Non-deterministic Turing machines

ian.mcloughlin@gmit.ie

Non-deterministic Turing machine

The usual Turing machines are often called deterministic Turing machines.

Deterministic Turing machines have exactly one row in their state table for every combination of (non-terminal) state and tape symbol.

This means there is only one path to follow at a given point in time.

Nondeterministic Turing machines can have any number of rows for each state/symbol (including none).

Essentially they allow for parallel computation – they can branch into two or more paths at the same time.

Non-deterministic Turing machine and languages

Languages are accepted by non-deterministic Turing machines, where an input string is accepted if any branch ends in the accept state.

Deciders – if a non-deterministic Turing machine always halts on all branches of computation, no matter what the input, then we say it decides the language it accepts.

Any language that is accepted (or decided) by a non-deterministic Turing machine has some deterministic Turing machine that accepts (or decides) it. So non-deterministic Turing machines don't really have any extra abilities over deterministic ones.

Non-deterministic polynomial time

Definition
A decision problem is in the NP complexity class if it is decidable by a non-deterministic Turing Machine in polynomial time.

P is a subset of NP
Note that every determinisitic Turing machine is also a non-deterministic one, by our definitions. The P complexity class is a subset of NP because of this.

Equivalent definition
An equivalent definition of NP that you may come across is that NP is the set of languages A that can be verified in polynomial time. By verified we mean that a deterministic Turing machine can accept a language $\{wc\}$ where w is in A and c is some string, called the certificate for w.

NP-complete problems

Definition
A problem is NP-hard if each problem in NP can be reduced to it in polynomial time.

Reduction
Reduction is a way of converting one problem into another, so that a solution to one is a solution to the other. By reducing decision problem A to decision problem B, we mean that we can transform inputs to A into inputs to B in such a way that a given input to A is accepted iff the corresponding input to B is.

Definition
A problem is NP-complete if it's in NP and is NP-hard.

Subset sum problem

Problem
Given a set of integers S, is there a non-empty subset whose elements sum to zero?

Example
Does $\{1, 3, 7, -5, -13, 2, 9, -8\}$ have such a subset?

Note
If somebody suggests a solution, it is very quick to check it. Being able to quickly verify a solution is a characteristic of NP problems.

SAT motivation

SAT	is short for Boolean SATisfiability.
It is	the archetypal NP-complete problem.
Informally	it asks if there is a way to efficiently decide if an expression containing a bunch of Boolean variables combined using ands, ors, nots and brackets has any setting of the variables that makes the expression true.

SAT instance example

Consider the expression $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge \neg x_1$, where x_1, x_2 and x_3 are true/false variables. Is there any setting of the variables that makes the expression true?

Propositional logic

Literals are Boolean variables (can be True or False), and their negations. They are represented by lower case letters like a and x_i.

Clause are expressions based on literals, that evaluate as True or False based on the literals. We use not, and and or on the literals. We'll sometimes call them expressions.

Not is depicted by $\neg$. So "not a" is denoted by $\neg a$.

Or is depicted by $\vee$. So "a or b" is denoted by $a \vee b$.

And is depicted by $\wedge$. So "a and b" is denoted by $a \wedge b$.

Normal forms

CNF A clause is in Conjunctive Normal Form if it is a "conjunction of disjunctions": $(a \vee b) \wedge (\neg a \vee c) \wedge d$.

DNF A clause is in Disjunctive Normal Form if it is a "disjunction of conjunctions": $(a \wedge b) \vee (\neg a \wedge c) \vee d$.

Converting to CNF and DNF
Every Boolean expression can be converted to CNF, and every Boolean expression can also be converted to DNF.

Four laws

The following four laws can be used to convert expressions to CNF and DNF. The first two are known as De Morgan's laws, and the latter two are called the distributivity laws.

Conversion laws

$$\neg(a \vee b) = \neg a \wedge \neg b$$

$$\neg(a \wedge b) = \neg a \vee \neg b$$

$$c \wedge (a \vee b) = (c \wedge a) \vee (c \wedge b)$$

$$c \vee (a \wedge b) = (c \vee a) \wedge (c \vee b)$$

Boolean Satisfiability Problem (SAT)

SAT is the set of all Boolean expressions that are satisfiable.

Satisfiable expressions have some setting of their constituent variables that causes them to evaluate as true.

The SAT decision problem is the problem of deciding which Boolean expressions are in SAT.

Note that it's easy to verify that a given setting of the variables in an expression make it true.

k-SAT

k-SAT is like SAT except that all expressions must be in CNF and each clause must be a disjunction of k literals.

2-SAT example: $(a \vee b) \wedge (\neg c \vee d) \wedge \ldots$

2-SAT is not NP-complete. There are polynomial time algorithms that solve it.

3-SAT example: $(a \vee b \vee c) \wedge (\neg c \vee d \vee \neg a) \wedge \ldots$

3-SAT is NP complete.

3-SAT is NP-Complete

Reduction

3-SAT is a special case of SAT, so 3-SAT must be in NP. We can reduce SAT to 3-SAT in polynomial time. First take the expression and convert it to a CNF expression (in polynomial time). Then we just need to convert each clause into a CNF expression with 3 literals per clause.

Suppose we have a clause with 1 literal: a. Convert this to $(a \vee u_1 \vee u_2) \wedge (a \vee u_1 \vee \neg u_2) \wedge (a \vee \neg u_1 \vee u_2) \wedge (a \vee \neg u_1 \vee \neg u_2)$.
Suppose we have a clause with 2 literals: $a \vee b$. Convert this to $(a \vee b \vee u_1) \wedge (a \vee b \vee \neg u_1))$.

Now suppose we have a clause with n literals: $a \vee b \vee c \vee \ldots$.
Convert this to
$(a \vee b \vee u_1) \wedge (c \vee \neg u_1 \vee u_2)) \wedge \ldots \wedge (i \vee \neg u_{n-4} \vee u_{n-3}) \wedge (j \vee k \vee \neg u_{n-3})$.

SUBSETSUM and NP

SUBSETSUM is NP-complete.

2^n is the number of subsets. Note that 2^n is also the number of settings of n Boolean variables.

The correspondence can be seen in terms of 0's and 1's. In SUBSETSUM the elements from the set that are included in a given subset are represented by 1's.

The usual proof that SUBSETSUM is NP-complete is done by reduction to 3-SAT.