Chapter 15

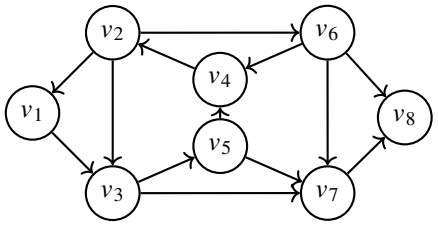
Non-Deterministic Turing Machines and Class $\mathcal{NP}$

In the previous two chapters, We have introduced the concept of time complexity of a (deterministic) Turing machine (DTM) and, based on this, we have defined classes of time complexity of TM's (DTIME($t(n)$)). Finally, we have introduced a class $\mathcal{P}$ of languages/problems, the languages/problems that have "efficient" solutions. We also call these languages/problems the *easy* problems. The next question we can ask is if there are any problems that are not easy? Obviously, all of the undecidable languages should be considered as non-easy problems, because, for an arbitrary string, we might not even be able to decide theoretically whether it belongs to the language or not. Much more complex question is whether there are decidable languages/problems that are not easy? That is, do there exist languages for which there exist Turing Machines that decide them, meaning there exist Turing Machines that halt on all inputs and accept only the strings from that language, but no efficient Turing Machine (that of polynomial time complexity) exists to solve them. This is a considerably harder question than whether undecidable languages exist, because we cannot use any similar argument that we used to come up with unrecognisable/undecidable language.

We are going to start by looking at an example of a potentially hard problem. That is, the problem for which no polynomial time solution is known (although it is not yet certain that one does not exist).

15.1 Hamiltonian Path in a Graph

A *graph* G is a pair (V, E), where V is the set of *vertices* (or *nodes*) and E is the set of *edges*. Each edge $e \in E$ is a pair of (v_1, v_2), where v_1 and v_2 are vertices from V. An example of a graph G is given in the figure below.



$$G = (V, E)$$

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$$

$$E = \{(v_1, v_3), (v_3, v_5), (v_3, v_7), (v_5, v_7), (v_7, v_8), (v_2, v_1), (v_2, v_6), (v_6, v_8),$$
$$(v_2, v_3), (v_6, v_7), (v_5, v_4), (v_4, v_2), (v_6, v_4)\}$$

A *Hamiltonian Path* between two nodes v_i and v_j in the graph G is a (directed) path between v_i and v_j that goes through each node of the graph *exactly once*. A Hamiltonian Path from v_1 to v_8 in our example graph can be obtained following the sequence of edges

$$(v_1, v_3), (v_3, v_5), (v_5, v_4), (v_4, v_2), (v_2, v_6), (v_6, v_7), (v_7, v_8).$$

Hamiltonian Path Problem can be defined in the following way.

Problem 15.1: HAMPATH Problem

Given a graph G and two nodes v_i and v_j, return yes if there exists a Hamiltonian Path between v_i and v_j in G and no otherwise.

For our example, we have seen that a Hamiltonian Path exists between nodes v_1 and v_8. But, for example, there does not exist a Hamiltonian Path between the nodes v_3 and v_8. This can easily be seen if we try different paths from v_3 - we can never reach v_8 and visit every other node exactly once. In general, whether there exists a Hamiltonian Path between the two nodes in a graph can be checked by considering all different paths between these two nodes, and for each of them, checking whether it visits every node exactly once. So, the HAMPATH problem is definitely decidable. However, checking all the paths in the graph can definitely *not* be done in polynomial time, using any reasonable definition of a size of an instance (i.e. the size of a graph). In general, for a graph G, its size is usually either the number of nodes in it, the number of edges, or the number of nodes plus the number of edges. Checking all the paths in a graph is, in the worst case, equivalent to checking all the subsets of the nodes of a graph and testing whether there exists a path connecting all these nodes. The complexity of this is roughly 2^n, which is the number of all subsets of nodes of a graph with n nodes. So, we have an exponential algorithm to decide each instance of the HAMPATH problem. *But currently no polynomial-time algorithm is known for solving this problem.*

HAMPATH problem has one more interesting feature. While *finding a solution* to an instance of the problem is potentially hard, *verifying* whether something is a solution is far easier. That is, finding a Hamiltonian Path between the two nodes might be hard, but if we are somehow given a path, it is easy to verify whether the path is Hamiltonian or not - we just need to verify that it starts in the starting node, ends in the ending node, that each node is visited exactly once and that there exists an edge between every two successive nodes in the path. All this can definitely be done in polynomial time. Ability to easily verify whether something is a solution or not will be a feature of many problems that we will see. It turns out that these are exactly the problems that can be solved in polynomial time by a *nondeterministic Turing Machine*, which we are going to introduce next.

15.2 Non-Deterministic Turing Machines

Definition 15.1: Non-Deterministic Turing Machine (NDTM)

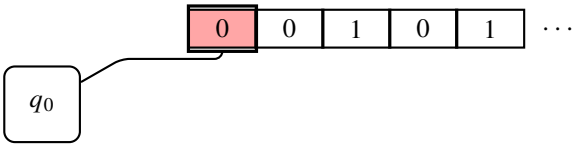
A Non-Deterministic Turing Machine M is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$, where

- Q is a finite set of states;
- Σ is an input alphabet;
- Γ is a set of allowed tape symbols;
- $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$ is a transition function (or *next move* function). Therefore, for a state q and a tape symbol X

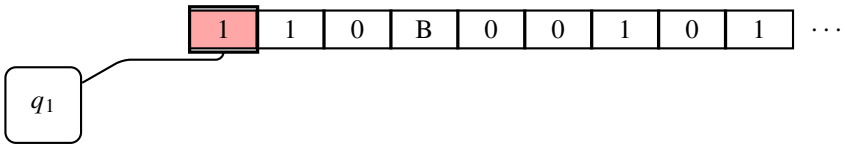
$$\delta(q, X) = \{(p_1, Y_1, D_1), (p_2, Y_2, D_2), \ldots, (p_k, Y_k, D_k)\},$$

  where $p_1, p_2, \ldots, p_k$ are the states from Q, $Y_1, Y_2, \ldots, Y_k$ are tape symbols and $D_1, D_2, \ldots, D_k$ are directions, each being either L or R. δ might be undefined for some pairs of states and tape symbols.
- q_0 is the starting state;
- B is a special, blank symbol from Γ, $B \notin \Sigma$;
- $F \subseteq Q$ is the set of final states;

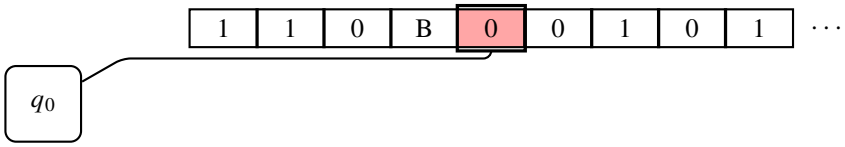
Similarly as with finite automata, the difference between a deterministic and non-deterministic Turing Machine is in the transition function. For an state q and a tape symbol X, a non-deterministic Turing Machine can have more than one possible move. Note that a single move is a fixed tuple of a state, a tape symbol and



(a) Start of computation.



(b) Guessing stage.



(c) Checking stage.

Figure 15.1: Example of a verifier non-deterministic Turing Machine

a direction. A non-deterministic Turing Machine cannot select a state from one tuple, a tape symbol from another and a direction from the third - all three have to come from the same tuple.

NDTM M halts on some input string w if any sequence of moves leads to M halting. The string w is accepted by a NDTM M if any sequence of moves leads to M halting in an accepting state. As is the case with finite automata, adding non-determinism to Turing Machines does not increase their power, in terms of the languages that are accepted. We will leave the following theorem without a proof.

Theorem 15.1: Equivalence Between Deterministic and Non-Deterministic Turing Machines

Let L be a language accepted by a NDTM M. Then there is a deterministic Turing Machine M' that accepts L.

So, the class of languages accepted by non-deterministic Turing Machines is the same as the class of languages accepted by deterministic Turing Machines. Furthermore, if the language is decided by a NDTM, then it is also decided by some deterministic Turing Machine. However, we cannot claim anything about efficiency of the equivalent deterministic Turing Machines. That is, if a language is decided by a NDTM that has polynomial time complexity, we do not know whether this also means that the language is decided by some deterministic Turing Machine in polynomial time.

There is also an alternative definition of a Turing Machine that is used in some literature, and that is directly related to our concept of verifiability. We can define a NDTM so that it has exactly the same components as a deterministic Turing Machine (including a transition function that makes at most one move for each state and each tape symbol). However, a NDTM operates on a *two-way infinite tape*, with the input string w being written on cells 1 to $|w|$ of the tape. The operation of such a machine proceeds in two stages:

1. *Guessing Stage:* A *guess* (a sequence of tape symbols) is written to the left of the string w on the tape (positions $-1, -2, \ldots$).

2. *Checking Stage:* If and when the guessing stage finishes, the tape head moves back to the beginning of the input string w and the Turing Machine proceeds in a deterministic way.

This is the definition of a NDTM as a *verifier*. In the remainder of the text, we will refer to such a NDTM as a *verifier NDTM*. The example of a verifier NDTM is given in Figure 15.1

In the checking stage, a verifier NDTM will examine and use the guess for computation in some way. Alternatively, it might disregard the guess altogether, in which case we get a deterministic Turing Machine

where the outcome of computation does not depend in any way on the guess. The computation ends if and when the verifier NDTM halts in the checking stage. The computation is *accepting* if it halts in an accepting state in the checking stage. In any other case, it is *non-accepting*. The string is *accepted* by the verifier NDTM if at least one computation (with some guess) is accepting. Note that, for each input string, there is an unlimited number of computations (with an unlimited number of guesses).

Language L accepted by a verifier NDTM is the set of all strings for which there is at least one accepting computation. A verifier NDTM is said to halt on a particular input if there exists at least one computation (for one guess) that halts in an accepting or non-accepting state. Therefore, if the verifier NDTM halts on all inputs, that means that for each input there exists at least one halting computation for one guess. We, however, do not make an assumption that the verifier NDTM halts on *all* guesses, not even for the strings that are accepted by the machine!

The two definitions of the NDTM, the standard definition and the verifier NDTM, seem to be completely different, but in fact they are equivalent.

Theorem 15.2

If the language L is accepted by some verifier NDTM M, then there also exists a NDTM M' defined in a standard way that accepts L. The converse is also true - if the language L is accepted by some NDTM defined in a standard way, then L is also accepted by some verifier NDTM.

15.3 Time Complexity of a Non-Deterministic Turing Machine

With a deterministic Turing Machine that always halts, determining time complexity was easy, because there was only a single computation on each fixed input. With either of the two definitions of a NDTM, the situation is more complicated. Focusing on the verifier NDTM, we know that there are infinitely many computations for each fixed input. We haven't assumed that all of the computations (with all of the guesses) for a particular input halt. The question then is how we can define the time complexity of such a machine.

We are going to define the time it takes for a verifier NDTM to finish computation for some input only in the cases when the input is accepted.

Definition 15.2: Time it Takes for a Verifier NDTM to Accept a String

The time it takes for a verifier NDTM M to accept some string $w \in L(M)$, denoted by $t_M(w)$, is the *minimal* time over all the accepting computations for w.

It might sound strange that we are defining the time it takes a verifier NDTM to accept a string as a minimal time over all the accepting computation, especially since we defined the time complexity of a deterministic Turing Machine as a *maximal* time over all strings of size n. However, for a fixed string, we have a good justification for defining the time for computation in this way (note that we are not yet talking about time complexity of the verifier NDTM for inputs of size n, but rather the number of steps it takes for one fixed string). We can imagine, for a fixed input string, a verifier NDTM running on all guesses *in parallel*, and halting as soon as the first of these computations accepts and terminate all the others. Therefore, the time it takes for such a machine to halt is the time it takes for the quickest accepting computation to halt.

Remark 15.1

In most of the literature, the standard definition of a NDTM is used, and it is assumed that if the NDTM decides a language, that all the computation paths for all the strings, regardless of whether they are in the languge or not, halt. Here, the time it takes for a NDTM to decide on a string is the maximum time over all the execution paths. Even some literature that considers the verifier NDTMs makes the same assumption, that if the verifier NDTM decides a language, then its computation halts for every guess and every input string. We find our definition of the time it takes for a verifier NDTM to accept a string more intuitive, as it more closely resembles the way we think about computation of a NDTM - which is essentially trying all possible computation paths in parallel. Note also that, for a verifier NDTM that decides the language L, we can define the time it takes a machine to reject the strings in the same way as we did for the acceptance, because we know that the verifier NDTM will halt for some guess also for the strings that are not in its language.

Definition 15.3: Time Complexity of a Verifier NDTM

The time complexity of a verifier NDTM M, $T_M(n)$, is the maximum time that M takes to accept any of the input strings w of size n:

$$T_M(n) = \max\{t_M(w) | |w| = n\}.$$

If M does not accept any string of length n, then $T_M(n) = 1$.

This looks more familiar, as we are defining the time complexity of a verifier NDTM as a function of the size of the input instance, and furthermore, for a given instance size n, the time complexity is the maximum time it takes to accept any string of length n. This is similar as what we had for deterministic Turing Machines. Note again that the time a verifier NDTM takes to accept *one instance* is the *minimal* time over all the guesses, but the time complexity for a given *instance size* is the *maximal* time over all accepted instances of that size.

15.4 Time Complexity Classes for NDTM and Class $\mathcal{NP}$

Definition 15.4: NTIME

We say that the language/problem L belongs to the class of languages/problems NTIME($t(n)$) if there exists a verifier NDTM M that accepts L such that the time complexity $T_M(n)$ of M is $O(t(n))$.

Definition 15.5: Class $\mathcal{NP}$

The language/problem L belongs to the class of languages $\mathcal{NP}$ if it belongs to the class NTIME(n^k) for some $k \in \mathbb{N}$. In other words,

$$\mathcal{NP} = \bigcup_{k=1}^{\infty} \text{NTIME}(n^k).$$

Thus, the language L is in $\mathcal{NP}$ if there exists a verifier NDTM of polynomial-time complexity that accepts exactly the language L. The name $\mathcal{NP}$ comes from *nondeterministic polynomial* time. Also, the class $\mathcal{NP}$ is robust with respect to which definition of a NDTM we use. It can be easily shown that $L \in \mathcal{NP}$ if and only if there exists a standard NDTM M that decides on strings in L in polynomial time. Also, the class $\mathcal{NP}$ does not change if we change the definition of the time it takes to accept a string to the maximal time over all the accepting computations (see Remark 15.1).

What does it mean for the problem to be in the class $\mathcal{NP}$. If the problem is in $\mathcal{NP}$, then there is a verifier NDTM M with the property that for each instance of the problem w for which the answer to the problem is "yes", it is possible to find a guess c for which M accepts that instance in the polynomial time. c is usually called a *certificate* or a *proof*, and most often it represent the guess of the solution of the more general problem. In other words, M *verifies* a solution in a polynomial-time. The problem might not be *solvable* in polynomial-time, but it is verifiable in polynomial-time.

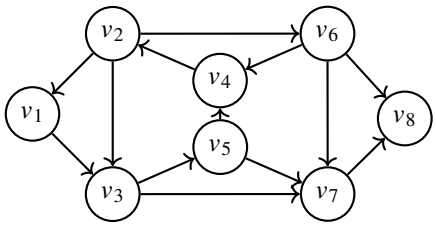
15.5 Hamiltonian Path Problem Revisited

Recall that when we talked about the HAMPATH problem, we said that it has exactly the feature that solving an instance of the problem, in terms of finding a sequence of nodes that constitute a Hamiltonian path, might be hard, but verifying whether something is a solution is far easier. We can design a verifier NDTM that takes as an input an encoding of a graph G and an encoding of the two nodes from it, v_i and v_j, and that guesses a sequence of nodes in the graph $u_1, u_2, \ldots, u_n$, such that $u_1 = v_i$, $u_n = v_j$ and G has n nodes. The checking stage for this guess is then verification whether the guessed sequence of nodes constitutes a Hamiltonian path. This is easily done in polynomial time with respect to the number of nodes n.

The complete computation over some input instance (G, v_i, v_j) would include computations over all different guesses (guessing and checking stage for each of them). If there exists a Hamiltonian Path in G from v_i to v_j, then a certificate for this instance will exist and the verifier NDTM will answer "yes". If there does

not exist a Hamiltonian Path in G, no guess will result in an accepting computation, therefore the verifier NDTM will not answer "yes".

Let us take a look again at our example instance of the HAMPATH problem.



We could guess the solution for this graph and nodes v_1 and v_8 is $(v_1, v_3, v_5, v_4, v_2, v_6, v_7, v_8)$ and verify easily that this is a solution. In the case of the HAMPATH problem, it is equally easy to verify that some sequence of graphs is *not* a solution, too. The verifier NDTM would follow exactly the same procedure and halt with an answer "no" when it discovers that a (wrong) guess does not constitute a valid Hamiltonian path in the graph.

Alternatively, we can consider a standard NDTM that follows all the paths from v_i. Whenever there is more than one edge from some node, we split the computation in as many branches as there are edges, and follow them all in parallel. A computation halts in a non-accepting state when the length of path is less than n (where n is the number of nodes) and we encounter the same node we have already found in that path or when there are no more edges to follow. If the path is of length n, and if the last node is v_j then we halt in an accepting state. Otherwise we halt in a non-accepting state. Each path is of length at most n and it takes at most $O(n)$ time for each new node on the path to check whether it is already on the path. Therefore, each branch of computation will take at most $O(n^2)$ time. Each branch terminates in $O(n^2)$ time, therefore time complexity of this NDTM would be $O(n^2)$. Therefore, there is a polynomial-time standard NDTM that decides on each instance of the HAMPATH problem.

15.6 Non-$\mathcal{NP}$ Problems and $\mathcal{P} = \mathcal{NP}$ Problem

We may intuitively think that every problem has to belong to the class $\mathcal{NP}$. If we are given a solution to a problem, it is surely always easy to verify that this is a solution. However, this is not necessarily true. Consider the following problem:

Problem 15.2: $\overline{\text{HAMPATH}}$ Problem

Given a graph G and two nodes v_i and v_j, return yes if there **does not** exist a Hamiltonian Path from v_i to v_j and no otherwise.

Even if we know that the answer to this problem is *yes*, it is hard to verify that this is the correct answer without trying all the paths in the graph (which is definitely not a polynomial-time computation). Therefore, it is not clear whether $\overline{\text{HAMPATH}}$ is in $\mathcal{NP}$.

We now turn our attention to what is universally accepted as the most important open question in Computer Science. Since every deterministic Turing Machine can be seen as a standard NDTM (with one choice of a move for every state and every input symbol), it is obvious that $\mathcal{P} \subset \mathcal{NP}$. This is intuitively obvious also if we consider verifier NDTMs - if we can solve a problem in polynomial-time, we can certainly verify the guessed solution in polynomial time. We simply ignore the guessed solution altogether, compute a solution and then compare it with the guessed one.

It also looks like the $\mathcal{NP}$ class should be larger than the $\mathcal{P}$. It seems that there have to exist polynomially-verifiable problems that are not polynomially-solvable. But this has not been proven yet. There is still no known problem for which it was proven that it is in $\mathcal{NP}$, but not in $\mathcal{P}$. The question whether the classes $\mathcal{P}$ and $\mathcal{NP}$ are the same is known as the "$\mathcal{P} = \mathcal{NP}$" and is, as we have said, considered to be the most important open question in Computer Science.

To prove that $\mathcal{P} \neq \mathcal{NP}$, we would need to find a problem from the class $\mathcal{NP}$ for which no polynomial-time deterministic Turing Machine can exists that solves it. On the other hand, to prove that $\mathcal{P} = \mathcal{NP}$, we would need to prove that every polynomially-verifiable problem is also polynomially-solvable. Many scientists believe that $\mathcal{P} \neq \mathcal{NP}$, because many $\mathcal{NP}$ problems have been known for a long time for which still no polynomial-time solution has been found.