Design

Michael P. Fourman

February 2, 2010

Introduction

The design of algorithms tends to be a rather problem-specific activity. In order to find efficient algorithms, it is important to look beyond 'obvious' solutions to problems, and also to notice when two problems are sufficiently similar that a solution for one can be used as the basis of a solution for the other. In this lecture, we survey some general approaches to algorithm design: first, ways of adding 'higher level' organisation to algorithms using techniques that are applicable when solving many different kinds of problems; and second, ways of systematically introducing 'detail level' organisation to algorithms. Finally, we examine a method that allows us to place a lower bound on the time required to solve a problem using *any* algorithm of a certain style.

Higher-level organisation of algorithms

Top-down algorithms for many problems can be derived by using the general strategy of *divide-and-conquer*. This type of algorithm works by partitioning a problem into smaller parts, solving the parts recursively, and then combining the solutions for the parts into a solution for the whole. While such algorithms are nearly always elegant and easy to analyse, they are sometimes found to be inefficient because of the fact that the sub-problems are solved independently, and hence may perform the same computation many times unnecessarily. Examples of divide-and-conquer algorithms seen so far include merge and quick sorting.

Bottom-up algorithms for many problems can be derived by using the general strategy of *dynamic programming*. This type of algorithm works by solving a collection of smaller sub-problems, and saving the results for later use in solving larger problems; eventually, this process leads to a solution

for the actual problem. Since sub-problems are not solved independently, this method ensures that the same computation is not repeated needlessly. Indeed, dynamic programming is usually only of use if there are many sub-problems that crop up repeatedly when solving a problem. Examples of dynamic programming algorithms seen so far include Floyd's and Dijkstra's shortest path algorithms.

Algorithms to find *optimal* solutions to problems typically go through a sequence of steps, with a set of choices at each step. A *greedy algorithm* always makes the choice that looks best at that moment. This simple approach is taken in the hope that a locally optimal choice will lead to a globally optimal solution. Such simplicity does not always work but, for many practical problems, the greedy approach is effective. Examples of greedy algorithms seen so far include Kruskal and Prim's minimum spanning tree algorithms. Dijkstra's shortest path algorithm has a dynamic programming structure overall, and uses a greedy strategy to choose the sequence of sub-problems solved.

The ubiquitousness of graphs means that standard techniques for exploring graphs can be used as general strategies for algorithms. A fundamental graph-related technique is that of searching for a vertex with an 'interesting property'. In the special case of arrays, which are really just directed lists, it is possible to do an instantaneous search if the interesting property is of the form "n-th vertex in the list". Normally, however, it is necessary to traverse the graph, for example, using depth-first or breadth-first search; the choice of traversal strategy depends on the representational rôle of the graph in the problem to be solved. Performing *topological sorting* of directed acyclic graphs is often useful because properties of the form "n-th vertex in the graph" can then be defined.

Lower-level organisation of algorithms

A simple style of algorithm design is the *straight line program*: this contains no conditional actions or loops, and so just consists of a sequence of expression evaluations in order to obtain a result. A richer style of algorithm design is *structured programming*, which means that, in addition to functions, only three main constructs are used to build algorithms:

- Sequence — to allow consecutive processing steps;
- Condition — to allow selective processing; and
- Repetition — to allow looping.

For example, in C terms, these correspond to the ";" separator, **if/switch** statements and **while/do/for** statements respectively; the crucial feature is that arbitrary jumps (gotos) are forbidden. In (purist) SML terms, sequence and repetition are not explicit but, roughly, the three constructs correspond to applying a function to the result of another function, **if/case** expressions and repetitive recursion respectively. (In fact, SML also has ";" and **while**, but that is another story.) The intention is that algorithm design is limited to a small number of predictable constructions, because evidence suggests that program complexity is thereby reduced, and so readability, testability and maintainability are enhanced. One common problem with dogmatic application of structured programming is that enabling premature exits from condition or repetition constructs often necessitates contortion: strictly speaking, constructs like C's **break** and **continue** should not be used. ML's exception mechanism provides a more structured solution to this problem.

Analysis of algorithms can be easier when structured programming is used. Analyses of run time can be made for each function; when recursion is used, recurrence relations can often be used to find the run time. It is possible to roughly estimate the run time of a sequence construct by summing the run times of its components, the run time of a condition construct by taking the maximum run time of its action components, and the run time of a repetition construct by multiplying the run time of its action component by the number of loops (or by some upper bound if the exact number is not known). This scheme is rather crude, particularly in the case of the selection construct, but if borne in mind when developing algorithms, helps to avoid obvious inefficiencies. Careful use of big-O notation throughout reduces complication.

An alternative style of algorithm design, appropriate in some circumstances, is a *decision tree* approach. In marked contrast to straight line program algorithms, all of the operations in decision tree algorithms are conditional decision-making actions; there is no transformation of data. A typical application of decision tree algorithms is in searching for an interesting item in a collection of data. As a general mechanism for organising such algorithms, it is sometimes possible to employ *decision tables* to derive a *table-driven* algorithm. The idea is that a list is made of all decisions that can be made during execution. Then, for each combination of decisions that can occur (assuming there are not too many), a rule is given to specify the action that the algorithm should take.

Lower bounds on algorithm run time

When an algorithm involves conditional actions (e.g., it might be a decision tree algorithm, or it might be a structured programming algorithm with condition constructs), it is possible to make assertions about the minimum run time that it will require.

Suppose that, for an input of size n, the algorithm can take $f(n)$ different sequences of conditional actions, depending on the actual input supplied. For example, a comparison-based dictionary searching algorithm applied to a dictionary of size n will have at least n different conditional sequences, each one leading to a different dictionary element.

If the conditional actions are all **if**-style, i.e., with Boolean results, we could represent different conditional sequences by an ordered bit string, one bit representing the result of each condition. If there are $f(n)$ different sequences, then we know that at least $\lg f(n)$ bits are needed to represent each one — otherwise two sequences would have the same representation. Thus, at least one execution path of the algorithm must involve $\lg f(n)$ conditional actions — otherwise it would be possible to code up all of the different conditional sequences using less than $\lg f(n)$ bits. Given this, we have obtained an *information-theoretic lower bound* on the time requirement of any algorithm using an approach based on conditional actions: at least $\lg f(n)$ time is required. This argument also works if the conditional actions have more than two choices at any stage. In the case of a comparison-based dictionary search, as above, this means that at least $\lg n$ time is required, and so the $O(\lg n)$ 2-3 tree and AVL tree searching algorithms in CS2/GB/L4 have the best possible (big-O) run time.

As another example, when we sort a collection of n randomly-ordered values, there are $n!$ different ways in which the collection might need to be sorted. Thus, a comparison-based algorithm will have at least $n!$ different conditional sequences. The information theoretic lower bound on run time is therefore

$$\lg n! \;>\; \lg\left(\frac{n}{e}\right)^n \;=\; n\lg n - n\lg e \;>\; \left(\frac{1}{2}\right)n\lg n \quad \text{for } n \geq 8$$

(the first inequality uses Stirling's approximation to $n!$ and the constant $e \approx 2.718$ is the base of natural logarithms). This lower bound on time complexity grows at the same rate as $n\lg n$, and so we see that our familiar $\mathrm{O}(n\lg n)$ comparison sorting algorithms, such as merge and quick sorting, cannot be bettered by any other comparison-based sorting methods.