Algorithm Design Paradigms

*Algorithm Design Paradigms:* General approaches to the construction of efficient solutions to problems. Such methods are of interest because:

- They provide templates suited to solving a broad range of diverse problems.
- They can be translated into common control and data structures provided by most high-level languages.
- The temporal and spatial requirements of the algorithms which result can be precisely analyzed.

Although more than one technique may be applicable to a specific problem, it is often the case that an algorithm constructed by one approach is clearly superior to equivalent solutions built using alternative techniques.

http://www.csc.liv.ac.uk/~ped/teachadmin/algor/algor.html

Paul E. Dunne,
Department of Computer Science
University of Liverpool

Main paradigms (some of them you have seen already):

- Brute force
- Greedy algorithms
- Divide-and-Conquer
- Decrease-and-Conquer
- Dynamic programming
- Transform-and-Conquer
- Genetic algorithms
- Backtracking, Branch and Bound

**Brute Force**

*Brute force* is a straightforward approach to solve a problem based on the problem’s statement and definitions of the concepts involved. It is considered as one of the easiest approach to apply and is useful for solving small – size instances of a problem.
Some examples of brute force algorithms are:
- Computing $a^n$ ($a > 0$, $n$ a nonnegative integer) by multiplying $a \times a \times \ldots \times a$
- Computing $n!$
- Selection sort
- Bubble sort
- Sequential search
- Exhaustive search: Traveling Salesman Problem, Knapsack problem.

**Greedy Algorithms** "take what you can get now" strategy

The solution is constructed through a sequence of steps, each expanding a partially constructed solution obtained so far. At each step the choice must be locally optimal – this is the central point of this technique.

**Examples:**
- Minimal spanning tree
- Shortest distance in graphs
- Greedy algorithm for the Knapsack problem
- The coin exchange problem
- Huffman trees for optimal encoding

Greedy techniques are mainly used to solve optimization problems. They do not always give the best solution.

**Example:**
Consider the knapsack problem with a knapsack of capacity 10 and 4 items given by the <weight:value> pairs: <5:6>, <4:3>, <3:5>, <3:4>. The greedy algorithm will choose item1 <5:6> and then item3 <3:5> resulting in total value 11, while the optimal solution is to choose items 2, 3, and 4 thus obtaining total value 12.

It has been proven that greedy algorithms for the minimal spanning tree, the shortest paths, and Huffman codes always give the optimal solution.

**Divide-and-Conquer, Decrease-and-Conquer**
These are methods of designing algorithms that (informally) proceed as follows:

Given an instance of the problem to be solved, split this into several smaller sub-instances (of the same problem), independently solve each of the sub-instances and then combine the sub-instance solutions so as to yield a solution for the original instance.

With the divide-and-conquer method the size of the problem instance is reduced by a factor (e.g. half the input size), while with the decrease-and-conquer method the size is reduced by a constant.

**Examples of divide-and-conquer algorithms:**
- Computing $a^n$ ($a > 0$, $n$ a nonnegative integer) by recursion
Binary search in a sorted array (recursion)
Mergesort algorithm, Quicksort algorithm (recursion)
The algorithm for solving the fake coin problem (recursion)

Examples of decrease-and-conquer algorithms:
- Insertion sort
- Topological sorting
- Binary Tree traversals: inorder, preorder and postorder (recursion)
- Computing the length of the longest path in a binary tree (recursion)
- Computing Fibonacci numbers (recursion)
- Reversing a queue (recursion)

The issues here are two:
1. How to solve the sub-instance
2. How to combine the obtained solutions

The answer to the second question depends on the nature of the problem. In most cases the answer to the first question is: using the same method. Here another very important issue arises: when to stop decreasing the problem instance, i.e. what is the minimal instance of the given problem and how to solve it.

When we use recursion, the solution of the minimal instance is called “terminating condition”

**Dynamic Programming**

One disadvantage of using Divide-and-Conquer is that the process of recursively solving separate sub-instances can result in the same computations being performed repeatedly since identical sub-instances may arise.

The idea behind *dynamic programming* is to avoid this pathology by obviating the requirement to calculate the same quantity twice.

The method usually accomplishes this by maintaining a *table of sub-instance results*.

Dynamic Programming is a **Bottom-Up Technique** in which the smallest sub-instances are explicitly solved first and the results of these used to construct solutions to progressively larger sub-instances.

In contrast, Divide-and-Conquer is a **Top-Down Technique** which logically progresses from the initial instance down to the smallest sub-instance via intermediate sub-instances.

**Examples:**
- Fibonacci numbers computed by iterations.
- The Manhattan tourist problem
Transform-and-Conquer

These methods work as two-stage procedures. First, the problem is modified to be more amenable to solution. In the second stage the problem is solved.

Types of problem modifications
- problem simplification e.g. presorting
  Example: consider the problem of finding the two closest numbers in an array of numbers.
  Brute force solution: O(n²)
  Transform and conquer solution: O(nlogn)
  Presort the array – O(nlogn)
  Scan the array comparing the differences O(n)
- change in the representation
  Example: AVL trees guarantee O(nlogn) search time
- problem reduction
  Example: least common multiple
  \[ \text{lcm}(m,n) = \frac{m \times n}{\text{gcd}(m,n)} \]

Backtracking and branch-and-bound: generate and test methods

The method is used for state-space search problems. State-space search problems are problems, where the problem representation consists of:
- initial state
- goal state(s)
- a set of intermediate states
- a set of operators that transform one state into another. Each operator has preconditions and postconditions.
- a cost function – evaluates the cost of the operations (optional)
- a utility function – evaluates how close is a given state to the goal state (optional)

The solving process is based on the construction of a state-space tree, whose nodes represent states, the root represents the initial state, and one or more leaves are goal states. Each edge is labeled with some operator.

If a node \( b \) is obtained from a node \( a \) as a result of applying the operator \( O \), then \( b \) is a child of \( a \) and the edge from \( a \) to \( b \) is labeled with \( O \).

The solution is obtained by searching the tree until a goal state is found.

Backtracking uses depth-first search usually without cost function. The main algorithm is as follows:

1. Store the initial state in a stack
2. While the stack is not empty, do:
   a. Read a node from the stack.
   b. While there are available operators do:
i. Apply an operator to generate a child
ii. If the child is a goal state – stop
iii. If it is a new state, push the child into the stack

The utility function is used to tell how close is a given state to the goal state and whether a given state may be considered a goal state.

If no children can be generated from a given node, then we backtrack – read the next node from the stack.

**Example:** The following problems can be solved using state-space search techniques:

1. A farmer has to move a goat, a cabbage and a wolf from one side of a river to the other side using a small boat. The boat can carry only the farmer and one more object (either the goat, or the cabbage, or the wolf). If the farmer leaves the goat with the wolf alone, the wolf would kill the goat. If the goat is alone with the cabbage, it will eat the cabbage. How can the farmer move all his property safely to the other side of the river?

2. You are given two jugs, a 4-gallon one and a 3-gallon one. Neither has any measuring markers on it. There is a tap that can be used to fill the jugs with water. How can you get exactly 2 gallons of water into the 4-gallon jug?

We have to decide:
   a. representation of the problem state, initial and final states
   b. representation of the actions available in the problem, in terms of how they change the problem state.

For the example:

a. problem state: pair of numbers (X,Y): X - water in jar 1 called A, Y - water in jar 2, called B.
   Initial state: (0,0), Final state: (2, _) here "_" means "any quantity"

b. Available actions (operators):

<table>
<thead>
<tr>
<th>Description</th>
<th>Pre-conditions on (X,Y)</th>
<th>Action (Post-conditions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>O1. Fill A</td>
<td>X &lt; 4</td>
<td>(4, Y)</td>
</tr>
<tr>
<td>O2. Fill B</td>
<td>Y &lt; 3</td>
<td>(X, 3)</td>
</tr>
<tr>
<td>O3. Empty A</td>
<td>X &gt; 0</td>
<td>(0, Y)</td>
</tr>
<tr>
<td>O4. Empty B</td>
<td>Y &gt; 0</td>
<td>(X, 0)</td>
</tr>
<tr>
<td>O5. Pour A into B</td>
<td>a. X &gt; 3 - Y</td>
<td>(X + Y - 3, 3)</td>
</tr>
<tr>
<td></td>
<td>b. X ≤ 3 - Y</td>
<td>(0, X + Y)</td>
</tr>
<tr>
<td>O6. Pour B into A</td>
<td>a. Y &gt; 4 - X</td>
<td>(4, X + Y - 4)</td>
</tr>
<tr>
<td></td>
<td>b. Y ≤ 4 - X</td>
<td>(X + Y, 0)</td>
</tr>
</tbody>
</table>

**Branch-and-bound**

Branch and bound is used when we can evaluate each node using the cost and utility functions. At each step we choose the best node to proceed further. Branch-and bound
algorithms are implemented using a priority queue. The state-space tree is built in a **breadth-first** manner.

**Example 1:** the 8-puzzle problem. The cost function is the number of moves. The utility function evaluates how close is a given state of the puzzle to the goal state, e.g. counting how many tiles are not in place.

**Example 2:** the Knapsack problem

**Genetic algorithms**

Genetic algorithms (GAs) are used mainly for optimization problems for which the exact algorithms are of very low efficiency.

GAs search for good solutions to a problem from among a (large) number of possible solutions. The current set of possible solutions is used to generate a new set of possible solutions.

They start with an initial set of possible solutions, and at each step they do the following:

- evaluate the current set of solutions (current generation)
- choose the best of them to serve as “parents” for the new generation, and construct the new generation.

The loop runs until a specified condition becomes true, e.g. a solution is found that satisfies the criteria for a “good” solution, or the number of iterations exceeds a given value, or no improvement has been recorded when evaluating the solutions.

Key issues here are:

- How large to be the **size of a population** so that there is sufficient diversity? Usually the size is determined through trial-and-error experiments.
- How to represent the solutions so that the representations can be manipulated to obtain a new solution? One approach is to represent the solutions as strings of characters (could be binary strings) and to use various types of “crossover” (explained below) to obtain a new set of solutions. The strings are usually called **“chromosomes”**
- how to evaluate a solution? The function used to evaluate a solution is called **“fitness function”** and it depends on the nature of the problem.
- How to manipulate the representations in order to construct a new solution? The method that is characteristic of GAs is to combine two or more representations by taking substrings of each of them to construct a new solution. This operation is called **“crossover”**.
- How to choose the individual solutions that will serve as parents for the new generation? The process of choosing parents is called **“selection”**. Various
methods have been experimented here. It seems that the choice is dependent on the nature of the problem and the chosen representation. One method is to choose parents with a probability proportional to their fitness. This method is called “roulette wheel selection”

- How to avoid convergence to a set of equal solutions? The approach here is to change randomly the representation of a solution. If the representation is a bit string we can flip bits. This operation is called “mutation”.

Using the terminology above, we can outline the algorithms to obtain one generation:

- compute the fitness of each chromosome
- select parents based on the fitness value and a selection strategy
- perform crossover to obtain new chromosomes
- perform mutation on the new chromosomes (with fixed or random probability)

Examples:

- The knapsack problem solved with GAs
- The traveling salesman problem

Conclusion

Usually a given problem can be solved using various approaches however it is not wise to settle for the first that comes to mind. More often than not, some approaches result in much more efficient solutions than others. Consider again the Fibonacci numbers computed recursively using the decrease-and-conquer approach, and computed by iterations using dynamic programming. In the first case the complexity is $O(2^n)$, while in the second case the complexity is $O(n)$. On the other hand, consider sorting based on decrease-and-conquer (insertion sort) and brute force sorting. For almost sorted files insertion sort will give almost linear complexity, while brute force sorting algorithms have quadratic complexity.

The basic question here is: How to choose the approach? First, by understanding the problem, and second, by knowing various problems and how they are solved using different approaches.

Strongly recommended reading to learn more about how to design algorithms:

The Algorithm Design Manual

Steven S. Skiena
Department of Computer Science
State University of New York