Algorithm Analysis

Part 2

By Chuck Easttom

Introduction

This is the second of two papers for the Algorithm Analysis. In the first paper basic algorithms, methods for analyzing algorithms, and data structures were covered. This paper will focus on two topics. The first topic will be the design of algorithms. The second topic will be an examination of NP complete problems.

Designing Algorithms

Designing an algorithm is a formal process. Algorithms are developed to provide systematic ways of solving certain problems. Therefore it should be no surprise that there are systematic ways of designing algorithms. In this section we will examine a few of these methods.

Divide and Conquer

The Divide and Conquer approach to algorithm design is a commonly used approach. In fact it could be argued that it is the most commonly used approach. It works by recursively breaking down a problem into sub-problems of the same type, until a point is reached where these sub-problems become simple enough to be solved directly. Once the sub problems are solved, the solutions to the sub-problems are then combined to provide a solution to the original problem. In short you keep sub dividing the problem until you find managable portions that you can solve. Then after solving those smaller, more

managable sub-problems, you combine those solutions in order to solve the original problem.

When approaching difficult problems, such as the classic Tower of Hanoi puzzle, the Divide and Conquer method provides an efficient way to solve the problem. For many such problems the paradigm offers the *only* workable way to find a solution. The divide and conquer method often leads to algorithms that are not only effective, but that are also efficient.

The efficiency of the divide and conquer method can be examined by considering the number of sub-problems being generated. If the work of splitting the problem and combining the partial solutions is proportional to the problem's size n,then there are a finite number p of sub-problems of size ~ n/p at each stage. Furthermore, if the base cases require O(1) (i.e.constant-bounded) time, then the divide-and-conquer algorithm will have O(n log n) complexity. This is often used in sorting problems in order reduce the complexity from O(n^2). Recall from the first paper on algorithm analysis that an O of n log n is fairly good for most sorting algorithms.

In addition to allowing one to devise efficient algorithms for solving complex problems, the divide and conquer approach is well suited for execution in multi-processor machines. The reason for this is that the sub-problems can be assigned to different processors in order to allow each processor to work on a different subproblem. This leads to sub-problems being solved simultaneously thus increasing the overall efficacy of the process.

However no method is without disadvantages. One commonly argued disadvantage of a divide-and-conquer approach is that the recursion process can be quite slow. The

overhead of the repeated sub-routine calls, along with the overhead of storing the state at each point in the recursion, can outweigh the advantages of the approach, at least in some instances. Whether or not the disadvantages outweigh the advantages depends upon the implementation style. If you start with large enough recursive base cases, the overhead of recursion can become negligible for many problems.

One problem with the divide and conquer approach is that for simple problems it may be more complicated than an iterative approach. For example if the problem is to add N numbers, then a simple loop to add them up in sequence is much easier to program than a divide-and-conquer approach that breaks the set of numbers into halves, adds them recursively, and then adds the sums. This means that the divide and conquer approach is best reserved for more complex problems where a simple iterative approach is either ineffective or inefficient.

Of course, an algorithm can be designed by a divide and conquer approach and then implemented in an iterative manner. Any recursive algorithm can be simulated in an iterative style, but by reordering the computation it is sometimes possible to avoid simulating a recursive call stack altogether. A good expample of this is the Cooley-Tukey FFT algorithm. This algorithm was initially designed in a divide and conquer style. However it is often implemented in an interative style.

Practical Application: The Quick Sort

The quick sort (discussed in the first paper) is an excellent example of the divide and conquer approach. In this method a series of items that need to be sorted is divided into sub lists and each sub list is sorted, then the sublists are recombined. This C source code

shows the quick sort in action (note that comments are in yellow highlight to explain how certain key portions of this code use the divide and conquer approach):

```
void swap(int *a, int *b) { int t=*a; *a=*b; *b=t; }
void sort(int arr[], int beg, int end)
{
  if (end > beg + 1)
  {
   Note: The pivot point is a dividing point, thus a clear example of
the divide and conquer method.
    int piv = arr[beg], l = beg + 1, r = end;
    while (l < r)
    {
    Note: This sub section of the problem is solved (i.e. the half of
the problem below the pivot point).  Later this portion will be
combined with the other portion that was also sorted.  As you can see
the overall sorting problem is divided into two smaller problems and
solved, then the sub problems are combined.
      if (arr[l] <= piv)
        l++;
      else
        swap(&arr[l], &arr[--r]);
    }
    swap(&arr[--l], &arr[beg]);
    sort(arr, beg, l);
    sort(arr, r, end);
  }
}
```

You can see in this source code that a pivot point is selected, and the list is split based on that pivot point. Then each sub-list is sorted. This is a classic example of the divide and conquer approach.

Dynamic Programming

Dynamic programming is a method for reducing the runtime of algorithms. Put another way it is a method for improving the efficiency of algorithms. This method works with algorithms that have overlapping sub-problems and optimal sub-structure. Dynamic programming usually takes one of two approaches:

- Top-down approach: In this approach, the problem is broken into sub-problems. Then each of these sub-problems are solved and the solutions stored, in case they need to be solved again. This is similar to the divide and conquer approach.
- Bottom-up approach: With this approach the sub-problems that might be needed are solved in advance and then used to build up solutions to larger problems.

In essence this method is about solving an optimization problem by caching sub problem solutions, rather than re-computing them.

A Practical Example: Floyds Algorithm

An excellent example of the Divide and Conquer method is Floyd's algorithm. In 1967 Robert Floyd invented an algorithm for finding the shortest path between two graph vertices. Interestingly, I found references to this algorithm in games programming resources. It appears that this algorithm has a number of interesting applications.

Floyd's all-pairs shortest-path algorithm creates a matrix *S* in *N* steps In each step k, it constructs an intermediate matrix I(k). In any programming language the matrix, and the intermediate matrices, can be represented by arrays. The intermediate matrices I(k)

contain the best-known shortest distance between each pair of nodes. Initially each row is set to the length of the edge, if the edge exists, and to ∞ otherwise. Each *k*th step of the algorithm considers the rows and determines whether the best-known path from v_i to v_j is longer than the combined lengths of the best-known paths from v_i to v_k and from v_k to v_j. If so, the entry is updated to reflect the shorter path This comparison operation is performed a total of N^3 times. This means that we can approximate the cost of this algorithm as tN^3 where t is the cost of a single comparison operation.

Source Code Example

The following code represents a c language implementation of Floyd's algorithm. Again there are notes in yellow highlight in the code that will explain how this code sample illustrates the dynamic programming approach.

```
int GraphAlgo::ComputeFloydAPSP(int *C, int n, int *A)
{
  int i,j,k;
  // set all connected positions to 1
  // and all unconnected positions to infinity
  for (i=0; i<n; i++)
  {
    for (j=0; j<n; j++)
    {
      if ( *(C+i*n+j) == 0)
      {
        *(A+i*n+j) = 999999999;
```

```
    }
    else
    {
      *(A+i*n+j) = 1;
    }
  }
 }

 // set the diagonals to zero
 for (i=0; i<n; i++)
 {
  *(A+i*n+i) = 0;
 }
```

Note the following segment is what most appropriately illustrates the dnamic programming approach.

```
 // for each route via k from i to j pick
 // any better routes and replace A[i][j]
 // path with sum of paths i-k and j-k
 for (k=0; k<n; k++)
 {
  for (i=0; i<n; i++)
  {
  for (j=0; j<n; j++)
```

```
   {
```

Note the pointers used here. Integer pointers work well for describing matrices (often many programmers prefer them to arrays).

```
     if ( *(A+i*n+k) + *(A+k*n+j) < *(A+i*n+j) )
       {
         // A[i][j] = A[i][k] + A[k][j];
                    *(A+i*n+j) = *(A+i*n+k)+ *(A+k*n+j);
       }
     }
   }
  }
  return 0;
} // Floyd's algorithm
// this is for testing Floyd's algorithm
// demonstrates the allocation and
// deallocation of memory for the matrices
void FloydTest()
{

  // allocate the entire matrix in one linear array
  // trying to allocate it as an array of pointers to arrays
  // didn't quite work, possibly because the [] and * notation
  // aren't quite replaceable
```

```
 int n=4;
```

Note in this portion of the code actual arrays are used to define matrices.

```
 int *C=new int[n*n];
   C[0]=0; C[1]=1; C[2]=0; C[3]=0;
   C[4]=1; C[5]=0; C[6]=1, C[7]=0;
   C[8]=0; C[9]=1; C[10]=0;C[11]=1;
   C[12]=0;C[13]=0;C[14]=1;C[15]=0;


 int* A = new int[n*n];
 ComputeFloydAPSP (C,n,A);
 printf("Final shortest distances\n");
 for(int i=0;i<n;i++)
 {
   for(int j=0;j<n;j++)
   {
     printf("%d ",*(A+i*n+j));
   }
   printf("\n");
 }
 printf("End of All pairs Shortest paths\n");
   delete [] A;
   delete [] C;


} // end of FloydTest
```

Greedy Approach

The textbook *An Introduction to Algorithms*, defines greedy algorithms as those that select what appears to be the most optimal solution in a given situation. In other words a solution is selected that is ideal for a specific situation, but not be the most effective solution for the broader class of problems. The greedy approach is used with optimization problems. In order to give a precise description of the greedy paradigm we must first consider a more detailed description of the environment in which most optimization problems occur. In most optimization problems, one will have the following:

- A collection of candidates. That collection might be a set, list, array or other data structure. How the collection is stored in memory is irrelevant.
- A set of candidates which have previously been used.
- A predicate solution that is used to test whether or not a given set of candidates provide an efficient solution. This does not check to see if those candidates provide an optimal solution, just whether or not they provide a working solution.
- Another predicate solution (feasible) to test if a set of candidates can be extended to a solution.
- A selection function, which chooses some candidate which has not yet been used.
- A function which assigns a value to a solution.

Essentially, an optimization problem involves finding a subset S from a collection of candidates C where that subset satisfies some specified criteria. For example the criteria may be that it is a solution such that the function is optimized by S. Optimized may denote any number of factors, such as minimized or maximized. Greedy methods are

distinguished by the fact that the selection function assigns a numerical value to each candidate C and chooses that candidate for which:

SELECT(C) is largest

or SELECT(C) is smallest

All Greedy Algorithms have this same general form. A Greedy Algorithm for a particular problem is specified by describing the predicates, and the selection function. Consequently, Greedy Algorithms are often very easy to design for optimization problems.

The General Form of a Greedy Algorithm is as follows:

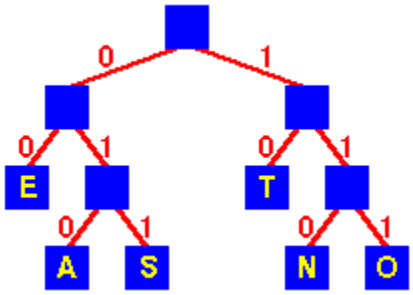
```
function select (C : candidate_set) return candidate;
function solution (S : candidate_set) return boolean;
function feasible (S : candidate_set) return boolean;
```

A Practical Example: Huffman Code

Huffman code is an answer to the question of what is the minimum number of bits that can be used to encode some text. One practical application for this algorithm is text compression. Huffman's approach uses a table of frequency of occurrence for each symbol. This table is derived from input data. For example, the frequency of occurrence of letters in normal English might be derived from processing a large number of text documents. That table could then be used for encoding all text documents.

The next step would be to assign a variable-length bit string to each character that unambiguously represents that character. This means that the encoding for each character must have a unique prefix, if the characters to be encoded are arranged in a binary tree.

The following table illustrates this (note this was found on a website http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/huffman.html):



Encoding tree for ETASNO

An encoding for each character is found by following the tree from the route to the character in the leaf: the encoding is the string of symbols on each branch followed.

For example:

String	Encoding
TEA	10 00 010
SEA	011 00 010
TEN	10 00 110

Using a greedy approach places our n characters in n sub-trees and starts by combining the two least weight nodes into a tree which is assigned the sum of the two leaf node weights as the weight for its root node. The time complexity of the Huffman algorithm is O(n log n). Frequently one will see a heap used to store the weight of each tree. In that case each iteration requires O(log n) time to determine the lowest weight and insert the new weight. There are O(n) iterations, one for each item.

The Greedy Approach compared to Dynamic Programming

Each approach is appropriate for different situations. The Dynamic Programming approach is contingent upon having a solution to a sub-problem already. The Greedy approach is not. Therefore if you have such a sub-problem solution available then you should at least consider using the Dynamic Programming approach. However if you

don't, then you must use the greedy approach. It should also be noted that the Greedy approach may not be as effective at producing the most optimal solution as the dynamic approach. The greedy approach simply provides you with a locally optimal solution, not necessarily the most optimal solution, nor necessarily a solution with more widespread applicability.

NP Complete Problems

The problems we studied in the first paper where all solvable via some polynomial equation. However not all problems are solvable in this fashion. Those that are not, are referred to as NP, or 'Not Polynomial'. In complexity theory, NP-complete problems are the most difficult problems to solve. Since there is no polynomial equation for their solution, they are much more difficult to solve

An interesting thing to note is that to date no one has proven that there is no polynomial solution for any NP complete problem. And it should be further noted that if one could provide a polynomial solution for *any* NP complete problem, that this would prove that all NP-complete problems do have a polynomial solution (in short it would prove that there is no such thing as an NP complete problem). The complexity class consisting of all NP-complete problems is sometimes referred to as NP-C. We have essentially three classes of problems. P problems are those that can be solved in polynomial time. These are the sorts of algorithms examined in the first paper. Then we have NP problems which cannot be solved in polynomial time. NP problems cannot be solved in polynomial time, but can be verified that a given solution is correct. NP complete problems simply cannot be addressed in any fashion in polynomial time.

One example of an NP-complete problem is the subset sum problem. That problem can be summarized as follows: given a finite set of integers, determine whether any non-empty subset of them sums to zero. A proposed answer is easy to verify for correctness, however to date no one has produced a faster way to solve the problem than to simply try each and every single possible subset. This is very inefficient.

At present, all known algorithms for NP-complete problems require time that is superpolynomial in the input size. In otherwords there is no polynomial solution for any NP complete problem Therefore, to solve an NP-complete problem one of the following approaches is used:

- **Approximation**: An algorithm is used that quickly finds a suboptimal solution that is within a certain known range of the optimal one. In other words and approximate solution is found.
- **Probabilistic**: An algorithm is used that provably yields good average runtime behavior for a given distribution of the problem instances
- **Special cases**: An algorithm is used that is provably fast if the problem instances belong to a certain special case. In other words an optimal solution is found only for certain cases of the problem.
- **Heuristic**: An algorithm that works well on many cases, but for which there is no proof that it is always fast. Heuristic methods are also considered 'rule of thumb'.

We have examined NP complete problems in a general way, but it is now time to come to a formal definition.A problem *C* is NP-complete if

1. it is in NP and
2. every other problem in NP is reducible to it.

In this instance, reducible means that for every problem *L*, there is a polynomial-time many-one reduction, a deterministic algorithm which transforms instances $l \in L$ into instances $c \in C$, such that the answer to *c* is yes if and only if the answer to *l* is yes. As a consequence of this definition, if you have a polynomial time algorithm for *C*, we could solve all problems in NP in polynomial time. This means that if someone ever finds a polynomial solution for any NP complete problem then all have a solution, as was mentioned earlier in this paper.

In the text *Introduction to Algorithms*, NP complete problems are illustrated by showing a problem that can be solved with a polynomial tiam algorithm, along with a related NP complete problem. For example that finding the shortest path is a polynomial time algorithm, whereas finding the longest path is an NP complete problem.

This definition was proposed by Stephen Cook in 1971. Cook also showed that the Boolean satisfiability problem is NP-complete. Since Cook's original results, thousands of other problems have been shown to be NP-complete by reductions from other problems previously shown to be NP-complete; many of these problems are collected in Garey and Johnson's, 1979 book *Computers and Intractability: A Guide to NP-completeness*.

What about problems that are not quite NP complete, but meet one of the two criteria? A problem that satisfies condition 2 but not condition 1 is said to be NP-hard. So we have a range of problems that are NP, but not necessarily NP-complete.

A practical Example: The Knapsack problem

The Corman textbook discusses a number of NP complete problems including finging the longest simple path, the Hamiltonian cycle problem, the circuit satisfiability problem,

the clique problem, the vertex-cover problem, the traveling salesman problem, the subset sum problem, and the graph coloring problem. However it does not address the knapsack problem**.** It should be noted that the subset sum problem is considered a special case of the knapsack problem. Therefore the two problems are related, but are not the same problem.

The knapsack problem is encountered in in combinatorics, cryptography, and other areas. The name derives from the scenario of choosing items to fit inside a knapsack when you can only carry a finite weight but wish to carry the greates value of times. The problem is this: given a set of items *S*, each with a cost *C* and a value *V*, determine how many of each item to include in a collection so that the total cost *T* is less than some given cost *G* and the total value is as large as possible. The additional constraint is that each item *Si* can only be included either entirely or not at all. It cannot be divided up and included fractionally. The decision problem form of the knapsack problem is the question "can a value of at least *V* be achieved without exceeding the cost *C*?" .

Another way to describe this problem is this:

Consider a knapsack of volume V and n piles (j=1,2,...,n) of items. Each pile contains a number of items having the same weight (w_j) and volume (v_j).

The objective is to determine how many items from each pile (x_j) should be placed in the knapsack so as to maximize the total weight of the knapsack - obviously without exceeding the total volume of the knapsack (V).

However this problem is worded the key is to maximize value, and minimize cost. This particular problem has many practical applications in the real world. A formal (i.e.

mathematical) definition of this problem is as follows: (note this definition was found at http://www.nist.gov/dads/HTML/knapsackProblem.html):

There is a knapsack of capacity $c > 0$ and N items. Each item has value $v_i > 0$ and weight $w_i > 0$. Find the selection of items ($\delta_i = 1$ if selected, 0 if not) that fit, $\sum_{i=1}^{N} \delta_i w_i \leq c$, and the total value, $\sum_{i=1}^{N} \delta_i v_i$, is maximized.

There is no polynomial time algorithm that solves this problem, and it is a problem with clear practical applications. Frequently problems of this sort are solved using dynamic programming. This means that the various sub-problems of the primary problem are solved, then the ideal solution of that set of solutions is selected.

This problem is considered an NP problem for several reasons. The fact that no polynomial time algorithm has yet been found to solve this problem, by definition makes it an NP problem. However the real question is why no such algorithmic solution has been found, or is likely to be found. When you consider this problem, note that the issue is to find an optimal solution in a very specific situation. These criteria make a generalized polynomial algorithm impossible to find. Considering the nature of this problem you can see that dividing it into small sub-problems and building a up to an overall solution (as in the dynamic programming model) can be the only method for solving this problem.