

CS1Bh Lecture Note 3

Greedy algorithms

Consider the following problem, known in the computer science literature as *bin packing*. You are given a set of bins, and a collection of n items of weights v_1, \dots, v_n . The maximum weight a bin can hold without breaking is k . The problem consists of packing the items using as few bins as possible.

We can think of the following strategy: we pack each of the items in turn, and put each item in the first bin that can hold it. For instance, assume that the bins can hold weight 6, and that the items have weights 3, 2, 4, 3. Then, the first two items go to the first bin, and for each of the other two items we open a new bin. We can denote this solution by (3,2)(4)(3). We use a total of three bins.

This is an example of a *greedy strategy*. The decision where to put the next object is taken independently of the objects that still have to be packed. At each step, we make a reasonable choice (do not open a new bin if we can still use an old one). An algorithm for an optimization problem is called *greedy* if it follows a greedy strategy.

Greedy algorithms do not always yield optimal solutions; for instance, the four items above can be packed in two bins in the solution (3,3)(4,2). However, for some problems they do. And for many others they provide a good compromise between the quality of the solution and the time required to find it. Finding an optimal solution often requires so much computing time that fast greedy strategies providing reasonable solutions are the best option.

In this lecture we examine two problems in which greedy strategies provide optimal solutions, and then revisit the bin packing problem. The discussion of the first two problems is heavily based on Chapter 17 of the book “Introduction to Algorithms”, by T.H. Corman, Ch.E. Leiserson, and R.L. Rivest (McGraw-Hill, 1990).

3.1 Scheduling a meeting room

Suppose we have a collection m_1, \dots, m_n of meetings that would like to use the same meeting room. Each meeting m_i has a *start time* s_i and a *finish time* f_i (for simplicity we assume that s_i and f_i are integers) such that $s_i < f_i$. Two meetings m_i and m_j are *compatible* if they do not overlap, that is, if $s_i \geq f_j$ or $s_j \geq f_i$. The problem consists of finding a subset of mutually compatible meetings having maximal size, or, in other words, of maximising the number of meetings that take place.

A straightforward greedy strategy is to examine the meetings m_1, \dots, m_n in this order,

and add m_i to the set of selected meetings if it is compatible with the set of meetings selected so far. However, this strategy can give very bad results. For instance, if the first meeting is incompatible with all the other meetings, then only this meeting is selected.

A better greedy strategy is to give preference to meetings that finish early, since, intuitively, they are less likely to be incompatible with others. So, at each step, instead of just blindly taking the next compatible meeting, we choose the compatible meeting with the earliest finish time. This is the greedy strategy that at each step maximizes the amount of unscheduled time remaining.

The Java method `select()` of the class `Meeting` implements this strategy. It first sorts the activities according to their finish time, and then repeatedly selects the first meeting (according to this new order) compatible with those selected so far.

```

1  import java.util.*;
2  class Meeting implements Comparator {
3      int      start;
4      int      finish;
5      boolean selected;
6
7      public Meeting(int s, int f) {
8          start = s;
9          finish = f;
10     }
11     public int compare(Object o1, Object o2) {
12         Meeting m1 = (Meeting) o1;
13         Meeting m2 = (Meeting) o2;
14         return (m1.finish - m2.finish);
15     }
16     public static void select(Meeting[] m) {
17         Arrays.sort(m,m[0]);
18         m[0].selected = true;
19         for (int i = 1; i < m.length; i++)
20             m[i].selected = false;
21         // j is index of the last selected meeting
22         int j = 0;
23         for (int i = 2; i < m.length; i++) {
24             // We check if m[i] is compatible with
25             // the meetings selected so far
26             if (m[i].start >= m[j].finish) {
27                 m[i].selected = true;
28                 j = i;
29             }
30         }
31     }
32 }

```

The sorting is done at line 17 by the `sort()` method of the `Arrays`. The second

argument `m[0]` is passed as a *comparator*: it tells `sort` that meetings should be compared according to the `compare` method of the class `m[0]` belongs to, i.e., according to the `compare()` method of the `Meeting` class.

Surprisingly, this improved greedy strategy always returns an optimal solution! Let us sketch an argument. Assume that m_1, \dots, m_n are sorted according to their finish time, i.e, that $f_1 \leq f_2 \leq \dots \leq f_n$. Let us call any maximum-size subset of mutually compatible meetings a *solution*. We make the following two claims:

- (a) There is an optimal solution containing m_1 .

To see this, we show that any optimal solution can be transformed into another optimal solution containing m_1 . Let A be an arbitrary optimal solution. Consider the set A' obtained by removing from A the meeting with the earliest finishing time, assume it is m_i , and replacing it by m_1 . Since m_i was compatible with all the other meetings of A , and we have $f_1 \leq f_i$, m_1 is also compatible with all the meetings in A . So A' is also a solution.

- (b) If A is an optimal solution for the subset of meetings of $\{m_2, \dots, m_n\}$ that are compatible with m_1 , then $A \cup \{m_1\}$ is an optimal solution of $\{m_1, \dots, m_n\}$.

To see this, imagine that some solution A' is better than $A \cup \{m_1\}$, that is, contains more meetings. Then $A' \setminus \{m_1\}$ is a solution to for the subset of meetings of $\{m_2, \dots, m_n\}$ that are compatible with m_1 . But this contradicts the optimality of A .

Now, (a) and (b) together tell us that we can obtain an optimal solution by choosing m_1 , the meeting with the earliest finish time, and then recursively finding an optimal solution to a smaller instance of the problem. So by repeatedly choosing the meeting with the earliest finish time compatible with those we have chosen so far we always hit an optimal solution.

3.2 Huffman codes

Suppose that we want to store the sequence `abaaabaaaaaac` for later reuse, using as few bits as possible. If we use the code $a = 00$, $b = 01$, $c = 10$, we get the 26 bit representation `00010000000100000000000010`. However, if we observe that the character a appears much more frequently than the other letters, we can do better by giving a a shorter codeword than b and c . We may take $a = 0$, $b = 10$, $c = 11$, and get the 16 bit representation `0100001000000011`. For longer words, this technique may save many kilobytes of memory. Notice that we also have to save the code, but this requires only a few bits, since the number of characters is typically much smaller than the length of the sequence.

The problem we address is: Assuming that the number of occurrences of each character in a string is known, design a code such that the encoding of the string is as short as possible. We consider only codes in which no codeword is a prefix of some other codeword, usually called *prefix codes*. (Otherwise, decoding gets complicated. For instance, if $a = 0$ and $b = 01$, then `0101` is the code of bb and no other word, but in order to check this we have to examine the possibility that the first `0` stands for a , and then discard it by checking that `101` is not the encoding of any word.)

Huffman invented a greedy algorithm that constructs an optimal prefix code, called a Huffman code, assuming that the number of occurrences of each character is known.

Let us see it at work on the example

$$a : 45, b : 13, c : 12, d : 16, e : 9, f : 5$$

meaning that a appears 45 times in the string, b 13 times, etc. First, the two symbols with the lowest number of occurrences (f and e in our case) are selected. They are 'merged' into a new 'combined character' fe whose number of occurrences is the sum of the number of occurrences of the two merged symbols (14 in our case)¹. The key observation is that if we can find a code for

$$a : 45, b : 13, c : 12, d : 16, fe : 14$$

then we also have a code for $a : 45, b : 13, c : 12, d : 16, e : 9, f : 5$: The codeword for e will be whatever the codeword for the combined character fe turns out to be at the end of the algorithm, followed by 0, while the codeword for f will be the codeword for fe followed by 1. So we are left with a smaller instance of the same problem. We can iteratively apply the same procedure to $a : 45, b : 13, c : 12, d : 16, fe : 14$, until we are left with one single combined character. We obtain

$$\begin{array}{ll} a : 45, bc : 25, d : 16, fe : 14 & (d \text{ and } fe \text{ selected for merge}) \\ a : 45, bc : 25, efd : 30 & (bc \text{ and } fed \text{ selected for merge}) \\ a : 45, bcfed : 55 & (a \text{ and } bcfed \text{ selected for merge}) \\ abcfe : 100 & \end{array}$$

The combined character $abcfe$ is assigned the empty string as code. It only remains to 'undo' the merges to get the codewords of each character:

$$\begin{array}{ll} abcfe = "" & (\text{'undo' merge of } a \text{ and } bcfed) \\ a = 0 \quad bcfed = 1 & (\text{'undo' merge of } bc \text{ and } fed) \\ a = 0 \quad bc = 10 \quad fed = 11 & (\text{'undo' merge of } fe \text{ and } d) \\ a = 0 \quad bc = 10 \quad fe = 110 \quad d = 111 & (\text{'undo' merge of } b \text{ and } b) \\ a = 0 \quad b = 100 \quad c = 101 \quad fe = 110 \quad d = 111 & (\text{'undo' merge of } f \text{ and } e) \\ a = 0 \quad b = 100 \quad c = 101 \quad f = 1100 \quad e = 1101 \quad d = 111 & \end{array}$$

Huffman's algorithm is a greedy algorithm. If we view the cost of a merge as the sum of the frequencies of the two nodes being merged, then Huffman's algorithm always chooses the merge of smaller cost.

Huffman codes are optimal, in the sense that they provide optimal compression factors. Proving this is not very difficult, but it is out of the scope of this lecture.

3.2.1 Java implementation

For the implementation in Java we define a class `Node`. Object of this class have a field `character` and a field `occurrences`, which stores the number of occurrences of the character in the string that we wish to encode. There are two more fields, `parent` and `bit`. When the character in `character` is combined with another one, a new node for

¹We consider fe as *one single character*. The name fe is chosen just to remember that this character comes from merging the characters f and e .

the combined character is created. We then assign this node to `parent`, and either '0' or '1' to `bit`. This allows us to recursively compute the codeword of a node `n` as the codeword of `n.parent` followed by `n.bit`.

Notice that Huffman's algorithm requires to compare nodes by comparing their number of occurrences. This is the purpose of the `compare` method.

```

1  import java.util.*;
2  class Node implements Comparator {
3      char    character;
4      int    occurrences;
5      char    bit;
6      Node    parent;
7
8      public Node(){};
9      public Node(char c, int n) {
10         character = c;
11         occurrences = n;
12     }
13     public int compare(Object o1, Object o2) {
14         Node n1 = (Node) o1;
15         Node n2 = (Node) o2;
16         return (n1.occurrences - n2.occurrences);
17     }
18 }
19
20 class Huffman {
21     public static Node merge(Node n1, Node n2) {
22         Node n = new Node();
23         n.occurrences = n1.occurrences + n2.occurrences;
24         n1.bit = '0'; n1.parent = n;
25         n2.bit = '1'; n2.parent = n;
26         return n;
27     }
28     public static void makeTree(LinkedList l) {
29         TreeSet t = new TreeSet(new Node());
30         t.addAll(l);
31         while (t.size() >= 2) {
32             Node n1 = (Node) t.first();
33             t.remove(n1);
34             Node n2 = (Node) t.first();
35             t.remove(n2);
36             Node n = merge(n1,n2);
37             t.add(n);
38         }
39     }
40 }

```

The `Huffman` class has a method `merge` that merges two nodes into a third node for the combined character. This new node becomes the parent of the two merged nodes. The method `makeTree()` takes a linked list of `Nodes` as inputs, and iteratively merges characters, until one single `Node` remains. We use a `TreeSet` for the computation, since on top of the methods `add()` and `remove()` it also provides a method `first()` that returns the smallest element of the set according to the comparison relation given by the `compare` method.

3.3 Bin packing

Bin packing is one of those problems for which computing an optimal solution requires too much computing time. (To be more precise, despite very considerable effort nobody has been yet able to find an efficient procedure to compute the optimal solution, and most computer scientist believe that there is no such procedure.)

Instead of finding an optimal solution, reasonable solutions are found in reasonable time using a number of greedy strategies. The best known are

- **First Fit.** A new item is placed in the leftmost bin that still has room.
- **Last Fit.** A new item is placed in the rightmost bin that still has room.
- **Next Fit.** A new item is placed in the rightmost bin, opening a new bin if necessary.
- **Best Fit.** A new item is placed in the fullest bin that still has room.
- **Worst Fit.** A new item is placed in the emptiest existing bin.
- **Almost Worst Fit.** A new item is placed in the second-emptiest bin.

As we can see, there can be many different greedy strategies for the same problem. Which one is the best usually depends on the application. In some rare cases, analysis may also show that a strategy is better than another one in all cases.

3.4 Summary

Many algorithms obtain a solution to a problem by making a sequence of choices. Greedy algorithms make at each point the choice that seems best at the moment. Such heuristic strategies do not always produce an optimal solution, but sometimes they do. Sometimes computing an optimal solution requires too many resources, and greedy heuristics can provide a reasonable solution in reasonable time. There can be many different greedy heuristics for the same problem.

Javier Esparza, January 20th, 2003.