Decision Problems

As

$$\log_a n = \log_a b \times \log_b n$$

$\log_\alpha n$ and $\log_\beta n$ are the same when it comes to O

Definition: Time complexity

For any function f, we say that the time complexity of a decidable language $\mathcal{L}$ is $O(f)$, or $\mathcal{L}$ is decidable in $O(f)$ time, if there exists a TM T which decides $\mathcal{L}$, and constants n_0 and c such that for all inputs x with $|x| > n_0$

$$Time_T(x) \leqslant c \cdot f(|x|)$$

1 Complexity Classes

Definition: Time complexity class TIME[f]

The class of all problems for which there exists an algorithm with time complexity in $O(f)$

This is sometimes called DTIME[f] - for deterministic time

2 The complexity class P

Definition: P

$$\mathbf{P} = \bigcup_{k \geq 0} TIME\left[n^k\right]$$

The class P is a reasonable mathematical model of the class of problems which are tractable or solvable in practice

However, the correspondence is not exact:

- When the degree of the polynomial is high then the time grows so fast that in practice the problem is not solvable
- The constants may also be very large

3 Different models of computation

Lemma
We can simulate t steps of k-tape TM with an equivalent one tape TM in $O[t^2]$ steps

Lemma
We cans simulate t steps of a two-way infinite k-tape machine with an equivalent k-tape TM in $O[t]$ steps

Hence the class P is the same for all of these models of computation (and many others)

4 Different encodings

Lemma
For any number n, the length of the encoding of n in base b_1 and the length of the encoding of n in base b_2 are related by a constant factor (provided $b_1, b_2 \geqslant 2$)

Hence the class P is the same for these encodings (and many others)

5 Proving a problem is in P

- The class P is said to be robust - it doesn't depend on the exact details of the computational model or encoding so we don't need to specify all the details of the machine model or even the encoding
- The most direct way to show that a problem is in P is to give a polynomial time algorithm which solves it
- Even a naive polynomial time algorithm often provides a good insight into how the problem can be solved efficiently
- To find such an algorithm we generally need to identify an approach to the problem that is considerably better than brute-force search

6 CNF

Some of the most important computational problems concern logical formulas

A logical formula f is said to be in conjunctive normal form (CNF) if

$$f = C_1 \wedge ... \wedge C_m$$

where each C_i is a clause, which is a disjunction (OR) of literals

$$C_i = l_{i1} \vee ... \vee l_{ik}$$

and a literal is either a variable or its negation

Definition: k-CNF

If a logical formula in CNF has at most k literals per clause then it is in k-CNF

7 Satisfiability

Definition: Satisfiable

The logical formula f is satisfiable if there exists an assignment of True and False to the variables of f which makes f true

- f is True iff all the clauses are True
- A clause is True iff at least one literal is true

Problem: Satisfiability

Instance - CNF formula f
Question - Is f satisfiable?

Problem: k-Satisfiability

Instance - k-CNF formula f
Question - Is f satisfiable?

7.1 2 Satisfiability

Proposition - 2-Satisfiability is in P

Proof:

1. Declare all clauses unsatisfied and literals unassigned
2. Select an arbitrary unassigned variable x and assign x the value True and $\neg x$ the value False
3. Select an unsatisfied clause $l_i \vee l_j$
   (a) If both literals are unassigned, ignore the clause
   (b) If at least one literal is assigned True, declare the clause satisfied
   (c) If both are False, restart the algorithm setting x false and $\neg x$ True. If a conflict occurs again, declare unsatisfiable
   (d) If one literal is False and the other unassigned, set the other to True and its negation to False, and declare the clause satisfied
4. Repeat step 3 until either
   (a) All clauses are satisfied, return satisfiable
   (b) Or all clauses remaining not satisfied (yet) have all their variables unassigned. In this case return to step 2

8 Polynomial-time reducibility

Another way to show that a problem is in P is to use a reduction

Informally, a problem P is reducible to a problem Q if we can somehow use methods that solve Q in order to solve P

Definition: Polynomially reducible

A language $\mathcal{L}_1$ is polynomially reducible to $\mathcal{L}_2$, denoted $\mathcal{L}_1 \leqslant \mathcal{L}_2$, if a polynomial-time computable function f exists such that

$$x \in \mathcal{L}_1 \Leftrightarrow f(x) \in \mathcal{L}_2$$

Lemma

$$\mathcal{L}_1 \leqslant \mathcal{L}_2 \text{ and } \mathcal{L}_2 \in P \Rightarrow \mathcal{L}_1 \in P$$

Main idea - The composition of polynomials is a polynomial

Proof

- Let A_2 be a polynomial-time algorithm that decides L_2
- Let f be a polynomial-time reduction algorithm from L_1 to L_2
- We construct a polynomial-time algorithm A_1 that decides L_1
  1. Given input $x \in \{0,1\}^*$, compute $f(x)$ in polynomial time (we know that $x \in L_1 \Leftrightarrow f(x) \in L_2$)
  2. Use algorithm A_2 to decide whether $f(x) \in L_2$
  3. If $f(x) \in L_2$ then output YES; otherwise output NO

9 k-Colourability

- Let $G = (V, E)$ be a graph, with vertices V and edges E
- Recall that a function $f : V \to \{1, ..., n\}$ is a colouring if adjacent vertices are assigned different values (colours)

Problem: k-Colourability

Instance - A graph G
Question - Is there a colouring of G using at most k colours?

9.1 2-Colourability $\leqslant$ 2-Satisfiability

We can reduce 2-Colourability to 2-Satisfiability

- For each vertex v_i of the graph we create a variable x_i
- For each edge (v_i, v_j) we add two clauses $(x_i \vee x_j)$ and $(\neg x_i \vee \neg x_j)$

This translation of a 2-colourability problem to a 2-satisfiability problem is computable in polynomial time. now we check if it satisfies the reducibility condition:

$\Rightarrow$ If the graph is 2-colourable, use 2-colouring to assign truth values to variables (one colour is true, the other false)

- If the formula is satisfiable, define the 2-colouring by setting true variables to colour 1 and false to colour 2. If two adjacent vertices get the same colour then one of the associated clauses is not satisfied (contradiction). Thus we have a 2-colouring