COMPUTER TUTORIAL

ALL ABOUT COMPUTER CAN FIND HERE

MONDAY, APRIL 28, 2008

Computability theory

In computer science, **computability theory** is the branch of the theory of computation that studies which problems are computationally solvable using different models of computation.

Computability theory differs from the related discipline of computational complexity theory, which deals with the question of how efficiently a problem can be solved, rather than whether it is solvable at all.

INTRODUCTION

A central question of computer science is to address the limits of computing devices. One approach to addressing this question is understanding the problems we can use computers to solve. Modern computing devices often seem to possess infinite capacity for calculation, and it's easy to imagine that, given enough time, we might use computers to solve any problem. However, it is possible to show clear limits to the ability of computers, even given arbitrarily vast computational resources, to solve even seemingly simple problems. Problems are formally expressed as a decision problem which is to construct a mathematical function that for each input returns either `0` or `1`. If the value of the function on the input is `0` then the answer is "no" and otherwise the answer is "yes".

To explore this area, computer scientists invented automata theory which addresses problems such as the following: Given a formal language, and a string, is the string a member of that language? This is a somewhat esoteric way of asking this question, so an example is illuminating. We might define our language as the set of all strings of digits which represent a prime number. To ask

whether an input string is a member of this language is equivalent to asking whether the number represented by that input string is prime. Similarly, we define a language as the set of all palindromes, or the set of all strings consisting only of the letter 'a'. In these examples, it is easy to see that constructing a computer to solve one problem is easier in some cases than in others.

But in what real sense is this observation true? Can we define a formal sense in which we can understand how hard a particular problem is to solve on a computer? It is the goal of computability theory of automata to answer just this question.

FORMAL MODELS OF COMPUTATION

In order to begin to answer the central question of automata theory, it is necessary to define in a formal way what an automaton is. There are a number of useful models of automata. Some widely known models are:

Deterministic finite state machine
: Also called a deterministic finite automaton (DFA), or simply a finite state machine. All real computing devices in existence today can be modeled as a finite state machine, as all real computers operate on finite resources. Such a machine has a set of states, and a set of state transitions which are affected by the input stream. Certain states are defined to be accepting states. An input stream is fed into the machine one character at a time, and the state transitions for the current state are compared to the input stream, and if there is a matching transition the machine may enter a new state. If at the end of the input stream the machine is in an accepting state, then the whole input stream is accepted.

Nondeterministic finite state machine
: Similarly called a nondeterministic finite automaton (NFA), it is another simple model of computation, although its processing sequence is not uniquely determined. It can be interpreted as taking multiple paths of computation simultaneously through a finite number of states. However, it is proved that any NFA is exactly reducible to an equivalent DFA.

Pushdown automaton
: Similar to the finite state machine, except that it has available an execution stack, which is allowed to grow to arbitrary size. The state transitions additionally specify whether to add a symbol to the stack, or to remove a symbol from the stack. It is more powerful than a DFA due to its infinite-memory stack, although only some information in the stack is ever freely accessible.

Turing machine
: Also similar to the finite state machine, except that the input is provided on an execution "tape", which the Turing machine can read from, write to, or move back and forth past its read/write "head". The tape is allowed to grow to arbitrary size. The Turing machine is capable of performing complex calculations which can have arbitrary duration. This model is perhaps the most important model of computation in computer science, as it simulates computation in the absence of predefined resource limits.

Multi-tape Turing machine
: Here, there may be more than one tape; moreover there may be multiple heads per tape. Surprisingly, any computation that can be performed by this sort of machine can also be performed by an ordinary Turing machine, although the latter may be slower or require a larger total region of its tape.

POWER OF AUTOMATA

With these computational models in hand, we can determine what their limits are. That is, what classes of languages can they accept?

Power of finite state machines

Computer scientists call any language that can be accepted by a finite state machine a **regular language**. Because of the restriction that the number of possible states in a finite state machine is finite, we can see that to find a language that is not regular, we must construct a language that would require an infinite number of states.

An example of such a language is the set of all strings consisting of the letters 'a' and 'b' which contain an equal number of the letter 'a' and 'b'. To see why this language cannot be correctly recognized by a finite state machine, assume first that such a machine M exists. M must have some number of states n. Now consider the string x consisting of $(n + 1)$ 'a's followed by $(n + 1)$ 'b's.

As M reads in x, there must be some state in the machine that is repeated as it reads in the first series of 'a's, since there are $(n + 1)$ 'a's and only n states by the pigeonhole principle. Call this state S, and further let d be the number of 'a's that our machine read in order to get from the first occurrence of S to some subsequent occurrence during the 'a' sequence. We know, then, that at that second occurrence of S, we can add in an additional d (where $d > 0$) 'a's and we will be again at state S. This means that we know that a string of $(n + d + 1)$ 'a's must end up in the same state as the string

of $(n + 1)$ 'a's. This implies that if our machine accepts x, it must also accept the string of $(n + d + 1)$ 'a's followed by $(n + 1)$ 'b's, which is not in the language of strings containing an equal number of 'a's and 'b's.

We know, therefore, that this language cannot be accepted correctly by any finite state machine, and is thus not a regular language. A more general form of this result is called the Pumping lemma for regular languages, which can be used to show that broad classes of languages cannot be recognized by a finite state machine.

Power of pushdown automata

Computer scientists define a language that can be accepted by a pushdown automaton as a **Context-free language**, which can be specified as a **Context-free grammar**. The language consisting of strings with equal numbers of 'a's and 'b's, which we showed was not a regular language, can be decided by a push-down automaton. Also, in general, a push-down automaton can behave just like a finite-state machine, so it can decide any language which is regular. This model of computation is thus strictly more powerful than finite state machines.

However, it turns out there are languages that cannot be decided by push-down automaton either. The result is similar to that for regular expressions, and won't be detailed here. There exists a Pumping lemma for context-free languages. An example of such a language is the set of prime numbers.

Power of Turing machines

Turing machines can decide any context-free language, in addition to languages not decidable by a push-down automaton, such as the language consisting of prime numbers. It is therefore a strictly more powerful model of computation.

Because Turing machines have the ability to "back up" in their input tape, it is possible for a Turing machine to run for a long time in a way that is not possible with the other computation models previously described. It is possible to construct a Turing machine that will never finish running (halt) on some inputs. We say that a Turing machine can decide a language if it eventually will halt on all inputs and give an answer. A language that can be so decided is called a **recursive language**. We can further describe Turing machines that will eventually halt and give an answer for any input in a language, but which may run forever for input strings which are not in the language. Such Turing machines could tell us that a given string is in the language, but we may never be sure based on its behavior that a given string is not in a language, since it may

run forever in such a case. A language which is accepted by such a Turing machine is called a **recursively enumerable language**.

The Turing machine, it turns out, is an exceedingly powerful model of automata. Attempts to amend the definition of a Turing machine to produce a more powerful machine are surprisingly met with failure. For example, adding an extra tape to the Turing machine, giving it a 2-dimensional (or 3 or any-dimensional) infinite surface to work with can all be simulated by a Turing machine with the basic 1-dimensional tape. These models are thus not more powerful. In fact, a consequence of the Church-Turing thesis is that there is no reasonable model of computation which can decide languages that cannot be decided by a Turing machine.

The question to ask then is: do there exist languages which are recursively enumerable, but not recursive? And, furthermore, are there languages which are not even recursively enumerable?

The halting problem

> *Main article: Halting problem*

The halting problem is one of the most famous problems in computer science, because it has profound implications on the theory of computability and on how we use computers in everyday practice. The problem can be phrased:

> *Given a description of a Turing machine and its initial input, determine whether the program, when executed on this input, ever halts (completes). The alternative is that it runs forever without halting.*

Here we are asking not a simple question about a prime number or a palindrome, but we are instead turning the tables and asking a Turing machine to answer a question about another Turing machine. It can be shown (See main article: Halting problem) that it is not possible to construct a Turing machine that can answer this question in all cases.

That is, the only general way to know for sure if a given program will halt on a particular input in all cases is simply to run it and see if it halts. If it does halt, then you know it halts. If it doesn't halt, however, you may never know if it will eventually halt. The language consisting of all Turing machine descriptions paired with all possible input streams on which those Turing machines will eventually halt, is not recursive. The halting problem is therefore called non-computable or **undecidable**.

An extension of the halting problem is called Rice's Theorem, which states that it is undecidable (in general) whether a given language possesses any specific nontrivial property.

Beyond recursive languages

The halting problem is easy to solve, however, if we allow that the Turing machine that decides it may run forever when given input which is a representation of a Turing machine that does not itself halt. The halting language is therefore recursively enumerable. It is possible to construct languages which are not even recursively enumerable, however.

A simple example of such a language is the complement of the halting language; that is the language consisting of all Turing machines paired with input strings where the Turing machines do *not* halt on their input. To see that this language is not recursively enumerable, imagine that we construct a Turing machine M which is able to give a definite answer for all such Turing machines, but that it may run forever on any Turing machine that does eventually halt. We can then construct another Turing machine M' that simulates the operation of this machine, along with simulating directly the execution of the machine given in the input as well, by interleaving the execution of the two programs. Since the direct simulation will eventually halt if the program it is simulating halts, and since by assumption the simulation of M will eventually halt if the input program would never halt, we know that M' will eventually have one of its parallel versions halt. M' is thus a decider for the halting problem. We have previously shown, however, that the halting problem is undecidable. We have a contradiction, and we have thus shown that our assumption that M exists is incorrect. The complement of the halting language is therefore not recursively enumerable.

POSTED BY SOTONK AT 5:43 PM
LABELS: COMPUTER

0 COMMENTS:

Post a Comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)