CS350 ABET Objective 10

Part 2 – Dynamic Programming

Design dynamic programming algorithms.

Textbook Section 8, ~15%.

Dynamic programming is a technique for solving problems with overlapping subproblems. Typically, these subproblems arise from a recurrence relating a solution to a given problem with solutions to its smaller subproblems of the same type. Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which we can then obtain a solution to the original problem [Levitin 2012].

The technique is presented by a set of examples.

Example 1

Coin-row: pick maximum amount from a row of coins, but no adjacent coins.

c_1	c_2	c_3	c_4	c_5	c_6
5	1	2	10	6	2

$F(n)$ is the maximum amount that can be picked from the first n coins with $F(0)=0$ and $F(1)=c_1$.

Should I pick coin c_n ?

YES: $F(n)=c_n+F(n-2)$

can't pick c_{n-1}

NO: $F(n)=F(n-1)$

forget c_n , do best with the rest

Choose the maximum.

Trace 1

F(0): 0 by definition

F(1): 5 by definition

F(2): choose F(1)=5 or 1+F(0)=1, take 5

F(3): choose F(2)=5 or 2+F(1)=7, take 7

F(4): choose F(3)=7 or 10+F(2)=15, take 15

F(5): choose F(4)=15 or 6+F(3)=13, take 15

F(6): choose F(5)=15 or 2+F(4)=17, take 17

The maximum amount that can be picked from the first 6 coins, without using adjacent coins, is 17.

Example 2

Change-making: give change with minimum number of coins of given denominations.

d_1	d_2	d_3
1	3	4

Instance: change is 6 coins are 3 and 3.

$F(n)$ is the minimum number of coin that add up to n ; $F(0)=0$.

Should I put a coin of denomination d_j ?

1. look at $F(n-d_j)$ for all <u>acceptable</u> d_j ;
2. choose the minimum;
3. add 1 to obtain $F(n)$.

Trace 2

change goal 1
use denomination 1
subproblem change 0 coins []
accept denomination 1 number of coins 1
solve goal 1 coins [1]

change goal 2
use denomination 1 #

subproblem change 1 coins [1]
accept denomination 1 number of coins 2
solve goal 2 coins [1,1]

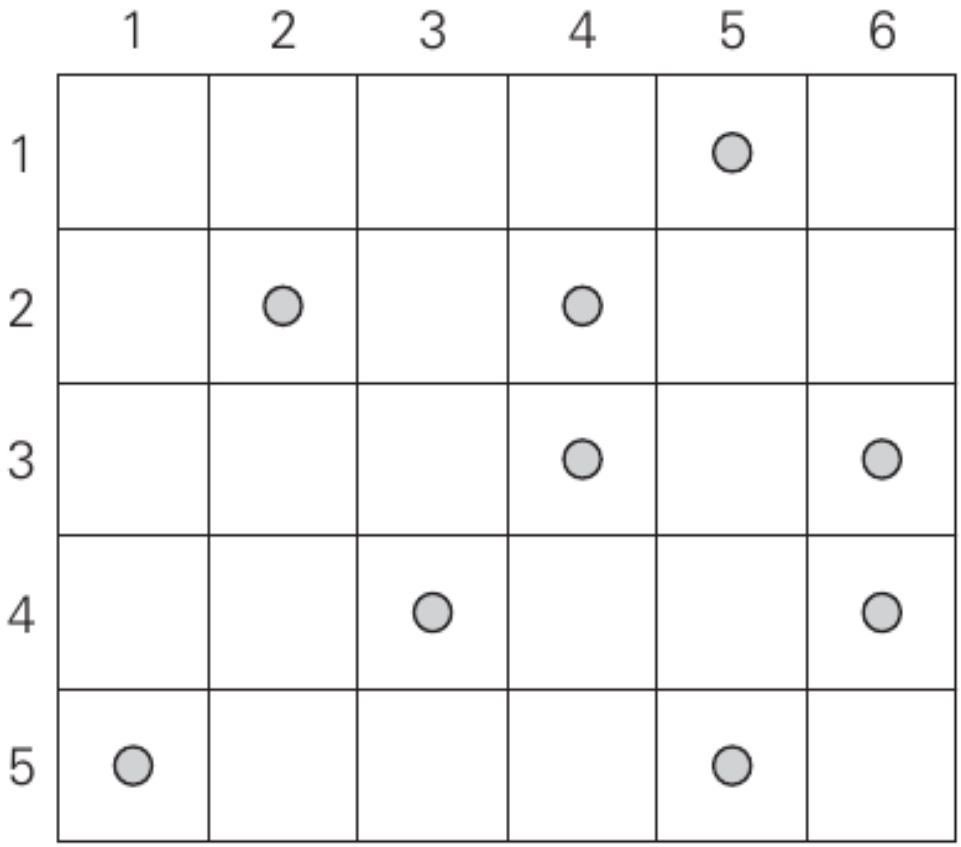
change goal 3
use denomination 1
subproblem change 2 coins [1,1]
accept denomination 1 number of coins 3
use denomination 3 #

subproblem change 0 coins []
accept denomination 3 number of coins 1
solve goal 3 coins [3]

change goal 4
...

Example 3

Coin-collection: collect the most coins, each one in the cell of board, by moving over the cell in a right-down path from the top-left corner to the bottom-right corner of the board.



A board cell, $C(i,j)$, is either 0 or 1.

$F(i,j)$ is the maximum number of coins that can be collected from cell (i,j), with

$$
\begin{aligned}
F(1,1) &= C(1,1) \\
F(1,j) &= F(1,j-1) + C(1,j) \\
F(i,1) &= F(i-1,1) + C(i,1) \\
F(i,j) &= max(F(i-1,j), F(i,j-1)) + C(i,j)
\end{aligned}
$$

The above equations hold with $i>1, j>1$.

The result is in the bottom-right entry of F.

If the computation looks “reversed” to you, consider moving up and left from the lowest-rightmost cell of the board.

Example 4

Knapsack: given n items $(w_{1,}v_1)\ldots(w_n,v_n)$ where w_i is a weight and v_i is a value, find a most valuable combination of items that fit into a knapsack of capacity W .

The smaller instance is the problem for the first i items, $1\leqslant i\leqslant n$, and a knapsack of capacity j , $1\leqslant j\leqslant W$. Let $F(i,j)$ be the value of this instance.

Should item i go into the knapsack?

No: $F(i,j)=F(i-1,j)$

Yes: $F(i,j)=v_i+F(i-1,j-w_i), \quad w_i\leqslant j$

Choose the largest.

Memory Functions

The examples so far produced bigger subproblems from smaller ones —— a bottom-up approach. We can solve a problem top-down by remembering (memoization) computed instances of subproblems (to avoid recomputation).

ALGORITHM *MFKnapsack*(i, j)

```
//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer i indicating the number of the first
//       items being considered and a nonnegative integer j indicating
//       the knapsack capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays Weights[1..n], Values[1..n],
//and table F[0..n, 0..W] whose entries are initialized with −1's except for
//row 0 and column 0 initialized with 0's
if F[i, j] < 0
    if j < Weights[i]
        value ← MFKnapsack(i − 1, j)
    else
        value ← max(MFKnapsack(i − 1, j),
                    Values[i] + MFKnapsack(i − 1, j − Weights[i]))
    F[i, j] ← value
return F[i, j]
```

Warshall 1

Compute the transitive closure of a directed graph (binary relation).

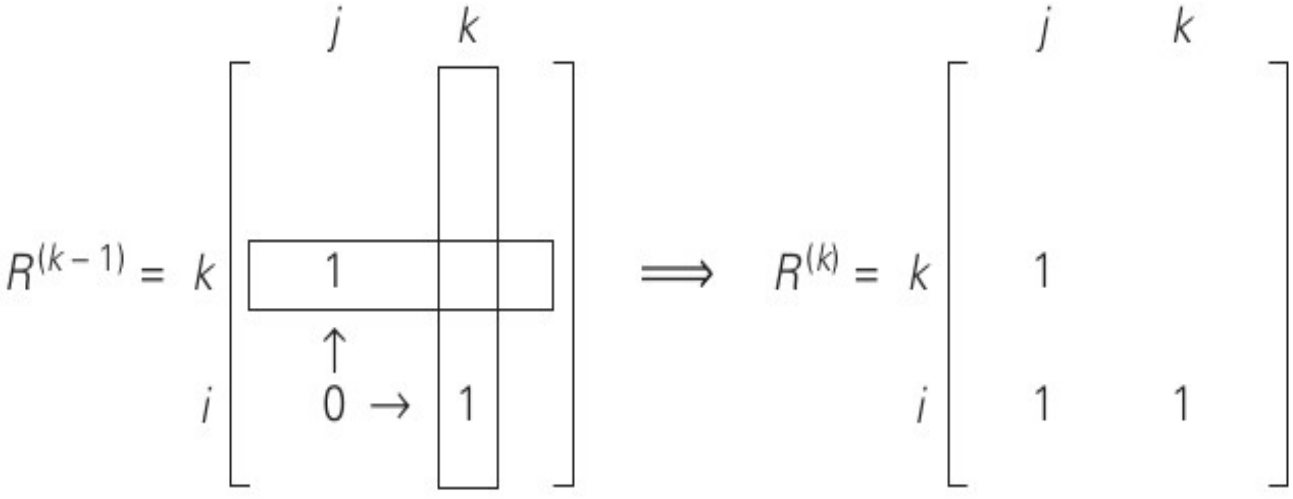
Input: adjacency matrix of graph,entry 0/1.

Output: adjacency matrix of transitive closure.

Vertices (nodes) are numbered $1,2,\ldots$

Compute matrices $R^{(0)},\ldots R^{(k-1)},R^{(k)},\ldots R^{(n)}$.

Element $r_{ij}^{(k)}=1$ iff exists path from i to j with intermediate nodes no higher than k .



Warshall 2

Compute the transitive closure of a directed graph

ALGORITHM *Warshall*$(A[1..n, 1..n])$

```
//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
R^(0) ← A
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            R^(k)[i, j] ← R^(k−1)[i, j] or (R^(k−1)[i, k] and R^(k−1)[k, j])
return R^(n)
```

Running time is $\Theta(n^3)$.

Floyd

Compute the shortest path of every pair of nodes in a directed graph (binary relation). Similar to Warshall. Adjacency matrix contains ∞ for no edge.

ALGORITHM *Floyd*$(W[1..n, 1..n])$

```
//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
D ← W  //is not necessary if W can be overwritten
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
return D
```

Running time is $\Theta(n^3)$.

A variation with a second matrix of equal size computes also the shortest path in addition to its cost.

References

Textbook Section 8

Web:
http://en.wikipedia.org/wiki/Floyd-Warshall_algorithm