CS450: Structure of Higher Level Languages

Assignment 4, pt 2

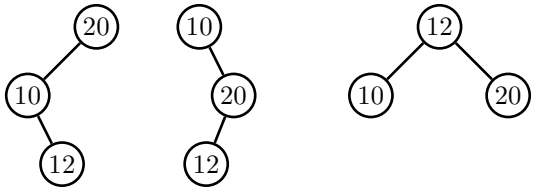
Due: Sunday, Nov. 1, 2020

Dynamic Programming and the Remarkable Effectiveness of Memoization

In this part you will explore different ways to implement a minimum cost binary search tree.
See https://www.geeksforgeeks.org/optimal-binary-search-tree-dp-24/ for more information. It also contains a C/C++ implementation which you may refer to for ideas, but a scheme implementation should be rather different.

Statement of the problem: Notice, this is all CS310 material! I am sure you remember it all very well, but I'll still include it here for completion. Here is a problem: we are given a set of keys and their weights. You may think of the keys are terms in a dictionary, and the weights are the number of searches for each node. In other words – the list `(1 10)` indicates the key "1", which has a weight of 10 (you can think of it as the probability of it being looked up but it doesn't have to be! The weights can be any positive number). We want to store the keys in a binary search tree (BST), ordered by the keys (NOT the weights!). There are many possible BSTs for a given set of keys, so we want the one which has the minimal expected cost of locating each node.

For example, if our input is: keys= {10, 12, 20}, weights = {34, 8, 50}, three possible trees are:



The overall search cost for a particular node is the number of operations we have to preform to find it, which is equal to the level of this node in the tree. the level of a node – $l(n) = l(parent) + 1$, where $l(root) = 1$, multiplied by its weight.. We define the overall search cost of a tree as $\sum_{i=1}^{n} l(i)w(i)$ where $w(i)$ is the weight of node i. Therefore, the search costs for the three trees in the figure above are (left to right):

50*1+34*2+8*3 = 142, 34*1+50*2+8*3 = 158, 8*1+50*2+34*2 = 176.

This makes the left most tree better than the other two, since the overall search cost is lower. As a matter of fact, this is the minimum cost tree.

Recursive Computation of the Optimal Solution:

Let us define the optimal cost of the binary search tree made of a set of keys $i \ldots j$, sorted by key value (not weight) as $OptCost(i, j)$. The cost can be recursively calculated using following formula:

$$OptCost(i, j) = \sum_{k=1}^{j} weight(k) + \min_{r=i..j} [OptCost(i, r-1) + OptCost(r+1, j)]$$

Explanation: Any one of the keys can be the root of a binary search tree. The choice of root r divides the set of keys into three parts: The root, the left subtree and the right subtree (each subtree can be empty. The optimal cost of an empty tree is trivially 0). The idea of the above formula is simple: we try all nodes as root (r runs from i to j in the second term). When r is the root, we recursively calculate the optimal cost of the left subtree made of keys i to r-1 and the right subtree made of r+1 to j. We add to it the sum of the weights from i to j (see first term in the above formula), to account for the fact that the left and right subtrees are hanging from the root and therefore their levels are one more than they would be if they were standalone trees.

For the overall tree we need to calculate $OptCost(1, n)$. This formula shows that the problem has *optimal substructure*. In other words – if $OptCost(i, j)$ is the minimum cost for keys $i \ldots j$, then the left and right subtrees represent the trees with minimal costs for their respective subsets of keys. This can be shown by a simple *cut-and-paste* argument: If any of the subtrees did not represent the minimal cost BST for their set of keys, we could replace them with a better BST, getting a smaller cost overall, which contradicts our assumption that $OptCost(i, j)$ is the minimum cost for keys $i \ldots j$.

How the data is input: Before we go on, let us explain how the data will be input to the program you are going to write. There will be a file containing the keys and weights.

A typical (but very small) files called `keys-small.dat` file might look like this:

```
(10 34)
(12 8)
(20 50)
```

The file is attached as part of the handout. You can also generate other test cases. Each line in the file looks like a Scheme list. Each line represents a key and its weight. You may assume the keys are sorted (by key, not by weight).

Reading in the graph: As in the previous assignment, you can use the following code to read in the graph:

```
;; read-file produces a list whose elements are the expressions in the file.

(define (read-file)
  (let ((expr (read)))
    (if (eof-object? expr)
        '()
        (cons expr (read-file)))))

;; Here we go:  read in the file that defines the graph

(define data (with-input-from-file "keys-small.dat" read-file))
```

In the example above, the variable `data` evaluates to `((10 34) (12 8) (20 50))`.

Using recursion to solve our problem: The Pseudo-code for the recursive algorithm might look something like this. (Bear in mind that this pseudo-code is not really "pseudo-Scheme" – the syntax is wrong – but it could be easily translated into it.):

```
// returns a number, takes a list of lists which represents a set of keys and weights
procedure min-cost-naive(list)
  initialize return value to infinity
  if the list is empty, return 0
  If the list has only one member, return its weight // a single leaf
  otherwise, for each key k on the list
    split the list around k to ((left) k (right))
```

```
  // left and right are the lists of keys on the left and right subtrees, resp.
   compute (sum of weights + min-cost-naive(left) + min-cost-naive(right))
return the minimum of those computed values
```

You can use a variable whose initial value is "infinity" for the starting value. You can use the number 1000000 (i.e., 1 million) if you want – I guarantee that I won't try your code out on any trees with weights that could possibly add up to a million. And of course any trees that you make up should satisfy the same constraint. That shouldn't be hard.

Your first task – the function min-cost-naive: Write a program `(min-cost-naive data)` that takes a list of list (like the variable `data`) and calculates the cost of the minimum binary search tree based on the formula above. It needs to output that number, so in the example above the function should print out 142. Write your code in the `ASanswers.scm` you started in part 1. Notice that the formula above returns the cost of the tree, not the tree itself (and that's ok because that's what I'm asking for at this point).

As you implement your solution you will (hopefully) realize **you don't really need to build the actual tree at this point**! You will need to figure out how to split your data into root, left and right, but list operations can do the work just fine. You will need the tree for later, though.

Towards a more efficient solution

The recursive solution to the problem is computationally very expensive! The reason is that the larger the tree is, the more subtrees you have to calculate over and over again. Every time we try to calculate $OptCost(i, j)$ we end up calculating all the subsets of contiguous keys even if we calculated them before. This is unnecessary, because we will always get the same answer, and there are only $O(n^2)$ unique subsets of contiguous keys. Therefore, there are many *overlapping subproblems*. This allows us to devise a much more efficient Dynamic Programming (DP) solution by calculating and the values of subproblems, and saving for later.

Introducing memoization: As we have already mentioned, the naïve algorithm is just too costly to run on trees, but based on the discussion above, we can get around this difficulty by making sure that we compute the cost of each subtree just once. We do this by a process called memoization (again, see CS310).

The word "memoization" (without the "r") is used only in Computer Science. Every computer scientist knows what it means, and almost no one else does. The idea is simple: we keep a table of costs. For each set of keys, when we compute the cost of the BST for this set, we enter that subset and their cost to the table. Then every time after that, when we want to compute the cost of the BST for that subtree, we first look it up in that table. If it's there, then we know the cost. Otherwise, we compute it and put it and its cost in the table.

In effect we are making a "memorandum" (or "memo") of the cost of each BST as we compute it. That's where the term "memoize" comes from. This use of memoization, which is made possible by the optimal substructure property we proved above, in conjunction with a naïve recursive algorithm, is basically dynamic programming.

Building the main lookup table: The next thing you will need to do is to build a lookup table. This table should enable you to implement a lookup function (in fact, let's call it lookup) that takes a list of contiguous keys and weights (representing the segment $i \ldots j$ and points to the minimum cost of the BST for this set of keys. The lookup function should evaluate to #f if the list is not in the table. **Note:** For this stage you only need the cost, but for the next step you will also need to root that minimizes the cost of the tree, so you may want to have the lookup evaluate to a pair `(root . cost)`.

We talked in class about how to build lookup tables (and the same thing is in the textbook, in section 3.3.3). In this case, we are talking about a one-dimensional table. You should use the method we talked about in class.

Your second task: You should implement this memoized version. Call it `min-cost` (as opposed to `min-cost-naive`). In doing this you will want to be very careful that you really are not redoing computations that can be memoized.

You will probably not need more than one table, but you may want to build more than one, up to you. Notice that `min-cost` and `min-cost-naive` should not be extremely different from one another, other than the lookup. In other words when you need to retrieve the cost of a subtree you should first look at the table and calculate only of it's not there. Also, once you calculated a subtree, you should insert it into the table(s). Obviously, the two functions should return exactly the same cost, but `min-cost` should run faster. If the costs are different, you are doing something wrong.

Your third task: Finally, we would like to print out the tree itself (As mentioned, the procedures above only compute the cost!). To do this, you will also need to memoize not only the minimum cost but also the root information for each level in the tree. In principle, this is not at all hard to do, but you will need to be careful.

I'm sure you will need some helper functions in doing all this. And you may also need an additional table. I provided a message passing implementation of a binary search tree which includes lookup, insert and print. Please use this one. I also initialized a tree called table. Use this one for the task. Your output should be the function `(table 'print)`

Test cases: In developing your program you will certainly want to work with small trees that you make yourself, so that you can debug your code easily. For debugging I suggest you take a small test case (3-4 nodes) and calculate the minimum cost by hand, making sure it matches your code. Later you might like to have a somewhat larger graph to test your code on. I have put two examples called `keys-medium.dat` and `keys-large.dat` as a handout. You can certainly feel free to use it. The large example should give you a minimum cost of 4.56 (my racket printed out 4.5600000000000005). The medium example should give you a minimum cost of 2.84 You should also find that your function cost runs much faster than `naive-cost` on this test case. (And if it doesn't, you have definitely done something wrong.)

Final warnings/notes:

- We have just learned about mutable data; in particular about the special form `set!` and the functions `set-car!` and `set-cdr!`. You should actually not need to use `set!` or `set-car!` at all in this assignment. You will need to use `set-cdr!`, but only in one context: when you are defining an insertion procedure for a lookup table or the tree. (And you will probably have several tables, so you will have an insertion procedure for each one.) If you find yourself using `set!` or `set-car!` anywhere, or if you find yourself using `set-cdr!` in any other circumstance than the one I just specified, you are undoubtedly doing things wrong. Remember, we're writing in Scheme here, not in some C-derived language. This is important. We're trying to learn the functional programming.
- When you create your own test cases you may assume no key appears twice (weights can be repeated as many times as you want) and that the input data is sorted by key, even though it's not critical if you do it right.
- Make sure that you are indeed using DP for the second part! It will reflect in your timing.

Of course, there are design decisions that you will make as you implement this, and different people will do things somewhat differently. That's perfectly OK, and in fact I would expect that. That's not what I'm talking about here. If you're uncertain what I mean in this all, please just send me email.

Delivery

Use the handout `ASanswers.scm` – the same one from part 1. Don't change the name and **if you modify the functions I wrote you do it at your own risk**. You can add whatever you want to it. You should add, at the minimum, the following three procedures:

1. `(min-cost-naive data)` where `data` is a list of lists of keys and weights (see above). The function should evaluate to a number – the cost of the minimum tree, using the naïve recursive algorithm.

2. `(min-cost data)` where `data` is a list of lists of keys and weights (see above). The function should evaluate to a number – the cost of the minimum tree, using the DP algorithm.

3. `(build-min-tree data)` that evaluates to the actual BST that gives the minimum cost. Notice that if you do it right, you don't have to repeat question 2, but use what you already have.

Theoretical questions

Submit the answers to these questions in a separate single file on gradescope, similar to HW3.

1. Write a short essay (1-2 paragraphs) about your design choices - how many tables did you use? the difficulties you ran into and how you found the assignment in general.

2. Time your naïve function and DP function. You can use the timed function from HW3 by copying the following code:

```
(#%require (only racket/base current-milliseconds))
(define (runtime) (current-milliseconds))

(define (timed f . args)
  (let ((init (runtime)))
    (let ((v (apply f args)))
      (display (list 'time: (- (runtime) init)))
      (newline)
      v)))
```

   Try the `keys-medium.dat` and `keys-large.dat` examples, available from the course webpage. What is the run time in both cases for the naïve and DP? The DP implementation should be much faster. If not, you are doing it wrong.

3. Time your tree construction function (third task) for `keys-large.dat` . It should be about the same order of magnitude as your DP implementation - that is, very little time. If it takes much more time, you're doing it wrong.

4. Draw the tree resulting from the set of keys in the medium test case named `keys-medium.dat` . For this purpose you need to translate the printed tree from scheme into a drawable binary search tree. You may have to experiment with a few smaller examples.