COMP 200: Elements of Computer Science
Fall 2004
Lecture 18: October 6, 2004

More on Greedy Algorithms

On the Board

Homework 4: Available today, due Friday October 15, 2004

One Last Word on Mergesort

The key notion that I was trying to get across with the mergesort example was that a single abstraction may not work well for all the parts of a problem. The divide step of mergesort is trivial in a vector and pairs representation (using a vector to hold the data and <lower,upper> pairs to represent the subranges of the vector). That same representation leads to a complicated implementation for the combine step—it must copy the list elements into a new vector in order to merge them & then copy them back into the space that they occupy in the vector of Keys.[1]

Using Scheme's list abstraction (cons, first, and rest) makes the combine step simple and elegant (see the code posted with the notes for lecture 17). However, the divide step becomes conceptually complex because the list abstraction in Scheme only lets the program access the front of the list. The key to implementing the divide step is to recognize that the algorithm can split the data any way that it wants, rather than keeping elements that start out in contiguous positions next to each other. Thus, the two posted solutions split a list by tossing one element to the left and the other to the right, creating (roughly) equal-size lists but discarding the notion of contiguity.

Greedy Algorithms

The next category of algorithm that Chapter 4 covers is the so-called "greedy" algorithms. Last class, we looked at my favorite instance of a greedy algorithm —driving from Rice to the Galleria (a weighted and simplified version of the Manhattan Grid problem). On that problem, the

[1] The alternative is to use two equal-size vectors, one as the source and the other as the destination for these copies. At each generation of the recursion — that is lists of 1, lists of 2, lists of 4, and so on — the algorithm can swap the designation of source and destination. This swapping scheme eliminates half the data movement, but complicates the conceptual picture.

greedy algorithm will produce an approximate solution — not guaranteed to be the best solution — while considering many fewer paths than would be required to examine every possible path. (I think of the greedy algorithm as a realistic solution because it is an online solution — it makes decisions as it encounters information. An offline solution — one that has access to all the information at the outset — can do better. Driving, unfortunately, is an online problem. We will see an offline version of the problem later.)

An Example

To see the distinction between a problem where the greedy algorithm produces the best answer and one where it may not, consider a simple problem that arises in everyday academic life — classroom scheduling. While the full-blown problem is complex — when is the best time to offer COMP 200 — the subproblem of assigning rooms to classes at a given hour is not.

Class Sizes	Room Sizes
19	150
70	100
25	75
25	75
95	30
6	25
132	25
74	15

Get them to propose an algorithm

A simple greedy algorithm will do well on this problem.

> In decreasing size order, assign each class to the largest available room. If that room is too small, mark the class *unassigned*.

This algorithm works and is optimal. If an assignment exists that satisfies the list of class sizes, it will find one such assignment. It is fast. We must sort the class size list; after that, the algorithm requires constant time for each class. Its overall complexity is *cost of sorting N elements* + *N*.

Problems with Simple Greedy

This simple greedy algorithm is wasteful. In an hour when we have more classrooms than classes, it will ensure that the largest classrooms are booked. Aesthetically, that is a shame. Teaching 10 students in a room that holds 75 is less pleasant than teaching 10 students in a more intimate setting. (As a teacher, you don't want the front of the class to be empty.)

A variation on this algorithm can create a more pleasing assignment.

```
for each class C,
  for each unassigned room R in ascending order by size
      if C fits in R then
         assign C to R
         stop iterating over rooms      (break out of inner loop)
```

This version of the algorithm, sometimes called a best-fit assignment, places each class in the smallest remaining room that will hold it. We might get slightly different results if we consider the classes *C* in different orders (largest to smallest, smallest to largest, or unsorted). However, because it uses the smallest room possible for each class, it will find an assignment if one exists. The cost of this algorithm is, in the worst case, $|C| \times |R|$, or the product of the number of classes and the number of rooms. (At each case, the algorithm might need to traverse the entire list of unassigned rooms. The first time, that list has $|R|$ elements. The next time, it has $|R| - 1$ elements, and so on.

Adding Constraints

What happens to the problem if we add other considerations? For example, some classes need computer projection equipment. What happens if some classes need special resources and not all rooms contain those resources?

Class Sizes	Room Sizes
19	150p
70	100
25	75p
25	75
95	30
6p	25p
132p	25
74p	15

Get them to propose an algorithm

Can we modify our algorithm to handle the constrained problem well?

> *The simple greedy technique will fail. The best-fit greedy technique given earlier will fail.*
>
> *Easy solution is to place constrained classes first, using the best-fit algorithm, then to place the other classes. However, as this example shows, we can construct problems that are not satisfiable.*

What about harder constraints? Faculty members prefer to teach, if possible, in classrooms that are physically close to their offices. This constraint creates a complicated web of calculations — the obvious goal is to minimize, over all classes, the distance from the instructor's office to their assigned classroom. That computation looks much to complex for greedy to guarantee an optimal solution. The interesting question is: how good (or bad) is the greedy solution?

Another Example - Minimum Cost Spanning Trees

A *spanning tree* of a graph is a connected set of edges that touch every vertex and contains no cycles (exactly one path from *a* to *b*). Given an arbitrary graph with weighted edges, construct a minimum cost spanning tree. A greedy algorithm does as well as any algorithm on this problem.

```
TreeNodes ← arbitrary node n
TreeEdges ← Ø

while (|TreeNodes| < number of nodes in the graph)
    let e = (x,y) be the lowest cost edge such that
        x is in TreeNodes and y is not

    add y to TreeNodes
    add (x,y) to TreeEdges
```

The cost of this algorithm depends on the structure of the graph. The algorithm always runs through the while loop *N* times, where *N* is the number of nodes in the graph. Each time it executes the "let e be the lowest cost edge" step, it examines a subset of the edges — the unused edges that leave a node in TreeNodes. The details of how to implement this step are critical to the cost of the algorithm.

The obvious algorithm looks at all the edges leaving TreeNodes. This strategy makes the step cost |Edges| time.

A better strategy maintains a list of unused edges leaving TreeNodes. That idea reduces the cost somewhat, but the worst case is the same. (Each time we add an edge (x,y) to the tree, we add to the list all of the other edges that leave leaving its destination node, y.

If the list of unused edges is kept in sorted order by cost, we can find the next edge in constant time — it is the first edge in the list. Of course, this scheme requires that we insert the edges in sorted order. Chapter 4 discusses one data structure that can maintain such a list — a *heap* or *priority queue*.