$ since we can always design a verification algorithm that ignores the certificate and just runs the algorithm with instance x.

2.6 Polynomial reducible

A language L_1 is polynomial reducible to another language L_2

$$L_1 \leq_p L_2$$

if there exists a polynomial time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$we get that

$$x \in L_1 \iff f(x) \in L_2$$

This means that the reduction function f provides a polynomial time mapping, such that every $x \in L_1$will be in $f(x) \in L_2$ and if $x \notin L_1$will not be in $f(x) \notin L_2$. The function f maps instances x of the decision problem identified by L_2 into instances $f(x)$ of the decision problem identified by language L_2 and solving a decision problem in either L_2/L_1 provides the solution to the decision problem in the other language. We call f a ***reduction function*** and the algorithm, F, that computes f is called the ***reduction algorithm***, since the decision problem of solving problem instance for the language L_1 is reducible to solving $f(x)$ in L_2. *In a sense, solving $L_1$is no harder than solving L_2since we can always convert the problem instance to a problem in L_2.*

We can actually say that

$$L_1 \leq_p L_2 \wedge L_2 \in L_1 \implies L_1 \in P$$

since we can use f to convert a problem in L_1 to a problem in L_2 in polynomial time, and solve the decision problem in L_2 in polynomial time.

2.7 NPC

Definition: A language $L$is NP-complete if

1. $L \in NP$
2. L'$\leq_p L$for every $L' \in NP$

This states that NP are the hardest problemst in NP, since they to a polynomial factor are at least as hard as all other NP problems. The second problem is known as NP-hardness.

2.8 Proving NPC

The general formula for showing NPC for a new language L, is to use reduction from an already known NPC language $L' \in$ NPC.

1. Prove that $L \in$ NP
2. Select an known NPC language L'
3. Show that $L' \leq_p L$. That is, show that there is a polynomial time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$we have $x \in L' \iff f(x) \in L$. That is, show that L is NP-hard.

2.9 Satisfiability

A problem instance for SAT is a boolean formula ϕ consisting of

1. n boolean variables x_i
2. m boolean connectives - this is any unary of binary boolean function
3. parenthesis - this is used to specify the ordering of operators

A satisfying assignment is an assignment of the n variables that causes ϕ to be 1. We also say that ϕ is satisfiable if such an assignment exists. The language of the problem SAT is defined as

$$\text{SAT} = \{\langle\phi\rangle : \phi \text{ is a satisfiable boolean formula}\}$$

and is the language of all boolean formulas that are satisfiable.

2.9.1 Showing the language is in NP

Showing that L is in NP means that we have to show that there is a polynomial time verification algorithm A that given a certificate and problem instance can verify if the instance x is in L. As the certificate we pick the assignment of the n variables. The verification has to prove an instance x is in L given the certificate. This means that when y is a an assignment of variables we can simply run the boolean formula described by x. If 1 is returned the assignment is satisfiable. Otherwise it is not. The running time of A is the running time of the formula. The running time must be polynomial in the size of the formula - number of connectives and boolean variables.

2.9.2 Showing NP-hardness

Showing NP-hardness means we have to take circuit-sat and reduce it into the SAT problem.

If we have a circuit-satisfiable circuit C then we start by assigning a variable x_i to each wire of the circuit. Then, we can view each gate of C as a mapping from its input wire to its output wire (there will always be one output wire, which might then later fan out). Then we simply construct a subformula ϕ_i for each gate in the circuit doing exactly what the gate did. For instance, if we have an AND gate with three input, we can express the output as $x_4 \iff (x_1 \wedge x_2 \wedge x_3)$. We can construct a subformula for each gate. To produce the formula ϕ, we AND the output wire x_m of C with all the subformulas ANDed together $\phi = x_m \wedge \phi_1 \wedge \phi_2 \wedge, ..., \wedge\phi_k$. Since we get a satisfying circuit when the output wire is 1, then this formula simply says that the output wire should be what we expect and that each gate should do what it is intended to do. Therefore, whenver C is satisfiable, then ϕ is also satisfiable. We have now shown f and F runs in polynomial time in the number of gates and wires, since we just assigned a variable for each wire, and then made a subformula for each gate.

2.10 Showing 3CNF is NPC

2.11 The Clique problem

A **clique** in an undirected graph $G = (V, E)$ is a set of vertices $V' \subseteq V$, such that each pair of vertices in V' are connected by an endge in E. That is, for each pair of distinct $u, v \in V'$, then $(u, v) \in E$ is an edge in the original graph.

The size of a clique is the number of vertices in a clique. **The clique problem** is an optimization problem of finding the largest clique in a graph. The decision problem can be formalized as

$$\text{CLIQUE} = \{\langle G, k\rangle : G\text{is a graph containing a clique of size } k$$

The naive solution would be to consider each k subset of vertices and check if it is a clique. This would be ${}_{|V|}C_k$which is $\Omega(k^2(\binom{|V|}{k}))$. The problem is that k will often be near $|V|/2$ and it is therefore not polynomial.

2.11.1 Clique is in NP

The verification algorithm will take the subset of vertices $V' \subseteq V$ as a certificate. It will then check that $|V'| = k$ and that each pair of vertices in V' has an edge in E. This can be done in polynomial time in V' and therefore also in V.

2.11.2 Clique is NP-hard

It can be shown that CLIQUE is NP-hard by the reduction

$$\text{3-CNF-SAT} \leq_p \text{CLIQUE}$$

The reduction algorithm will now be defined. A formula ϕ in 3-CNF consists of a sequence of k clauses ANDed together $\phi = C_1 \wedge C_2 \wedge, ..., \wedge C_k$. Each clause consists of 3-different litterals that are ORed together. We will construct a graph that only has a clique of size k if and only if ϕ is satisfiable. Note the if and only iff from the definition of the function f - it needs to be a mapping.

Now, we take each clause $C_r = l_1^r \vee l_2^r \vee l_3^r$ and place the triple $v_1^r \vee v_2^r \vee v_3^r$ into the vertex set V of the graph to be created. We also place an edge between two vertices v_i^r and v_j^s *it holds that* $r \neq s$(two different triplets) *and the two literals* v_i^r *and*v_j^s *are consistent*(not the negation of each other - $x_1, \sim x_2$) .

Showing the 3CNF-SAT implies CLIQUE

Suppose that ϕ is in fact a satisfying assignment. Then each clause C_r must contain at least one literal l_i^r that is assignment 1. This literal will in G be the vertex v_i^r. If we from each clause pick a literal that gets assigned 1, then we will have k vertices V' in V. Now, any vertices $v_i^r, v_j^s \in V'$ map to 1 and are of different clauses (that is how we picked them). Furthermore since both literal l_i^r, l_j^s are 1 they cannot be complements of each other - satisfying the other part of how we constructed G, and therefore there must be an edge between v_i^r, v_j^s. We picked two random vertices, so this holds for all vertices.

Showing CLIQUE implies 3CNF-SAT

2.12 Vertex cover

First we suppose that G has a clique with vertices $V' \subseteq V$ of size k. Then we pick any edge $(u, v) \in \bar{E}$. Since the edges in the clique will all be edges from G, and the edge $(u, v) \notin E$ then this means that either u or v or both cannot be in

V'(since if both were in V' it violates our assumption that $(u,v) \in \bar{E}$, since the clique only contains edges in G. One of them might not be in V' also, since it could be that v is in V', but there is not edge from u to v).

If we look at the set of vertices $V - V'$, then we know theses are the vertices that are not in the clique. We also know that the edges in the clique cannot be any edges in $\bar{E}$. We must remember that u, v are still vertices in V. And since either one or both were not in V', then at least one (or both) of u, v will be in $V - V'$. Therefore we now know that the edge (u,v) will be covered by $V - V'$. We took the edge (u,v)randomly from $\bar{E}$, and therefore we know that all $\bar{E}$ are covered by $V - V'$. We know the size of this cover set is $|V| - k$.

Now we need to show the other direction, so we will have to assume that $\bar{G}$has a vertex cover V' of size $|V'| = |V| - k$. If we have an edge $(u,v) \in \bar{E}$,then we must have that u or v or both are in V', since we know that $\bar{G}$ is covered by V' (therefore the edges in $\bar{E}$ must be incident on the vertices in V'). This means that for all vertices $u, v \in V$ if $u \notin V'$ and $v \notin V'$ then we will have that $(u,v) \in E$, since when both edges $u, v \in V'$ then they are an edge of $\bar{E}$(we chose that V' was a cover of $\bar{G}$which has the edge set $\bar{E}$), and whenever the pair $(u,v)$are not an edge of $\bar{E}$they are an edge of E (we chose $\bar{E}$ to be exactly all edges not in E).

2.13 Traveling salesman

In the traveling salesman problem, the salesman wants to make a full **tour** visiting n cities at a minimal cost. This can be modelled as a graph $G = (V, E)$ with the vertices $v \in V$, $|V| = n$ representing cities, and the edges representing roads. The salesman must now visit each vertex in G exactly once. The cost function $c(i,j)$ states the cost of traveling from city i to city j. The formal language defining the problem is

$$\begin{aligned}
G = \{\langle G, c, k\rangle : G = (V, E)&\text{is a complete graph} && (7)\\
c &\text{ is a function } V \times V \mapsto N && (8)\\
k &\in N && (9)\\
G &\text{ has a tour with cost at most } k && (10)\\
\} & && (11)
\end{aligned}$$

Showing TSP is in NP

Given an instance of the tour, we can as a certificate take the sequence of n vertices visited on the tour. The verification algorithm will then check that each vertex in V is visited exactly once, and that the edge cost over tour is at most k. This can be done in polynomial time the number of edges visited.

Showing NP-hardness

It can be shown that TSP is reducible from HAM-CYCLE: HAM-CYCLE $\leq_p$ TSP. We let the graph $G = (V, E)$ be an instance of the HAM-CYCLE problem. Then we will reduce this to a TSP problem be constructing the graph $G' =$

(V, E'), where we let the set of edges $E' = \{(i,j) : i,j \in V\}, i \neq j$. That is, there is an edge between every pair of vertices. Since the TSP needs a cost function, we will define

$$c(i,j) = \begin{cases} 0 \text{ if } (i,j) \in E \\ 1 \text{ if } (i,j) \notin E \end{cases}$$

The transformed instance to the TSP problem is $\langle G, c, 0\rangle$, since this is the tour of the graph with minimal cost. The graph can easily be created in polynomial time in the number of vertices. What is needed to show is then that G has a hamiltonian cycle if and only if G' has a tour of 0 (minimal cost tour). If we suppose that G has a hamiltonian cycle h, then each edge visited will be in E meaning that the cycle will visit all cities/vertices with cost 0 due to the way we defined the cost function. That is, G' has a tour with cost 0, and both problems are yes-instances. Now the other side of the bi-implication. if we suppose that G' has a tour with cost at most 0. Then we defined the cost function in such a way that edges in E' have costs as either 1 or 0, so the cost must be exactly 0. We need the at most 0 due to the definition of TSP. And since the cost is 0, then the tour visists only edges in E, and since a tour visists all cities then we visist all edges, so the tour is also a hamiltonian cycle. We have now showed that the bi-implication holds and TSP is also NPC.

2.14 Subset-sum