Complexity Classes

A complexity class is a collection of languages. We determine a complexity class by specifying three things:

1. A model of computation (such as a deterministic Turing machine, or a nondeterministic Turing Machine, or a parallel Random Access Machine).
2. A resource (such as time, space or number of processors).
3. A set of bounds. This is a set of functions that are used to bound the amount of resource we can use.

What resources it is reasonable to consider depends on the model of computation we have chosen. We will, in general, consider Turing machines (either deterministic or nondeterministic). The resources we are primarily concerned with are *time* (which means the number of steps in a computation), and *space*, which means the maximum number of tape cells used in a computation. Many other models of computation allow similar notions of time and space to be defined. For instance, on a register machine, one could define space as the number of registers used, though a more reasonable measure might be to count the number of bits required to represent the numbers stored in the registers. Time could still be defined as the number of steps in the computation.

As long as we are considering deterministic machines, if we choose our set of functions (in item 3) to be broad enough, the languages that are included in the complexity class do not depend on the particular model of computation. So, if we take the collection of functions to be the set of all polynomials, it does not matter whether we consider Turing machines or register machines, we get the same class of languages either way. This is not the case if we take the class of all linear functions. So, whether or not a language is decidable in linear time is not just a property of the language, but is sensitive to the model of computation. However, it is reasonable to say that whether or not a language is decidable in polynomial time is a property of the language itself, much like the property of being decidable itself.

Constructible Functions When choosing a function f to serve as a bound of resources, for example in defining a complexity class such $\mathsf{TIME}(f(n))$, we need to be careful. For one thing, as we saw before, it makes sense to only consider computable functions. However, there are some quite unnatural computable functions—for instance, a function that is 2^n for even n, and $\log n$ for odd numbers, which will naturally lead to quite unnatural classes of languages if used in the definition of a complexity class. From now on, we restrict the functions we use for bounds to what are called *constructible* functions.

Definition
A function $f : \mathbb{N} \to \mathbb{N}$ is *constructible* if:

- f is non-decreasing, i.e. $f(n+1) \geq f(n)$ for all n; and
- there is a machine M which, on any input of length n, replaces the input with the string $0^{f(n)}$, and M runs in time $O(n + f(n))$ and uses $O(f(n))$ *work space* (recall we do not count space on the input tape).

The intuition behind the second requirement is that computing the function f shouldn't require more resources than the limit imposed by f itself. This will, in particular, allow us to compose the computation of f with any other computation that takes place within $O(f(n))$ time and space. Thus, we can prove results such as the following:

> If L is in $\mathsf{TIME}(f(n))$, then there is a machine M that accepts L, and which halts on all inputs in $O(f(n))$ steps.

Examples All of the following functions are constructible:

- $\lceil \log n \rceil$;
- n^2;
- n;
- 2^n.

If f and g are constructible functions, then so are $f + g$, $f \cdot g$, 2^f and $f(g)$ (this last, provided that $f(n) > n$). This, together with the above examples, allows us to generate all the constructible functions we will ever need, including all polynomials $p(n)$, all functions $2^{p(n)}$ for polynomials p, and much else besides.

Nondeterministic Classes We have already defined $\mathsf{TIME}(f(n))$ and $\mathsf{SPACE}(f(n))$. We can define similar classes for nondeterministic computation.

> $\mathsf{NTIME}(f(n))$ is defined as the class of those languages L which are accepted by a *nondeterministic* Turing machine M, such that for every $x \in L$, there is an accepting computation of M on x of length at most $f(n)$.
>
> $\mathsf{NSPACE}(f(n))$ is the class of languages accepted by a *non-deterministic* Turing machine using at most $f(n)$ work space.

Complexity Classes However, in general we are not interested in complexity classes defined by single functions. We consider wider classes, in order to obtain robust definitions of complexity classes that are independent of particular machine models. The classes we are particularly interested in are the following:

> $\mathsf{P} = \bigcup_{k=1}^{\infty} \mathsf{TIME}(n^k)$
> The class of languages decidable in polynomial time.
>
> $\mathsf{NP} = \bigcup_{k=1}^{\infty} \mathsf{NTIME}(n^k)$
> The class of languages recognisable by a nondeterministic machine in polynomial time.
>
> $\mathsf{L} = \bigcup_{k=1}^{\infty} \mathsf{SPACE}(k \cdot \log n)$ The class of languages decidable using logarithmic workspace.
>
> $\mathsf{NL} = \bigcup_{k=1}^{\infty} \mathsf{NSPACE}(k \cdot \log n)$ The class of languages recognisable by a nondeterministic machine using logarithmic workspace.

$\mathsf{PSPACE} = \bigcup_{k=1}^{\infty} \mathsf{SPACE}(n^k)$
The class of languages decidable in polynomial space.

- $\mathsf{NPSPACE} = \bigcup_{k=1}^{\infty} \mathsf{NSPACE}(n^k)$ The class of languages recognisable by a nondeterministic machine using polynomial space.

Complement Classes For any language $L \subseteq \Sigma^\star$, the complement of L, sometimes written $\bar{L}$ is the language $\Sigma^\star \setminus L$. For any of the classes $\mathsf{P}, \mathsf{L}, \mathsf{PSPACE}$, defined in terms of deterministic machines, if L is in the class, then so is $\bar{L}$. We say that the classes P, L and PSPACE are *closed under complementation*. To see why this is the case, note that if M is a deterministic machine accepting a language L, and f is a constructible function bounding the running time or space of M, then we can obtain a machine M' which halts on all inputs within bounds $O(f)$. If we now define a machine M'' which is the same as M', but with the states `acc` and `rej` interchanged, it is easily seen that M'' accepts $\bar{L}$ within resource bounds given by f.

This argument does not work for nondeterministic languages. Interchanging the accepting and rejecting states of a nondeterministic machine accepting a language L does not result in a machine that accepts $\bar{L}$. For a particular input string x, a nondeterministic machine may have computations leading to both `acc` and `rej`. The machine with these two states interchanged will still have computations to both of them, and will also accept x. Nondeterministic complexity classes are not necessarily closed under complementation. We, therefore, define some more complexity classes:

co-NL — the languages whose complements are in NL.

co-NP — the languages whose complements are in NP.

co-NPSPACE — the languages whose complements are in NPSPACE.

As it happens, we are able to show that NPSPACE = co-NPSPACE, and NL = co-NL, or in other words, that NPSPACE and NL are closed under complementation. It remains an open question whether NP is closed under complementation, though it is widely believed that it is not.

Inclusions

We can show that the following inclusions hold among the complexity classes we have defined above (some of these inclusions are easier to prove than others):

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{NPSPACE}.$$

Moreover, since the classes L, P and PSPACE are all closed under complementation, we can strengthen some of these to the following:

$$\mathsf{L} \subseteq \mathsf{NL} \cap \text{co-}\mathsf{NL}, \mathsf{P} \subseteq \mathsf{NP} \cap \text{co-}\mathsf{NP} \quad \text{and} \quad \mathsf{PSPACE} \subseteq \mathsf{NPSPACE} \cap \text{co-}\mathsf{NPSPACE}.$$

To prove these inclusions, we show the following general inclusions hold for any constructible function f:

1. $\mathsf{SPACE}(f(n)) \subseteq \mathsf{NSPACE}(f(n))$;
2. $\mathsf{TIME}(f(n)) \subseteq \mathsf{NTIME}(f(n))$;
3. $\mathsf{NTIME}(f(n)) \subseteq \mathsf{SPACE}(f(n))$;
4. $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{TIME}(k^{\log n + f(n)})$ for some constant k;

Of these, 1 and 2 are straightforward from the definitions, as any deterministic machine is just a nondeterministic machine in which the transition relation δ happens to be functional.

The inclusion in 3 is an easy simulation. A deterministic machine can simulate a nondeterministic machine M by backtracking. In order to do this, it has to keep track of the current configuration of M as well as all the choices taken wherever a nondeterministic choice was available to M. The space required to record these choices is clearly no more than a constant multiple of the length of the computation. Moreover, the space required to store the configuration of M also cannot be more than the number of steps in the computation of M so far. So, the total work space required by the simulating machine is at $O(f)$, where f is the time bound on M.

Inclusion 4 requires some more work to establish, and we will return to it later.

Hierarchy

While proving lower bounds for specific problems such as the travelling salesman problem remains the holy grail of complexity theory, one instance where we know how to prove lower bounds is for problems that are constructed specifically for this purpose. That is, we can use diagonalisation to construct a language with a specific lower bound. This allows us to show, in particular that increasing the time (or space) bounds on a complexity class does give us the ability to recognise more languages.

One such hierarchy theorem we can show is:

Time Hierarchy Theorem
For any constructible function f with $f(n) \geq n$, $\mathsf{TIME}(f(n))$ is properly contained in $\mathsf{TIME}(f(2n+1)^3)$.

To see this, we define a version of the halting problem with time bound f. That is, define the language:

$$H_f = \{[M], x \mid M \text{ accepts } x \text{ in } f(|x|) \text{ steps}\}$$

We now make two observations. First:

$$H_f \in \mathsf{TIME}(f(n)^3).$$

This is actually a rather loose upper bound. A machine for recognising H_f would first compute $f(|x|)$, and on a separate work tape, write out 0, $f(|x|)$ times, to use as a clock for the rest of the computation. It would then simulate machine M on input x for $f(|x|)$ many steps, at each step looking through the description of M given for the appropriate transition. The calculation of the time bound is left as an exercise.

The second observation is:

$$H_f \notin \mathsf{TIME}(f(\lfloor n/2 \rfloor)).$$

The argument for this is similar to the argument that the halting problem H is undecidable. Suppose $H_f \in \mathsf{TIME}(f(\lfloor n/2 \rfloor))$. Then, we can construct a machine N which accepts $[M]$ if, and only if, $[M], [M] \notin H_f$. The machine simply copies $[M]$, inserting a comma between the two copies, and then runs the machine that accepts H_f. Moreover, the running time of N on an input of length n is $f(\lfloor (2n+1)/2 \rfloor) = f(n)$. We can now as whether N accepts the input $[N]$, and we see that we get a contradiction either way.

From these two observations, the Time Hierarchy Theorem immediately follows.

Among the consequences of the Time Hierarchy Theorem is that there is no fixed k such that all languages in P can be decided in time $O(n^k)$. Another consequence is that the complexity class EXP, defined by:

$$\mathsf{EXP} = \bigcup_{k=1}^{\infty} \mathsf{TIME}(2^{n^k}),$$

is a proper extension of P. That is, $\mathsf{P} \subseteq \mathsf{EXP}$, but $\mathsf{EXP} \not\subseteq \mathsf{P}$.

Similar results can be obtained for space complexity, by proving a Space Hierarchy Theorem. See Exercise Sheet 2.

Reachability

To establish the fourth of the inclusions we claimed, namely that

$$\mathsf{NSPACE}(f(n)) \subseteq \mathsf{TIME}(k^{\log n + f(n)})$$

we analyse a particular problem on graphs, called the *graph reachability problem*. This problem is central to our understanding of nondeterministic space complexity classes. Before we begin its study, it is worth pointing out that the above mentioned inclusion implies that $\mathsf{NL} \subseteq \mathsf{P}$. This is because if we let $f(n)$ be $\log n$ in the inclusion, we have that:

$$\mathsf{NSPACE}(\log n) \subseteq \mathsf{TIME}(k^{2\log n}) = \mathsf{TIME}(n^{2\log k}).$$

The class on the right is clearly contained in P.

The Reachability problem is defined as the problem where, given as input a directed graph $G = (V, E)$, and two nodes $a, b \in V$ we are to decide whether there is a path from a to b in G. A straightforward algorithm for doing this searches through the graph, proceeding as follows:

1. mark node a, leaving other nodes unmarked, and initialise set S to $\{a\}$;

2. while S is not empty, choose node i in S: remove i from S and for all j such that there is an edge (i, j) and j is unmarked, mark j and add j to S;

3. if b is marked, accept else reject.

The algorithm as presented is somewhat vague in the details, but it can clearly be turned into a working implementation. To give a more detailed specification, one would have to state what data structure is used to implement S, and how the node i is chosen in step 2. For instance, S could be implemented as a stack, which would result in a depth-first search of the graph G, or it could be a queue, resulting in a breadth-first search. However, it should be reasonably clear that any implementation can be carried out on a Turing machine.

What is the time and space complexity of this algorithm? During the running of the algorithm, every edge in G must be examined at least once. Moreover, it can be seen that each edge is not examined more than once. This is because no vertex is added to S more than once, since once it is added, it is marked, and each edge is examined only when the vertex at its source is removed from S. So, we can safely say, if n is the number of vertices in the graph, that the running time of the algorithm is $O(n^2)$. An actual implementation on a Turing machine may require more time, but it can certainly be done in polynomial time, a point that has been emphasised several time earlier.

In terms of space, the only requirements for work space are the two sets—S and the set of marked vertices. Each can be implemented using n bits, one for each vertex. We may need some additional counters, each of $\log n$ bits, but the total work space requirement can be bounded by $O(n)$.

So, the above algorithm establishes that $\mathsf{Reachability}$ is a problem in P, and in $\mathsf{SPACE}(O(n))$. However, the latter upper bound can be improved. We first demonstrate a nondeterministic algorithm for solving the $\mathsf{Reachability}$ problem that shows that this problem is in NL. The algorithm is the following:

```
1. write the index of node $a$ in the work space;
2. if $i$ is the index currently written on the work space:
    (a) if $i = b$ then accept, else
        guess an index $j$ ($\log n$ bits) and write it on the work space.
    (b) if $(i, j)$ is not an edge, reject, else replace $i$ by $j$ and return to (2).
```

In the above description, to "guess an index j" means to perform $\log n$ steps, each of which has a nondeterministic choice of either writing a 0 or a 1 on a work tape and moving to the right. At the end of these steps, $\log n$ bits have been written. Moreover, for every index j, there is a computation path that results in j being written on the tape. Essentially, this algorithm can be seen as trying all possible indices j in parallel. For those j for which there is is an edge (i, j), the computation can continue. If there is any path from a to b in the graph, there will be a computation of this machine which successively visits all the nodes on that path.

The space requirements of the above algorithm are simple. It needs to store two indices, each of $\log n$ bits, and therefore uses $O(\log n)$ space. Hence $\mathsf{Reachability}$ is in NL.

The significance of the fact that $\mathsf{Reachability}$ is in NL is in that this problem can stand in for all problems in NL. There is a precise sense in which this is true. See Exercise sheet 2 for details. Here, we will just note that the fact that there is a polynomial time deterministic algorithm for $\mathsf{Reachability}$ can be used to show that all problems in NL are in P. In general, we wish to show that for any constructible function f, $\mathsf{NSPACE}(\log n) \subseteq \mathsf{TIME}(k^{2 \log n})$.

Suppose M is a nondeterministic machine working with workspace bounded by $f(n)$ for inputs of length n. For a given input string x of length n there is a fixed finite number of configurations of M that are possible. The finite state control can be in any of q states, where q is a number which does not depend on x at all. The work tape can have one of $s^{f(n)}$ strings on it, where s is the number of distinct symbols in the tape alphabet. The head on the input tape can be in one of n different positions, and the head on the work tape can be in one of $f(n)$ different positions. Thus, the total number of distinct configurations is no more than $qnf(n)s^{f(n)}$. For some constant c, this is less than $nc^{f(n)}$.

We define the *configuration graph* of the machine M on input x to be the graph whose nodes are all possible configurations of M with x on the input tape, and the work tape having at most $f(|x|)$ symbols, and there is an edge between two configurations i and j if, and only if, $i \to_M j$, i.e. the machine M can make the transition from configuration i to configuration j in one step.

Then, it is clear that M accepts x if, and only if, there is a path from the starting configuration $(s, \triangleright, x, \triangleright, \varepsilon)$ to an accepting configuration (that is a configuration with state acc). So, the problem of determining whether M accepts x is the same as the graph reachability problem on the configuration graph of M on x. We have a deterministic algorithm that solves the graph reachability problem in time $O(n^2)$. Thus, given any nondeterministic machine M that runs using workspace $f(n)$, we can define a deterministic machine that accepts the same language which runs by first generating the configuration graph of M on the given input x, and then using the Reachability algorithm. The time taken by this deterministic machine is $O(g^2)$, where g is the size of the configuration graph. That is the time is at most $c'(nc^{f(n)})^2$ for some constant c'. But this is $k^{\log n + f(n)}$, for some constant $k \leq c'c^2$.

In addition to establishing that NL $\subseteq$ P, this also shows that NPSPACE $\subseteq$ EXP.

Savitch's Theorem

We can get more information about nondeterministic space complexity classes by examining other algorithms for Reachability. In particular, we can show that Reachability is solvable by a *deterministic* algorithm which uses only $O((\log n)^2)$ space. If we are only concerned about space, and not about time, this is an improvement on the deterministic algorithm we saw before. Consider the following recursive algorithm for determining if there is a path in the graph from a to b of length i or less.

```
Path(a,b,i).

 if i=1 and there is no edge (a,b)
    then reject
 else if there is an edge (a,b) or a=b
    then accept
 else for each vertex x
        if Path(a,x,ceil(i/2)) and Path(x,b,ceil(i/2)) then accept
```

Where ceil(i/2) is $\lceil i/2 \rceil$.

There is a path from a to b in a graph G with n vertices if, and only if, there is a path of length n or less. So, we can solve the reachability problem by checking if Path(a, b, n) holds. To analyse the space complexity of this algorithm, observe that the recursion can be implemented by keeping a stack of records, each of which contains a triple (a, b, i). The candidate middle vertex x can be implemented as a counter that takes values between 1 and n, and therefore requires $\log n$ bits. Each activation record on the stack can be represented using $3 \log n$ bits ($\log n$ for each of the three components). The maximum depth of recursion is at most $\log n$, since the value of i is halved at each nested recursive call. Moreover, for each nested call, at most two activation records are placed on the stack. Thus, we need space on the stack for at most $2 \log n$ records. It follows that $6(\log n)^2$ bits of space on the stack suffice. The algorithm therefore used space $O((\log n)^2)$.

We can use this algorithm to show that for any constructible function f such that $f(n) \geq \log n$, $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{SPACE}(f(n)^2)$. The idea, once again, is to solve the Reachability problem on the configuration graph of a nondeterministic machine M, which uses workspace $O(f(n))$. The configuration graph has $g = c^{\log n + f(n)}$ nodes, for some constant c, and therefore the reachability problem can be solved using space

$$O((\log g)^2) = O((\log n + f(n))^2) = O((f(n))^2)$$

. The last of these equalities follows from the fact that $f(n) \geq \log n$.

This would work, except that, in order to run the $O((\log n)^2)$ algorithm for reachability, we have to first produce the configuration graph on tape. And the tape that contains the configuration graph is part of the work space of the machine that simulates M. However, the configuration graph has $c^{\log n + f(n)}$ nodes, and therefore takes more than $f(n)^2$ space. The solution is that we do not keep the entire configuration graph on tape. Rather, whenever we need to look up the graph, that is, when we need to check for a pair (i, j) of configurations whether there is an edge between them, we write out the pair of configurations and check, by looking at the machine M, whether configuration j can be reached from i in one step. In effect, the description of M serves as a compact description of the configuration graph. With this, we can see that the total amount of work space needed is no more than $O((f(n))^2)$.

From the inclusion $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{SPACE}(f(n)^2)$ follows Savitch's theorem:

Theorem

PSPACE = NPSPACE.

From which it also follows that NPSPACE = co-NPSPACE, since PSPACE is closed under complementation. However, a more general result about the closure of nondeterministic space classes under complementation is known. Immerman and Szelepcśenyi proved that for any constructible function f with $f(n) \geq \log n$, $\mathsf{NSPACE}(f(n)) = \mathsf{co\text{-}NSPACE}$. The proof is based on a still more clever algorithm for Reachability, which shows that there is a nondeterministic machine, which with $O(\log n)$ work space determines the number of nodes reachable from a node a.