Chapter 6

Greedy Algorithms

If greedy algorithms are the first family of algorithms that we examine in detail in this book, the reason is simple: they are usually the most straightforward. As the name suggests, they are shortsighted in their approach, taking decisions on the basis of information immediately at hand without worrying about the effect these decisions may have in the future. Thus they are easy to invent, easy to implement, and—when they work—efficient. However, since the world is rarely that simple, many problems cannot be solved correctly by such a crude approach.

Greedy algorithms are typically used to solve optimization problems. Examples later in this chapter include finding the shortest route from one node to another through a network, or finding the best order to execute a set of jobs on a computer. In such a context a greedy algorithm works by choosing the arc, or the job, that seems most promising at any instant; it never reconsiders this decision, whatever situation may arise later. There is no need to evaluate alternatives, nor to employ elaborate book-keeping procedures allowing previous decisions to be undone. We begin the chapter with an everyday example where this tactic works well.

6.1 Making change (1)

Suppose we live in a country where the following coins are available: dollars (100 cents), quarters (25 cents), dimes (10 cents), nickels (5 cents) and pennies (1 cent). Our problem is to devise an algorithm for paying a given amount to a customer using the smallest possible number of coins. For instance, if we must pay $2.89 (289 cents), the best solution is to give the customer 10 coins: 2 dollars, 3 quarters, 1 dime and 4 pennies. Most of us solve this kind of problem every day without thinking twice, unconsciously using an obvious greedy algorithm: starting with nothing, at every stage we add to the coins already chosen a coin of the largest value available that does not take us past the amount to be paid.

The algorithm may be formalized as follows.

```
function {make-change}(n): set of coins
    {Makes change for n units using the least possible
     number of coins. The constant C specifies the coinage}
    const C = {100, 25, 10, 5, 1}
    S ← ∅ {S is a set that will hold the solution}
    s ← 0 {s is the sum of the items in S}
    while s ≠ n do
        x ← the largest item in C such that s + x ≤ n
        if there is no such item then
            return "no solution found"
        S ← S ∪ {a coin of value x}
        s ← s + x
    return S
```

It is easy to convince oneself (but surprisingly hard to prove formally) that with the given values for the coins, and provided an adequate supply of each denomination is available, this algorithm always produces an optimal solution to our problem. However with a different series of values, or if the supply of some of the coins is limited, the greedy algorithm may not work; see Problems 6.2 and 6.4. In some cases it may choose a set of coins that is not optimal (that is, the set contains more coins than necessary), while in others it may fail to find a solution at all even though one exists (though this cannot happen if we have an unlimited supply of 1-unit coins).

The algorithm is "greedy" because at every step it chooses the largest coin it can, without worrying whether this will prove to be a sound decision in the long run. Furthermore it never changes its mind: once a coin has been included in the solution, it is there for good. As we shall explain in the following section, these are the characteristics of this family of algorithms.

For the particular problem of making change, a completely different algorithm is described in Chapter 8. This alternative algorithm uses dynamic programming. The dynamic programming algorithm always works, whereas the greedy algorithm may fail; however it is less straightforward than the greedy algorithm and (when both algorithms work) less efficient.

6.2 General characteristics of greedy algorithms

Commonly, greedy algorithms and the problems they can solve are characterized by most or all of the following features.

- We have some problem to solve in an optimal way. To construct the solution of our problem, we have a set (or list) of candidates: the coins that are available, the edges of a graph that may be used to build a path, the set of jobs to be scheduled, or whatever.
- As the algorithm proceeds, we accumulate two other sets. One contains candidates that have already been considered and chosen, while the other contains candidates that have been considered and rejected.

- There is a function that checks whether a particular set of candidates provides a *solution* to our problem, ignoring questions of optimality for the time being. For instance, do the coins we have chosen add up to the amount to be paid? Do the selected edges provide a path to the node we wish to reach? Have all the jobs been scheduled?

- A second function checks whether a set of candidates is *feasible*, that is, whether or not it is possible to complete the set by adding further candidates so as to obtain at least one solution to our problem. Here too, we are not for the time being concerned with optimality. We usually expect the problem to have at least one solution that can be obtained using candidates from the set initially available.

- Yet another function, the *selection function*, indicates at any time which of the remaining candidates, that have neither been chosen nor rejected, is the most promising.

- Finally an *objective function* gives the value of a solution we have found: the number of coins we used to make change, the length of the path we constructed, the time needed to process all the jobs in the schedule, or whatever other value we are trying to optimize. Unlike the three functions mentioned previously, the objective function does not appear explicitly in the greedy algorithm.

To solve our problem, we look for a set of candidates that constitutes a solution, and that optimizes (minimizes or maximizes, as the case may be) the value of the objective function. A greedy algorithm proceeds step by step. Initially the set of chosen candidates is empty. Then at each step we consider adding to this set the best remaining untried candidate, our choice being guided by the selection function. If the enlarged set of chosen candidates would no longer be feasible, we reject the candidate we are currently considering. In this case the candidate that has been tried and rejected is never considered again. However if the enlarged set is still feasible, then we add the current candidate to the set of chosen candidates, where it will stay from now on. Each time we enlarge the set of chosen candidates, we check whether it now constitutes a solution to our problem. When a greedy algorithm works correctly, the first solution found in this way is always optimal.

```
function greedy(C: set): set
    {C is the set of candidates}
    S ← ∅ {We construct the solution in the set S}
    while C ≠ ∅ and not solution(S) do
        x ← select(C)
        C ← C \ {x}
        if feasible(S ∪ {x}) then S ← S ∪ {x}
    if solution(S) then return S
                   else return "there are no solutions"
```

It is clear why such algorithms are called "greedy": at every step, the procedure chooses the best morsel it can swallow, without worrying about the future. It never changes its mind: once a candidate is included in the solution, it is there for good; once a candidate is excluded from the solution, it is never reconsidered.

The selection function is usually related to the objective function. For example, if we are trying to maximize our profit, we are likely to choose whichever remaining candidate has the highest individual value. If we are trying to minimize cost, then we may select the cheapest remaining candidate, and so on. However, we shall see that at times there may be several plausible selection functions, so we have to choose the right one if we want our algorithm to work properly.

Returning for a moment to the example of making change, here is one way in which the general features of greedy algorithms can be equated to the particular features of this problem.

- The candidates are a set of coins, representing in our example 100, 25, 10, 5 and 1 units, with sufficient coins of each value that we never run out. (However the set of candidates must be finite.)
- The solution function checks whether the value of the coins chosen so far is *exactly* the amount to be paid.
- A set of coins is feasible if its total value *does not exceed* the amount to be paid.
- The selection function chooses the highest-valued coin remaining in the set of candidates.
- The objective function counts the number of coins used in the solution.

It is obviously more efficient to reject all the remaining 100-unit coins (say) at once when the remaining amount to be represented falls below this value. Using integer division to calculate how many of a particular value of coin to choose is also more efficient than proceeding by successive subtraction. If either of these tactics is adopted, then we can relax the condition that the available set of coins must be finite.

6.3 Graphs: Minimum spanning trees

Let $G = \langle N, A \rangle$ be a connected, undirected graph where N is the set of nodes and A is the set of edges. Each edge has a given nonnegative *length*. The problem is to find a subset T of the edges of G such that all the nodes remain connected when only the edges in T are used, and the sum of the lengths of the edges in T is as small as possible. Since G is connected, at least one solution must exist. If G has edges of length 0, then there may exist several solutions whose total length is the same but that involve different numbers of edges. In this case, given two solutions with equal total length, we prefer the one with least edges. Even with this proviso,