2. Optimization to Problems

Thursday, April 16, 2020 7:18 AM

Review

Greedy Algorithm

Pros:

- Easy to implement
- Very Fast
- If a problem fits the greedy property, you can take a shortcut: the globally optimal solution can be arrived at by making locally optimal choice (i.e. greedy choice property)

Cons:

- Doesn't actually solve the problem (may or may not be optimal)
- We don't know how close to optimal it is

Search Trees

Using a Search Tree to Implement the Greedy Algorithm

Search tree enumerates possibilities. Using a root to draw out consideration for each of the two element options (children). You run the recursive function until you reach the leaf (end)

Left First Depth First

Traversing a tree by all the left-hand side (taking all your options)

- The levels reflect the number of choices
- Number of node = 2^n+1

Use

- It's helpful to list all potential outcomes at every point that you need to make a decision-- all outcomes are unique because it captures everything you take and don't take (inside and outside the bag)
- You can then weigh the value and constraint at the end (implementing the greedy algorithm)
- It doesn't actually build a tree, but just stores each legal solution at every iteration and updates the best solution when it finds one

Implementation

Header for Decision Tree Implementation

```
def maxVal(toConsider, avail):
    """Assumes toConsider a list of items,
               avail a weight
       Returns a tuple of the total value of a
           solution to 0/1 knapsack problem and
           the items of that solution"""
```

`toConsider`. Those items that nodes higher up in the tree (corresponding to earlier calls in the recursive call stack) have not yet considered

`avail`. The amount of space still available

Body of maxVal (without comments)

```
if toConsider == [] or avail == 0:
        result = (0, ())
    elif toConsider[0].getUnits() > avail:
        result = maxVal(toConsider[1:], avail)
    else:
        nextItem = toConsider[0]
        withVal, withToTake = maxVal(toConsider[1:],
                                     avail - nextItem.getUnits())
        withVal += nextItem.getValue()
        withoutVal, withoutToTake = maxVal(toConsider[1:], avail)
    if withVal > withoutVal:
            result = (withVal, withToTake + (nextItem,))
        else:
            result = (withoutVal, withoutToTake)
    return result
```

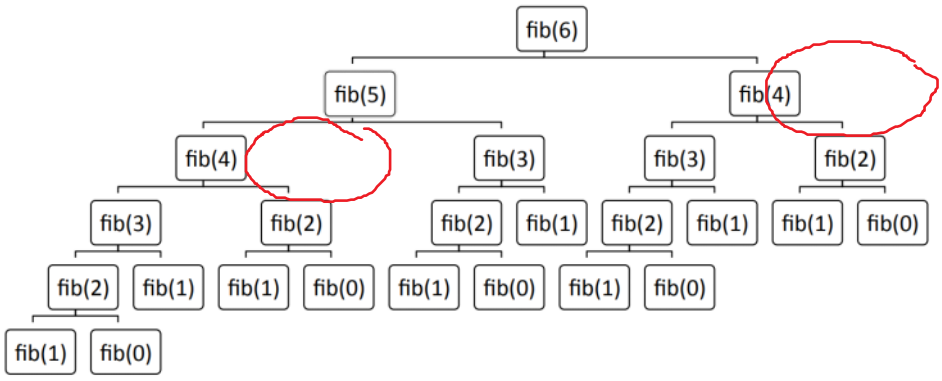
Does not actually build search tree
Local variable `result` records best solution found so far

Limitation

Search trees can get really slow past a certain number of options.

The call tree shows that it can be a wasteful algorithm because it does the same calculations over and over again (i.e. lots of duplicate calculations)

Call Tree for Recursive Fibonnaci(6) = 13



Solution: you store the answer and look it up when you need it (this is the foundation of dynamic programming)

- Aka Memo-ization (creating a memo)
- You save time by trading space (negligible because it saves you a lot of time without a lot of space)

Dynamic Programming

The problems it can help is when problems have optimal substructure and overlapping subproblems

- **Optimal substructure**: a globally optimal solution can be found by combining optimal solutions to local subproblems
  - For x > 1, fib(x) = fib(x - 1) + fib(x – 2)

- **Overlapping subproblems**: finding an optimal solution involves solving the same problem multiple times
  - Compute fib(x) or many times

Laymen's Term:

- You can solve two independent problems and then putting them together
- Creating a memo is less effective if you don't solve the same problems multiple times

Example

- Dynamic cannot help us with Merge sort: they're independent, but they're different sort problems. In other words, when you separate the problems, there's no overlapping subproblems
- Search tree help with dynamic programming?
  - Does have optimal substructure: choosing between the left branch or the right branch
  - Overlapping subproblems: Non of the branch looks identical, so though you can have dynamic programmed algorithm but you won't see a speed up

Link

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0002-introduction-to-computational-thinking-and-data-science-fall-2016/lecture-videos/lecture-2-optimization-problems/