Computational problems

David Keil, June 2010

Topic 1 of Analysis of Algorithms addresses problems, as distinguished from algorithms. Textbooks in the algorithms field present problems as vehicles for presenting algorithm examples. Some problems are solved by different algorithms, of different efficiencies. The grouping of classes of problems together, before discussing their solutions, has merit. This short paper provides discussion missing from current textbooks.

This presentation of the course called “Analysis of Algorithms” would more accurately be called, “Analysis of Algorithms, Interactive Processes, and Computational Problems.” Standard algorithms textbooks focus on algorithms, as opposed to problems, and on algorithms, as opposed to interaction.

Some textbooks, like (Johnsonbaugh, 2004), move in the direction of a broader approach, covering computational complexity, which is about problems, and parallel computing, which is partly about interaction among processors.

In general, research and writing about theoretical computer science remain in the algorithmic paradigm, which distinguishes computation (algorithms) from communication (interaction). Thus this course challenges the limitations of that paradigm while presenting the powerful results of research in algorithmic computing.

Complexity of algorithms and problems

In this course, we’re interested in analyzing the time performance or complexity of algorithms, and the complexity of problems. The complexity of a problem is defined as the performance of the fastest algorithm that solves the problem – not the fastest known algorithm, but the fastest algorithm, known or unknown. Some problems have infinite complexity, in the sense that no algorithm (finite transformation procedure) exists that solves the problem.

Some claims about certain problems can be proven, so we have significant theoretical or mathematical results about algorithmic problems.

When we speak of algorithmic problems, we mean problems that have specifications of finite input and finite output, where all the input must be provided before any output is expected. In software engineering, the specification, or analysis, stage of system development produces a specification of some amount of input, and some amount of output, with a fixed relationship between input and output. For a given input value, the same output value is expected. For that reason, algorithms are said to compute functions. Thus algorithmic computational problems may be said to be functional problems.

Algorithmic problems may be classified by the kind of data structure they operate on. These include representations of numbers; strings; objects; arrays; linked lists; matrices; sets of points \((x, y)\) on a coordinate axis; trees; and graphs.

Problems such as searching data structures and sorting arrays are familiar to computer science students.

Constraint problems

Some problems are not functional, in that their outputs may be any of a set of possible solutions to a problem specified by constraints, or restrictions. For example, a search problem may return the index of any array element that meets certain constraints; several of the array elements may satisfy the constraint.

A constraint problem may be converted to a decision problem (true/false) as follows. Rather than to search a large state space to find a value that satisfies a constraint, the problem is to determine whether a given parameter value satisfies the constraint.

Optimization problems

Another way to distinguish algorithmic problems is as optimization or non-optimization problems. An optimization problem requires finding a minimal or maximal-valued item that satisfies some constraint. Optimization problems tend to be hard.

Function optimization problems are those in which a solution is the parameter values for which the function returns a minimum or maximum value.
Two last categories of algorithmic problems are those that are \textit{intractable} and those that are \textit{undecidable}. The latter cannot be solved algorithmically in the general case; intractable problems can be solved but are thought to require time resources so great that the problems are not worth solving (see topic 8).

\textbf{Analysis of interactive problems}

Finally, two classes of computational problems may be identified that are \textit{non-algorithmic}. These are problems that are \textit{sequential-interactive} (services) or \textit{multi-stream interactive} (missions).

Sequential interactive problems involve a synchronized interlacing of input and output, with the possibility that the problem-solving system or agent can store previous inputs and the results of previous interactions, and that inputs can depend on previous outputs.

Multi-stream interactive problems are those in which interaction may be asynchronous (interrupting rather than taking turns) and in which three or more computing agents interact, including possibly the environment of two agents, taken as a computational entity. Thus the factor of \textit{indirect interaction} enters.

\textit{Services} may involve interaction of clients and servers, so that a service may be considered as provided to one client as isolated from others. Missions, on the other hand, may require the concurrent operation of multiple components, each interacting with active computing entities in the environment.

In solving algorithmic problems (computing functions), a system’s performance relative to size of input may be a factor. This is the concern of algorithm analysis. Time performance in the provision of services is likewise a concern in solving sequential-interactive problems. Here \textit{response time} and \textit{throughput} measure performance. To satisfy the time requirements of missions, multi-stream-interactive systems may be required to satisfy real-time processing constraints.

\textbf{Formal specifications}

Specification of algorithmic problems uses the tools of predicate logic to state the relation between the values of parameters and the values returned by the function computed by an algorithm. One formalism is \textit{Hoare triples}, after C. A. R. Hoare. A Hoare triple is a sequence $\langle \phi \rangle P \langle \psi \rangle$, where $\phi$ (phi) is a precondition on the input or parameter data, $P$ is a program, and $\psi$ (psi) is a postcondition, denoting the specification of the output of the algorithm. For example, for a sorting algorithm, a precondition is that the input to the algorithm is an array on whose elements an ordering relation exists, and a postcondition is that for every pair of consecutive array elements, the first is less than the second.

Formal specification of reactive (interactive) systems may be in a language of temporal logic, such as Computation Tree Logic (CTL). CTL’s operators include $G$ (globally, or always), $U$ (until), and $E$ (eventually). \textit{Always} $p$ means that the formula $p$ will hold at every instant on all execution paths; for example, the value of \textit{count} will always be greater than or equal to 0. Temporal logic formula $p U q$ means that formula $p$ will hold, on all paths, until $q$ holds. $E p$ is the negation of $A \neg p$.

Based on formal specifications, proofs of correctness of algorithms may be constructed. Correctness, or verification, of systems is a separate matter from problem specification.