CA320 - Computability & Complexity

Complexity

Dr. David Sinclair

Big-O Notation

A function $g(N)$ is said to be $O(f(N))$ if exists constants c_0 and N_0 such that

$$g(N) < c_0 f(N),\ \forall N > N_0$$

Algorithms can be generally classified as:

- $O(1)$ Run time is independent of the size of the problem, N.
- $O(log(N))$ Occurs when a big problem is solved by transforming it into a smaller size by some constant fraction. (Logarithmic)
- $O(N)$ Occurs when each element of the problem requires a small amount of processing. (Linear)

Big-O Notation (2)

$O(Nlog(N))$ Occurs when a problem is broken into smaller subproblems, solving them independently, and combining the solutions. (Linearithmic)

$O(N^2)$ Occurs when an algorithm processes all pairs of elements. (Quadratic)

$O(2^N)$ Exponential run time. To be avoided. Characteristic of "brute force" approaches. (Exponential).

If each step takes 1ms (10^{-3}s) then:

N	$O(log(N))$	$O(\sqrt{N})$	$O(N)$	$O(Nlog(N))$	$O(N^2)$	$O(2^N)$
100	6.64ms	10ms	0.1s	0.2s	10s	$4x10^{19}$ years
200	7.64ms	14.1ms	0.2s	0.46s	40s	$5.1x10^{48}$ years

The age of the universe is about $14x10^9$ years!

Complexity Measures

There are 2 different kinds of complexity measures:

- *static measures* (program size) that are based on the structure of the algorithm, and
- *dynamic measures* (time and space) that are based on the algorithm and the inputs and hence are based on the behaviour of the computation.

Dynamic complexity measures are more descriptive and therefore are the interesting ones. A model-independent dynamic complexity measure Φ with respect to a set of algorithms $\{P_i\}$ should have the following properties.

- For each i, the domain of Φ_i coincides with that of P_i, and its codomain is a subset of N.
- A total computable predicate M exists such that

$$M(i, x, m) = true \Leftrightarrow \Phi_i(x) = m, \ \forall i, x, m$$

Complexity Measures (2)

TIME

Using the Turing machine as the model of computation, let TIME $= \{TIME_i\}$, where $TIME_i(x)$ denotes the number of steps executed by computation $T_i(x)$. If $T_i(x)$ does not halt, then $TIME_i(x)$ is considered to be undefined.

SPACE

SPACE must be defined carefully so that $SPACE_i(x)$ is undefined whenever $Ti(x)$ does not halt. If $T_i(x)$ does halt, after reading at most h(x) cells on any of the k tapes, it will halt in $N = |\Sigma|^{h(x)k} h(x)^k |Q|$ steps at most. N is the number of distinct global states of Ti(x). (An upper bound on space implies an upper bound on time.)

Nondeterministic TIME and SPACE

For nondeterministic Turing machine the time measure is $NTIME_i$, where $NTIME_i(x)$ is:

1. The number of steps in the shortest accepting computation path of $NT_i(x)$, if one exists and all paths halt.
2. The number of steps in the shortest computation path of $NT_i(x)$, if all paths halt and reject.
3. Undefined, otherwise.

NSPACE is similarly defined.

Classes of Languages

Languages can be grouped into *classes*. A class C is the set of all languages L over alphabet Σ that satisfy a predicate π.

$$C = \{L | L \subseteq \Sigma^* \wedge \pi(L)\}$$

The complement of a class C, coC, is defined as

$$coC = \{L | L \subseteq \Sigma^* \wedge \pi(L^c)\}$$

Example

The class T of tally languages is defined as $C = \{L | L \subseteq \Sigma^* \wedge \pi(L)\}$ where $\Sigma = \{0, 1\}$ and $\pi(L) = true \Leftrightarrow L \subseteq 0^*$. What is coT?

Resource-bounded Reducibility

A generalisation of reducibility is resource-bounded reducibility, $\leq_r$, where the reducibility function f can be computed in some predefined resource bounds. Given a class of languages C, and language L is said to be C-complete with respect to $\leq_r$ if $L \in C \wedge L' \leq_r L, \forall L' \in C$.

A class C is said to closed with respect to reducibility $\leq_r$ if for any pair of languages L_1 and L_2 such that $L_1 \leq_r L_2, L_2 \in C \Rightarrow L_1 \in C$.

Lemma

Let C_1 and C_2 be two classes of languages such that $C_1 \subset C_2$, and C_1 is closed with respect to a reducibility $\leq_r$. Then any language L that is C_2-complete with respect to $\leq_r$ does not belong to C_1.

The Class **P**

If each step of an algorithm requires only 10^{-9}s to execute then:

size n	n^2	n^3	2^n	3^n
10	0.1 μs	1 μs	1μs	59 μs
30	0.9 μs	27 μs	1s	2.4 days
50	2.5 μs	0.125ms	13 days	$2.3x10^7$ years

Therefore for practical reasons we consider all problems solved by algorithms that require a hyperpolynomial ($f(x) \geq n^k, \forall k$) number of steps to be computationally intractable.

The set of tractable problems belongs to the set of problems whose language is accepted by some deterministic Turing machine in a polynomial-bounded number of steps. Or formally, the languages that encode tractable problems belong to the class **P**, which is defined as

$$\mathrm{P} = \bigcup_{k \geq 0} DTIME[n^k]$$

Examples of Class **P** Languages

Greatest Common Divisor (GCD)

The greatest common divisor (GCD) of two integers a and b is defined to be that largest integer that divides both a and b. We can derive an algorithm for GCD by observing that if r is the remainder of a divided by b ($a \geq b$), then the common divisors of a and b coincide with the common divisors of b and r.

```
function GCD (a:integer, b:integer):integer
begin
   if b = 0 then GCD = a else GCD = GCD (b, a mod b)
end
```

Complexity analysis : Since $a_k = q * b_k + b_{k+1}$ and $b_{k-1} = a_k$ we have $b_{k-1} > b_k + b_{k+1} > 2b_k$.
Then $b = b_0 > 2b_1 > 4b_2 > 8b_3 > ... > 2^k$
and thus $k < \log b$.
Therefore the time complexity of GCD is linear with respect to the size of the input length.

Examples of Class **P** Languages (2)

Shortest Path

The Shortest Path Problem consists of determining whether a path of length of at most k exists between a pair of nodes n_1 and n_2 of an input graph G.

The following algorithm makes use of a dynamic programming technique. Denote by n the number of node of G and let $A^h(i,j)$ be the length of the shortest path from node i to node j going through no node with an index greater than h. Then,

$$A^{h+1}(i,j) = \min(A^{h+1}(i,h+1) + A^{h+1}(h+1,j), A^{h+1}(i,j))$$

that is, either a shortest path from i to j going through no node of index greater than $h+1$ passes through node $h+1$ or it does not. By construction, the entry $A^n(n_1,n_2)$ yields the value of the shortest path between n_1 and n_2. If the value is not greater than k, then the SHORTEST PATH instance admits a yes answer.

Examples of Class **P** Languages (3)

begin {input: G, n_1, n_2, k}
  n = number of nodes of G;
  for all $i, j \leq n$ do
    if G includes an edge $\langle i,j \rangle$
      then $A^1(i,j) = 1$
      else $A^1(i,j) = \infty$;
  for $h = 2$ to n do
    for $i \leq n$ do
      for $j \leq n$ do
        $A^h(i,j) = \min(A^{h-1}(i,h) + A^{h-1}(h,j), A^{h-1}(i,j))$;
  if $A^n(n_1,n_2) \leq k$ then accept else reject;
end

Because of the nested for loops, clearly the number of steps required by the algorithm is $O(n^3)$.

Examples of Class **P** Languages (4)

2-Satisfiability Problem

Given a set of boolean variables $x_1, x_2, \ldots, x_n$, a literal is either one of the variables x_i or the negation of the variable, $\overline{x_i}$. A *clause* is a disjunction (logical or) of a subset of the boolean variables. Each boolean variable is assigned a truth value, true or false. A clause is true is one of the literals in the clause is true otherwise the clause is false. A set of clauses is satisfiable is there exists an assignment of truth values to the variables that makes all the clauses true.

The satisfiability problem (SAT): Given a set of clauses, does there exist a set of truth values, one for each variable, such that every clause is satisfiabe?

The 2-Satisfiability problem (2-SAT): Given a Boolean formula f that is a conjunction of clauses containing exactly 2 literals, does an assignment of truth values satisfying f exist?

Examples of Class **P** Languages (5)

Algorithm:

The objective is to guess the value of an arbitrary variable and deduce the consequences of this guess for the other variables of f.

Initially all clauses are declared unsatisfied. A starting variable x is selected arbitrarily with x and $\overline{x}$ receiving the values true and false respectively. The assignment is then extended to as many variables as possible by repeated applications of the following step.

Take an arbitrary unsatisfied clause $(l_h \vee l_k)$. If one of the two literals has the value true then declare the clause satisfied. Otherwise if one of the literals, say l_h, has the value false, then assign to l_k and $\overline{l_k}$ the values true and false respectively. Declare the clause $(l_h \vee l_k)$ satisfied.

Examples of Class **P** Languages (6)

3 exclusive cases may occur:

1. During the execution of any of the steps, a conflict takes place were a literal is assigned the value true which was previously assigned false. This means the initial guess of the literal value was wrong and all the steps starting from the assignment of the values true to x are cancelled. This time x and $\overline{x}$ receive the values false and true respectively and the assignment procedure starts again. If a second conflict occurs, the algorithm terminates and f is declared unsatisfied.
2. No conflict occurs and all variables receive a value. Then the formula f is satisfiable.
3. No conflict occurs but some variables remain unassigned. We may ignore the clauses already satisfied. Clearly the reduced formula is satisfiable if and only if the previous one is also satisfiable. A guess is then made for an arbitrary variable and the assignment procedure is again applied to the reduced formula.

Examples of Class **P** Languages (7)

```
begin {input: f}
  C = set of clauses of f ; declare the clauses of C unsatisfiable;
  V = set of variables of f ; declare the variables of V unassigned;
  while V contains a variable x do
    begin
      assign true to x; firstguess = true;
      while C contains an unsatisfied clause
        c = (l_h ∨ l_k) with at least one assigned literal do
          begin
            if l_h = true ∨ l_k = true then declare c satisfied;
            else if l_h = false ∧ l_k = false then
              begin
                if not firstguess then reject
                else
                  begin
                    declare the clauses of C unsatisfied and the variables of V unassigned;
                    assign false to x; firstguess = false;
                  end
              end
            else if l_h = false then assign true to l_k else assign false to l_k;
          end
            delete from C the satisfied clauses and V the assigned variables;
    end
  accept;
end
```

Polynomial-time Intractable Problems

The vast majority of problems that belong to the class **P** are computationally tractable, but there problems in **P** which are not tractable.

Problems in **P** are bounded by cn^k steps. If, for a given problem, c and k are very large then the problem is practically intractable.

Also there exists a small set of problems for which we can prove that the problem can be solved in a polynomial number of steps, but we can not describe the Turing machine which decides its language.

Polynomial-time Reducibility

How do we prove that an algorithm belongs to **P**? One way is to count the number of steps in the algorithm; another is to use the concept of polynomial reducibility.

Given two languages L_1 and L_2, L_1 is polynomially reducible to L_2, $L_1 \leq L_2$, if a function $f \in \mathbf{FP}$ exists such that $x \in L_1 \leftrightarrow f(x) \in L_2$.

Lemma

Let L_1 and L_2 be two languages such that $L_1 \leq L_2$ and $L2 \in \mathbf{P}$. Then $L_1 \in P$.

Proof.

Add a Turing transducer that computes the reduction to a Turing acceptor that decide L_2. The composition of two polynomials is still a polynomial. □

Polynomial-time Reducibility (2)

Lemma

Let B be any language included in **P** *such that $B \neq \emptyset$ and $B \neq \Sigma^*$. Then, for any language A included in* **P**, $A \leq B$.

Proof.

Let y be a word in B and z be a word in B^c. Then define a function f as

$$f(x) = \begin{cases} y, & \text{if } x \in A \\ z, & \text{otherwise} \end{cases}$$

The function f is polynomial-time computable since A is polynomial-time decidable, and $x \in A \leftrightarrow f(x) \in B$ is the definition of polynomial reducibility. □

The class **P** is closed with respect to reducibility.

The Class **NP**

By replacing the deterministic Turing machines by nondeterministic Turing machine, we can define another class of languages (and hence problems). The class **NP** is defined as:

$$\text{NP} = \bigcup_{k \geq 0} NTIME(n^k), k \geq 0$$

Since deterministic Turing machines are a restriction of nondeterministic Turing machines, $P \subseteq NP$. General consensus has it that **NP** is richer than **P**, because nondeterministic Turing machines are more "powerful" than deterministic ones since they execute several computation paths at the same time. A nondeterministic polynomial-time algorithm can be thought of as an algorithm that always chooses the correct alternative from many and runs in polynomial time.

The Class **NP** (2)

Is a nondeterministic Turing machine "more powerful" than a deterministic Turing machine?

Theorem

Given any k-tape Turing machine T with $k > 1$, it is always possible to derive an equivalent 1-tape Turing machine T that simulates the first t steps of T in $O(t^2)$ steps.

Proof.

We dedicate k consecutive cells of the single tape of T to storing the set of symbols contained in the i-th cells ($i = 0, 1, \ldots$) of the k tapes of T.
The k tape heads of T are simulated by making use of suitable marked symbols. For example, if the h-th tape head of T scanned a symbol a then the tape of T will contain the marked symbol a_h.
By convention, after a quintuple of T has been simulated the tape head of T is assumed to be positioned on the leftmost marked symbol.

The Class **NP** (3)

Proof (contd.)

The simulation proceeds as follows. First the k marked symbols are scanned sequentially rightwards to determine the quintuple of T to be simulated. Next, the tape is scanned k times rightwards to replace the old symbols with the new ones and to simulate the k tape moves. Assume that a left (or right) move must be simulated relative to tape h. This is accomplished by shifting the tape head k positions to the left (or right) and by replacing the symbol currently scanned, say a, with a marked symbol a_h. Since the tape is "semi-infinite" we need to be careful moving to the left. To prevent running past the left end of the tape we use a special end of tape symbol, say [, so that whenever to be marked is a [, T shifts back k positions to the right. A left move from cell 0 is replaced by a not moving the tape head.

The Class **NP** (4)

Proof (contd.)

The following configurations illustrate an example with 2 tapes where the tape heads are positioned on symbols c and f. The quintuple to be simulated replaces c with x, replaces f with y and performs a left move on tape 1 and a right move on tape 2.

	Before	After
tape 1	$[abq_icd\Delta$	$[abxq_jd\Delta$
tape 2	$[eq_ifgh\Delta$	$[q_jeygh\Delta$

The single tape simulation of this machine has the following configurations.

Before	After
$[[aebq_rf_2c_1gdh\Delta\Delta$	$[[aq_se_2byxgd_1h\Delta\Delta$

□

The Class **NP** (5)

Theorem

Given a nondeterministic Turing machine NT, it is always possible to derive an equivalent deterministic machine T. Furthermore, $\exists c$ such that if NT accepts x in t steps, then T will accept x in, at most, c^t steps.

Proof.

For any input x the aim is to systematically visit the computation tree associated with $NT(x)$ until T finds an accepting computation path (if it exists).

T needs to be careful since the computation tree may include computation paths of an infinite length. A simple depth-first search of the tree might end up following an infinite path and never get around to considering a finite accepting path.

The Class **NP** (6)

Proof (contd.)

The solution to this problem is to use a breadth-first search. T visits all the computation paths for, at most, one step and if an accepting state is encountered then T accepts x. Otherwise we visit all the computation paths for, at most, two steps, and so on. If NT accepts x then T will eventually accept x.
We still need to specify how to visit all the computation paths for, at most, i steps, for $i = 1, 2, \ldots$. Since we are now dealing with finite trees we can do a depth-first search. Since r is the degree of nondeterminism of NT then there are at most r^i partial computation paths. With each path we can associate a word with maximum length i over the alphabet $\Gamma = 1, 2, \ldots, r$. Let Γ^i represent all the words of length i over Γ. A deterministic Turing machine can generate all such words, in lexicographical order, in a linear number of operations with respect to i.

The Class **NP** (7)

Proof (contd.)

Then starting from the initial state T can simulate the execution path, pw, associated with each word w. Three situations can occur:

1. If pw does not represent a valid sequence of steps of NT then T moves to the next word in Γ^i.
2. If the first j symbols of $w, (0 \leq j \leq i)$, induce a sequence of steps that cause NT to accept x, then T also accepts x.
3. If the first j symbols of $w, (0 \leq j \leq i)$, induce a sequence of steps that cause NT to reject x, then T moves onto the next word in Γ^i.

The Class **NP** (8)

Proof (contd.)

Simulating *pw* also requires a linear number of steps with respect to i. Thus over all computation paths i steps can be visited in at most r^i steps where r is the degree of nondeterminism. Therefore if NT accepts x in t steps, then the total time spent by T in simulating NT is bounded by

$$\sum_{i=1}^{t} r^i \leq c^t$$

where c is a suitable constant.

□

Therefore a nondeterministic Turing machine doesn't "solve any more problems" than a deterministic Turing machine, it just does it more efficiently.

Checking solutions in **NP**

Theorem

A language L belongs to **NP** *if and only if a language* $L_{check} \in$ P *and a polynomial p exist such that*

$$L = \{x | \exists y . x, y \in L_{check} \wedge y \leq p(x)\}$$

Proof.

If $L = \{x | \exists y . x, y \in L_{check} \wedge y \leq p(x)\}$ where $L_{check} \in P$ and p is a polynomial, then the following nondeterministic algorithm decides L in polynomial time.

```
begin
   guess y in the set of words of length, at most, p(|x|);
   if < x, y >∈ L_check then accept;
     else reject;
end
```

Checking solutions in **NP** (2)

Proof (contd.)

Conversely, let L be a language in **NP**. Then a nondeterministic Turing machine NT exists that decides L in polynomial time. It is easy to verify that, for any x, each computation path of $NT(x)$ can be encoded into a word of length, at most, $p(|x|)$ where p is a polynomial. The language L_{check} is then defined as follows.
$\langle x, y \rangle \in L_{check}$ if and only if y encodes an accepting computation path of $NT(x)$.
Therefore $L_{check} \in$ P and for any x

$$x \in L \leftrightarrow \exists y.x, y \in L_{check} \wedge y \leq p(x)$$

□

This theorem tells us that **NP** is the class of problems for which it is easy to check the correctness of a claimed answer. We are not asking for a way to find a solution, only to verify that an alleged solution is correct.

Examples of Class **NP** Languages

Satisfiability Problem

Satisfiability is a generalisation of 2-Satisfiability introduced earlier. Here each clause C_i may include any number of literals rather than exactly two as in 2-SAT.

Solution: A nondeterministic algorithm for Satisfiability can be obtained by guessing any of the $2n$ assignments of value to the n variables of f and verifying whether it satisfies f:

```
begin {input: f }
   guess t in the set of assignments of values
                to the n variables of f;
   if t satisfies f then accept;
     else reject;
end
```

Examples of Class **NP** Languages (2)

Traveling Salesman Problem (TSP)

Given a completed weighted graph G and a natural number k, does a cycle exist passing through all the nodes of G such that the sum of the weights associated with the edges of the cycle is, at most, k?

Solution: A polynomial-time nondeterministic algorithm for the Traveling Salesman Problem (TSP) can be obtained by guessing any of the n! permutations of n nodes and verifying whether it corresponds to a cycle whose cost is, at most, k.

Examples of Class **NP** Languages (3)

3-Colourability Problem

Given an undirected graph G does there exists a way to color the nodes red, green, and blue so that no adjacent nodes have the same color?

Solution: A polynomial nondeterministic algorithm for 3-Colourability can be obtained by guessing any of the 3^n colourings of a graph of n nodes and checking if it has the required property.

Examples of Class **NP** Languages (4)

We can reduce the 3-Colourability problem to the Satisfiability problem. Consider a graph G of four vertices, and the decision problem "Can the vertices of G be coloured in 3 colours such that no edge connects vertices of the same colour?".

We will use 12 Boolean variables. The variable $x_{i,j}$ corresponds to the assertion that vertex i has been coloured in the colour j $(i = 1, 2, 3, 4; j = 1, 2, 3)$.

We can construct the 3-Colourability problem as an instance of the Satisfiability problem with 34 clauses. The first 16 clauses are:

$$\begin{aligned} C(i) &= \{x_{i,1}, x_{i,2}, x_{i,3}\} \\ T(i) &= \{\overline{x}_{i,1}, \overline{x}_{i,2}\} \\ U(i) &= \{\overline{x}_{i,1}, \overline{x}_{i,3}\} \\ V(i) &= \{\overline{x}_{i,2}, \overline{x}_{i,3}\} \end{aligned}$$

Examples of Class **NP** Languages (5)

The 4 clauses $C(i)$ assert that each node has been coloured in at least one colour (though possibly more). The clauses $T(i)$ state that no node has colour 1 and colour 2. Similarly the clauses $U(i)$ state that no node has colour 1 and colour 3, and the clauses $V(i)$ state that no node has colour 2 and colour 3. Taken all together these 16 clauses state that each node has been coloured by one and only one colour.

The remaining clauses ensure that the two endpoints of an edge do not have the same colour. We define for each edge e of the graph G and colour $j(= 1, 2, 3)$ a clause $D(e, j)$ as follows. Let u and v be the endpoints of e, then

$$D(e, j) = \{x_{u,j}, x_{v,j}\}$$

asserts that not both endpoints of an edge have the same colour.

Examples of Class **NP** Languages (6)

The original instance of the 3-Colourability problem has now been reduced to an instance of the Satisfiability problem. Specifically, does there exist an assignment of value true and false to the 12 Boolean variables $x_{1,1}, ..., x_{4,3}$ such that each of the 34 clauses contains at least one literal whose value is true if and only if each pair of nodes connected by an edge of the graph G have different colours? The graph is 3-Colourable if and only if the clauses are satisfied.

If we have an algorithm to solve the Satisfiability problem, then we can also solve the 3-Colourability problem.

NP-complete Languages

Are all problems in NP equally hard, or are some more difficult than others? Are there problems in NP which are the most difficult?

Completeness Revisited

Given a complexity class C and a language L, for any language $L \in C$ we derive a reduction f which, given a model of computation M which decides L and an input x, computes a word $f(M, x)$ such that M accepts x if and only if $f(M, x) \in L$.
L is reducible to L and since L is an arbitrary language in C, L is C-complete.

Informally, we say that a decision problem in **NP**-complete if it belongs to **NP** and every problem in **NP** is quickly reducible to it. Or formally, is there a class of languages in **NP** which can be polynomially-time reduced to any other language in **NP**?

Implications of **NP**-completeness

Suppose we could prove that a certain decision problem Q is **NP**-complete. Then we could concentrate our efforts on finding a polynomial-time algorithm for Q. If we succeed in finding a polynomial-time algorithm for all instances of Q then we could derive a fast algorithm for every problem in **NP**.

Conversely, if we could prove that it is impossible to find a polynomial-time algorithm for some particular problem R in **NP**, then we cannot find a fast algorithm for any **NP**-complete problem either.

Is there any **NP**-complete problems? Stephen Cook and Leonid Levin independently proved that *Satisfiability* was **NP**-complete in 1971 and 1973 respectively.

Cook-Levin Theorem

Theorem

Satisfiability *is NP-complete.*

Proof.

We want to prove that *Satisfiability* is **NP**-complete, i.e. that every problem in **NP** is polynomially reducible to an instance of Satisfiability.

Let Q be some problem in **NP** and let I be an instance of problem Q. Since Q is in **NP** there exists a nondeterministic Turing machine that recognises encoded instances of the problem Q in polynomial time.

Let TMQ be such a Turing machine and let $P(n)$ be a polynomial in its argument n with the property that TMQ recognises every input x in time $P(n)$, where x is a word in the language Q and n is the length of x.

Cook-Levin Theorem (2)

Proof (contd.)

We intend to construct an instance $f(I)$ of Satisfiability (SAT), corresponding to each word I in the language Q, for which the answer to the decision problem "are all the clauses simultaneously satisfiable" is Yes. Conversely, if the word I is not in the language Q, the clauses will not be satisfiable.

To construct an instance of SAT means that we are going to define a number of variables, literals and clauses in such a way that the clauses are satisfiable if and only if x is in the language Q, i.e. the machine TMQ accepts x.

What we must do then is to express the accepting computation of the nondeterministic Turing machine as the simultaneous satisfaction of a number of logical propositions. It is precisely here that the relatively simplicity of a Turing machine allows us to enumerate all of the possible paths to an accepting computation.

Cook-Levin Theorem (3)

Proof (contd.)

Now we will describe the Boolean variables that will be used in the clauses under construction.

$Q_{i,k}$ is true if after step i of the checking calculation it is *true* that the Turing machine TMQ is in state q_k, and *false* otherwise.

$S_{i,j,a} = \{\text{after step } i, \text{ symbol } a \text{ is in tape square } j\}$.

$T_{i,j} = \{\text{after step } i, \text{ the tape head is positioned over square } j\}$.

Let's count the variables that we've just introduced. Since the Turing machine TMQ does its accepting calculation in time $\leq P(n)$, it follows that the tape head will never venture more than $P(n)$ squares away from its starting position. Therefore the subscript j, which runs through the various tape squares that are scanned during the computation, can assume only $O(P(n))$ different values.

Cook-Levin Theorem (4)

Proof (contd.)

Index a runs over the letters in the alphabet that the machine can read, so it can assume at most some fixed number A of values.

The index i runs over the steps of the accepting computation, and so it takes at most $O(P(n))$ different values.

Finally, k indexes the states of the Turing machine, and there is only some fixed finite number, say K, of states that TMQ might be in. Hence there are altogether $O(P(n)^2)$ variables, a polynomial number of them.

The remaining task is to described precisely the conditions under which a set of values assigned to the variables listed above actually defines a possible accepting calculation for x . Then when set of satisfying values of the variables is found they will determine a real accepting calculation of the machine TMQ.

Cook-Levin Theorem (5)

Proof (contd.)

This will be done by requiring that the number of clauses be all true (satisfied) at once, where each clause will express one necessary condition.

1. At each step the machine is in at least one state.

Hence at least one of the K available state variables must be true. This leads to the first set of clauses, one for each step i of the computation.

$$\{Q_{i,1}, Q_{i,2}, \ldots, Q_{i,K}\}$$

Since i assumes $O(P(n))$ values, there are $O(P(n))$ clauses.

Cook-Levin Theorem (6)

Proof (contd.)

2. At each step the machine is not in more than one state.

Therefore, for each step i, and each pair (j', j'') of distinct states, the clauses

$$\{\overline{Q}_{i,j'}, \overline{Q}_{i,j''}\}$$

must be true. These are $O(P(n))$ additional clauses to add to the list.

Cook-Levin Theorem (7)

Proof (contd.)

3. At each step, each tape square contains exactly one symbol from the alphabet of the machine.

This leads to two lists of clauses which require:

1. That there is at least one symbol in each square at each step.
2. That there are not two symbols in each square at each step.

The clauses that do this are:

$$\{S_{i,j,1}, S_{i,j,2}, \ldots, S_{i,j,A}\}$$

where A is the number of letters in the machine's alphabet, and

$$\{\overline{S}_{i,j,k'}, \overline{S}_{i,j,k''}\}$$

for each step i, square j and pair (k', k'') of distinct symbols in the alphabet of the machine. This gives rise to $O(P(n)^2)$ additional clauses.

Cook-Levin Theorem (8)

Proof (contd.)

4. At each step the tape head is positioned over a single square.

First, for each step, the tape head is positioned on at least one square:

$$\{T_{i,1}, T_{i,2}, \ldots, T_{i,P(n)}\}$$

Second, for each step, the tape head is not positioned on two or more squares.

$$\{\overline{T}_{i,j'}, \overline{T}_{i,j''}\}$$

This gives another $O(P(n)^2) + O(P(n)^3) = O(P(n)^3)$ clauses.

Cook-Levin Theorem (9)

Proof (contd.)

5. Initially the machine in in state 0, the tape head is over square 1 and the input string x is in squares 1 to n.

$$\{Q_{1,0}\}$$
$$\{T_{1,1}\}$$
$$\{S_{1,1,x_1}\}$$
$$\{S_{1,2,x_2}\}$$
$$\ldots$$
$$\{S_{1,n,x_n}\}$$

Cook-Levin Theorem (10)

Proof (contd.)

6. At step $\leq P(n)$ the machine is in state q_Y.

$$\{Q_{1,q_Y}, Q_{2,q_Y}, \ldots, Q_{P(n),q_Y}\}$$

7. At each step the machine moves to its next global state (state, symbol, head position) in accordance with the application of its program module to its previous (state, symbol).

To find the clauses that will do this job consider first the following condition: the symbol in square j of the tape cannot change during step i of the computation if the tape head is not positioned on it at that moment.

Cook-Levin Theorem (11)

Proof (contd.)

This translates into the collection:

$$\{T_{i,j}, \overline{S}_{i,j,k}, S_{i+1,j,k}\}$$

of clauses, one for each triple (i, j, k) = (state, square, symbol). These clause express the condition that, at time i,

- either the tape head is positioned over square j ($T_{i,j}$ is *true*)
- or else the head is not positioned there, in which case
  - either symbol k is not in the jth square before the step
  - or else symbol k is (still) in the jth square after the step is executed.

Cook-Levin Theorem (12)

Proof (contd.)

It remains to express the fact that transitions from one global state of the machine to the next are the direct results of the operation of the program module, The three sets of clauses that do this are:

$$\{\overline{T}_{i,}, \overline{Q}_{i,k}, \overline{S}_{i,j,l}, T_{i+1,j+INC}\}$$
$$\{\overline{T}_{i,j}, \overline{Q}_{i,k}, \overline{S}_{i,j,l}, Q_{i+1,k'}\}$$
$$\{\overline{T}_{i,j}, \overline{Q}_{i,k}, \overline{S}_{i,j,l}, S_{i+1,j,l'}\}$$

Each clause can be interpreted:

"either the tape head is not positioned on square j, or the present state is not q_k or the symbol just read is not l, but if they are then ..."

Cook-Levin Theorem (13)

Proof (contd.)

There is a clause (as above) for each step $i = 0, \ldots, P(n)$ of the computation, for each square $j = 1, \ldots, P(n)$ of the tape, for each symbol l in the alphabet, and for each possible state q_k of the machine. In total a polynomial number of clauses. The new global state triple (k, l, INC) is as computed by the program.

Now we have constructed a set of clauses with the following property. If we execute a recognising computation on a string x, in at most $P(n)$ time, then this computation determines a set of (*true*, *false*) values for all the variables listed previously, in such a way that all of the clauses just constructed are simultaneously satisfied.

Cook-Levin Theorem (14)

Proof (contd.)

Conversely, if we have a set of values of the SAT variables that satisfy all of the clauses at once, then the set of values of the variables that would cause *TMQ* to do a computation that would recognise the string x. It also describes, in minute detail, the ensuing accepting computation that *TMQ* would do if it were given x.

Hence every language in **NP** can be reduced to SAT. It is not difficult to check through the above construction and prove that the reduction is accomplished in polynomial time. It follows that SAT is **NP**-complete.

□

P $\neq$ **NP**?

The fact that $P \subseteq NP$ is without doubt, but the really interesting question is:

Is P properly included in NP?

This $P \neq NP$ conjecture was first raised nearly 50 years ago and still not answered. We know that any nondeterministic Turing machine can be transformed into a deterministic Turing machine, but this machine will require an exponential number of steps. A major leap forward would be a proof, or disproof, of this conjecture.

$\mathbf{P} \neq \mathbf{NP}$? (2)

Reasons why we believe the $P \neq NP$ conjecture.

- No one has been able to find a polynomial-time algorithm for any of the over 3000 **NP**-complete problems
- Complexity theorists have developed a rich hierarchy of complexity classes, many of which would collapse together if $\mathbf{P} = \mathbf{NP}$
- There would be "no fundamental gap between solving a problem and recognizing the solution once it's found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss..." (Scott Anderson, MIT)

However, as yet, there is no proof that $\mathbf{P} \neq \mathbf{NP}$.