Dynamic Programming - 1

David Bernick
Dynamic Programming

In problem cases that have:

- optimal substructure, and
  - The optimal solution can be decomposed into smaller problems that can be optimally solved, and those problems can also be decomposed and optimally solved, recursively

overlapping subproblems
  - Subproblems that come about in one part of the decomposition exist in other parts of the problem decomposition

we can use *Dynamic Programming* to avoid the recomputation of those overlapping subproblems.
Dynamic Programming – 2

• The idea behind Dynamic Programming is:
  – Remember the result of a computation, and
  – Check if you have the result before recomputing it.

• Also called: memoization
• Also called: caching
• Also called: tabling
• And maybe: “squirreling away”
Dynamic Programming – 3

- Used to reduce complexity from exponential $O(A^X)$ to polynomial ($O(X^k)$)
Change Problem

• Consider the problem of computing the minimum number of coins to be returned in order to provide a specific value to a customer

• How do we do this task?
  Usually a greedy algorithm.
  – Choose the largest coin available that is less than the remaining change needed
  – Repeat until remaining change needed is zero.

• Optimality of the above algorithm is dependent on the denominations of coin available in the specific currency (JP Ch. 2)
Change Problem – 2
Consider calculating with denominations: 1, 3, 7 to yield 37

A beautiful recursive solution:

def recursiveChange(M,coins):

global calls
    calls += 1

    if M==0:
        return 0

    bestNumCoins = sys.maxint
    for coin in coins:
        if M >= coin:
            thisNumCoins = recursiveChange(M-coin,coins)
            if (thisNumCoins + 1) < bestNumCoins:
                bestNumCoins = thisNumCoins + 1

    return bestNumCoins

M = 37
c = [1,3,7]
calls = 0
bestNum = recursiveChange(M,c)
print bestNum, calls
Change Problem – 2 demo
Change Problem – 3

Well ......

that is really slow

and we seem to be doing lots of recalculation
def recursiveChangeCached(M, coins):
    global calls
    calls += 1
    global changeCache
    if M <= 0:
        changeCache[0] = 0
        return 0
    bestNumCoins = sys.maxint
    for coin in coins:
        if M >= coin:
            try:
                thisNumCoins = changeCache[M-coin]
            except KeyError:
                # print "calling (", M-coin, ")"
                thisNumCoins = recursiveChangeCached(M-coin, coins)
                changeCache[M-coin] = thisNumCoins

            if (thisNumCoins + 1) < bestNumCoins:
                bestNumCoins = thisNumCoins + 1
    return bestNumCoins

M = 37
C = [1, 3, 7]
calls = 0
changeCache = {}
bestNum = recursiveChangeCached(M, C)
print bestNum, calls
Change Problem – 3 demo
Change Problem – 4

... can we prefill the cache?

def preFillCache (M, coins):
    global changeCache
    changeCache[0] = 0
    for iM in range (1, M + 1):
        bestNumCoins = sys.maxint
        for coin in coins:
            if iM >= coin:
                if changeCache[iM - coin] + 1 < bestNumCoins:
                    bestNumCoins = changeCache[iM - coin] + 1
                changeCache[iM] = bestNumCoins

M = 37
c = [1, 3, 7]
calls = 0
changeCache = {}
preFillCache (M, c)
bestNum = recursiveChangeCached(M, c)
print bestNum, calls
Change Problem – 4 demo
Dynamic Programming

Uses Caching of results from redundant subproblems to accelerate run time. Redundant computation(s) happen(s) once

Will not help to solve problems that do not have redundant pieces, or those that can not be decomposed to simpler problems, or those with side-effects.
Knapsack problem

You have a fixed size knapsack that allows carrying 20 units of weight. Maximize the total value of the items carried.

There are 17 items of varying weight and value. How many possible sets exist?

Is this problem amenable to Dynamic Programming?
Knapsack

• A collection of objects
  – Maximum # objects subject to weight constraint
• N objects, $2^n$ possible sets
• Optimal substructure?
  – Decision tree
  – Weights = \([5,3,2]\), max=5
  – Values = \([9,7,8]\)

– \((2,5,0)\) index, weight available, value
  • Left (don’t take) \((1,5,0)\)
    – (don’t take) \((0,5,0)\)
      » (don’t take) \((-5,0)\)
      » (take)(-,0,9)
    – (take)(0,2,7)
      » (don’t)(-,2,7)
      » (take) none
  • (take)(1,3,8)
    – (don’t)(0,3,8)
      » (don’t)(-,3,8)
      » Take(None)
    – (take)(0,0,15)
      » (don’t)(-,0,15)
      » (take)(None)