GB21802 - Programming Challenges

Week 3 - Dynamic Programming (Part I)

Claus Aranha
caranha@cs.tsukuba.ac.jp

College of Information Science

2015-05-13,16

Last updated May 12, 2016

Last Week Results

Week 0 - Simple Problems

Problems Solved

- Division Of Nlogonia: 28/28
- Cancer Or Scorpio: 24/27
- 3n+1: 25/25
- Request for Proposal: 19/22

Manaba Submission: 29 Students

Week 1 - Data Structures

Problems Solved

- Jolly Jumpers: 27/27
- Army Buddies: 16/22
- Rotated Square: 18/18
- File Fragmentation: 5/6
- Contest Scoreboard: 10/11
- Multitasking: 5/7
- Jolibee Tournament: 10/10

Manaba Submission: 25 students

Week 2 - Search Problems

Problems Solved (as of Thu 17:00)

- Division: 19/22
- Social Constraints: 9/9
- Simple Equations: 1/7
- Bars: 11/13
- Rat Attack: 9/10
- Little Bishops: 2/2
- Water Gate: 3/3
- Through the Desert: 4/4
- Dragon of Loowater: 12/13
- Shoemaker's Problem: 5/6

Don't Give Up!

Special Notes

Dynamic Programming (DP)

Dynamic programming (DP) is a more challenging, but much more versatile, approach for search algorithms.

Basic idea of DP

Define a table with all unique search states of a problem, and fill that table (in a top-down or bottom up fashion) until the desired solution is found.

Key skills needed for DP

- Determine state space for a problem;
- Determine transition between spaces;

Uses of DP

DP is most often used in *optimization* or *counting* problems.

If your problem can be described as:

- Maximize this...
- Minimize that...
- Count the ways to do...

Then there is a very high chance that you should use DP to solve it.

Problem Example: Wedding Shopping – UVA 11450

Best way to understand DP is to do a lot of examples!

Problem outline:

- Consider budget $M < 200$ to buy $C < 20$ classes;
- Each class has $K_i < 20$ items, each with different cost;
- Goal: Choose one item from each class so that the money used is maximum; Do not go over the budget!

Problem Example: Wedding Shopping – UVA 11450

Sample case 1: $C = 3$

Class	1	2	3	4
0	6	4	8	
1	5	10		
2	1	5	3	5

For budget $M = 20$, the answer is 19, which can be reached by buying items $(8 + 10 + 1)$, $(6 + 10 + 3)$ or $(4 + 10 + 5)$.

On the other hand, if the budget $M = 9$, the answer is "no solution", because the minimal possible budget is 10, reached by buying items $(4 + 5 + 1)$.

Problem Example: Wedding Shopping – UVA 11450

Sample case 1: $C = 3$

Class	1	2	3	4
0	6	4	8	
1	5	10		
2	1	5	3	5

This looks like a search problem! Can we do a greedy search?

One approach: For each class, choose most expensive item: $(8 + 10 + 1)$ – It works! ... Not if $M = 12$...

How about divide and conquer?

Wedding Shopping (11450) – Complete search

How can we enumerate the search space? (all the solutions)

Recursive Approach

- Function shop(m,g) discovers the best item k in class C_g that can be bought with total money m
- For each k in C_g, the value for that choice is $V_k = \text{cost}_k + \text{shop}(m - \text{cost}_k, g + 1)$
- shop(m,g) returns the maximum $V_k =< m$;
- shop(m,$|C|$), when we pass the last item, returns 0;

This works! ... but TLE.

Wedding Shopping (11450) - Complete search

Time Limit Exceeded:

20 categories of items, with 20 items each, a complete search will take: 20^{20} operations.

Problem: Too many overlapping subproblems

Sample case 2: $C = 4$

Class	1	2	3	4
0	6	4	8	12
1	4	6	6	2
2	1	5	1	5
3	2	4	6	2

How many times *shop(15,3)* is called?

Every time we call *shop(15,3)*, the solution is the same.

Wedding Shopping – the DP approach

Since the problem has an overlapping subproblem structure, we can think about using a DP approach.

The first step of using DP is constructing the state table.

How many states do we need?

Since we know the unique states are *(Money,Class)*, we need to make a table of *MxC*.

Since $0 \leq M \leq 200$ and $0 < C \leq 20$, our table will have $201 * 20 = 4020$ states.

Only 4020 states! This looks promising!

Wedding Shopping – the DP approach

Now that we have our state table, there are two approaches for building a DP solution:

- The top-down approach:
  Use the state table as a look-up table. Save the result of visited states, and do not re-calculate those.

- The bottom-up approach:
  Fill the base-states of the table, and iteratively fill the other states transitioning from the base.

Wedding Shopping – top-down DP

```
memset(table,-1,sizeof(table))   ** DP <3 memset **

shop(m,g):
    if (m < 0) return -INF
    if (g == C) return M - money
    if (table[m][g]) != -1 return table[m][g]   **NEW**
    return table[m][g] = max(shop(m-price[g][k],g+1)
                             for every k)
```

To implement top-down DP, simply add a table check to a complete recursive search.

Make sure that your states are independent from the parent. If they are not, you need to rethink your state table.

Wedding Shopping – bottom-up DP

Algorithm:

- Prepare a table with the problem states (same as top-down);
- Fill the table with base case values;
- Find the topological order in which the table is filled;
- Fill the non-basic cases;

The main problem for bottom-up DP is finding the base cases and the ordering between the cases. In some good cases, the ordering is just a list of nested loops!

Wedding Shopping – bottom-up DP

Example: M=10, G1=(2,4), G2=(4,6), G3=(1,3,2,1)

Money	0	1	2	3	4	5	6	7	8	9	10
G0											1
G1											
G2											
G3											

- Initial state: With 0 items, we can reach money 10;
- For each reachable money state in G_i, we can reach money state in G_{i+1} according to the costs of the items.
- If any money state in G_C is reachable, the problem is solvable.
- Solution is M - minimal reachable state;

Wedding Shopping – bottom-up DP

Example: M=10, G1=(2,4), G2=(4,6), G3=(1,3,2,1)

Money	0	1	2	3	4	5	6	7	8	9	10
G0											1
G1							1		1		
G2											
G3											

- Initial state: With 0 items, we can reach money 10;
- For each reachable money state in G_i, we can reach money state in G_{i+1} according to the costs of the items.
- If any money state in G_C is reachable, the problem is solvable.
- Solution is M - minimal reachable state;

Wedding Shopping – bottom-up DP

Example: M=10, G1=(2,4), G2=(4,6), G3=(1,3,2,1)

Money	0	1	2	3	4	5	6	7	8	9	10
G0											1
G1							1		1		
G2	1		1		1						
G3											

- Initial state: With 0 items, we can reach money 10;
- For each reachable money state in G_i, we can reach money state in G_{i+1} according to the costs of the items.
- If any money state in G_C is reachable, the problem is solvable.
- Solution is M - minimal reachable state;

Wedding Shopping – bottom-up DP

Example: M=10, G1=(2,4), G2=(4,6), G3=(1,3,2,1)

Money	0	1	2	3	4	5	6	7	8	9	10
G0											1
G1							1		1		
G2	1		1		1						
G3	1	1	1	1							

- Initial state: With 0 items, we can reach money 10;
- For each reachable money state in G_i, we can reach money state in G_{i+1} according to the costs of the items.
- If any money state in G_C is reachable, the problem is solvable.
- Solution is M - minimal reachable state;

Wedding Shopping – bottom-up DP

M=10, cost[1] = [2,4], cost[2] =(4,6), G3=(1,3,2,1)

Money	0	1	2	3	4	5	6	7	8	9	10
g = 0											1
g = 1							1		1		
g = 2	1		1		1						
g = 3	1	1	1	1							

```
memset(table,0,sizeof(table))

table[0][10] = 1

for g in (0:G-1)
  for i in (0:M):
    if table[g][i] == 1:
       for k in C[g+1]:
          table[g+1][i-cost[k]] = 1
```

DP: Top-down or Bottom-up?

Top-Down

Pros:

Easy to implement starting from a recursive search. Only compute states if necessary.

Cons:

Recursive calls are slower if there are many levels, state table memory usually can't be reduced.

Bottom-Up

Pros:

Faser if many sub-problems are visited. Can sometimes save memory space by keeping only s and s+1 in memory.

Cons:

Not very intuitive. If there are X states, all states values will have to be checked.

DP: What about the decision set?

In the previous example, we only cared about the final value of the solution. What if we want to know the exact items in an optimal solution?

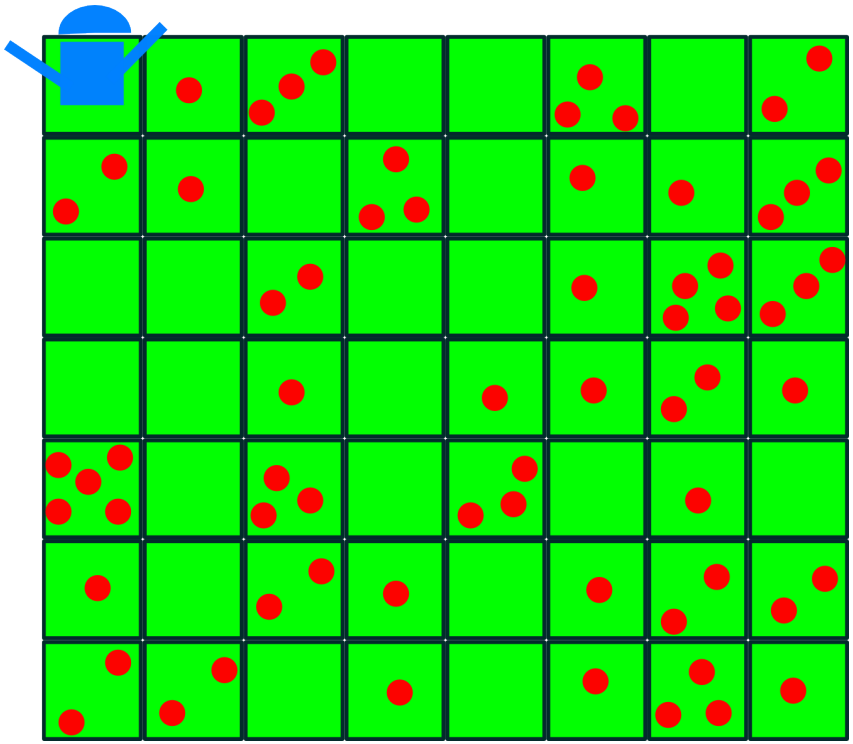
Together with the *state* table, keep a second *parent* table. Every time you update a value on the state table, write in the parent table which cell(s) was used to update that value.

Pay attention to whether the problem requires you to print the first solution, or the last, or a solution with some particular properties.

Let's see some code in the next DP example.

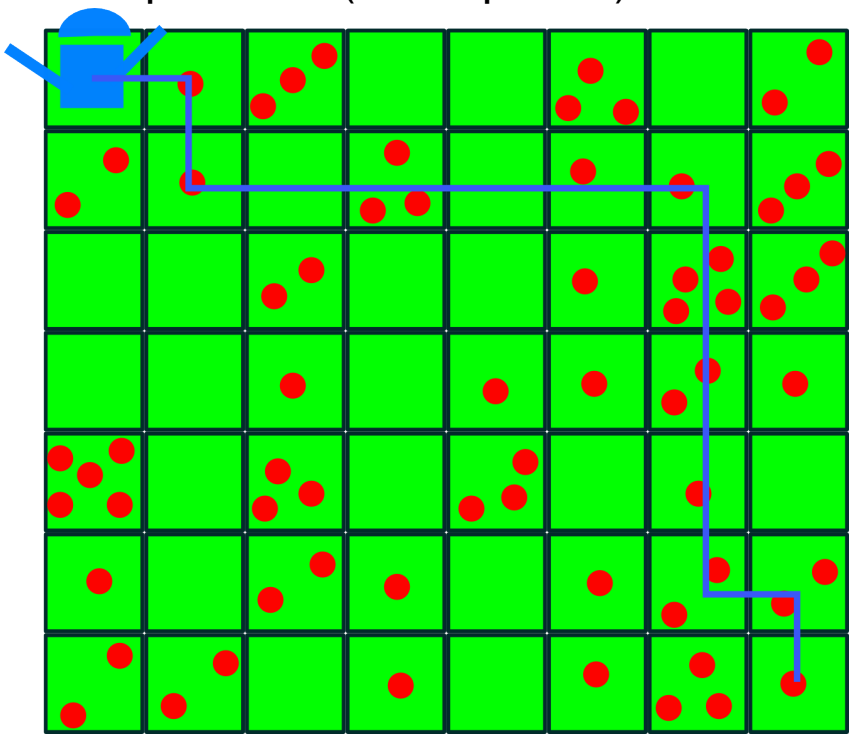
Example 2: Apple Field – not on UVA

Farmer Tanaka is growing apples in his field. He has a robot to collect the apples, but the robot can only walk east and south. If you know how many apples there are in each tree of his $m * n$ field, can you calculate the path that the robot should make to collect most apples?



Example 2: Apple Field – complete search

One possible (non-optimal) solution

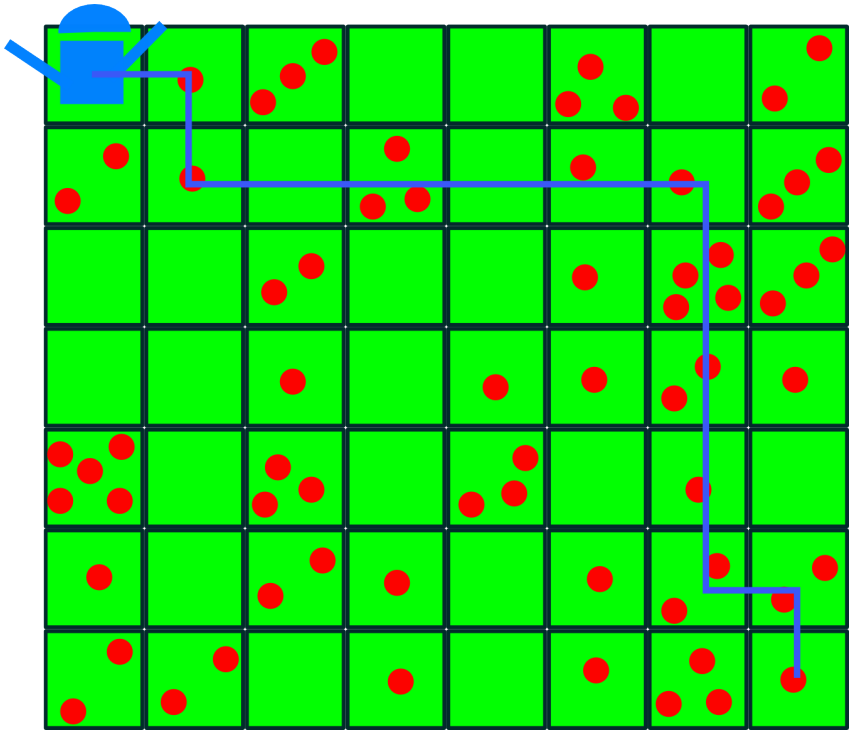


For every square, the robot can choose to go right (east) or down (south). How many possible paths exist?

And how many overlapping solutions are there?

Example 2: Apple Field – overlapping solutions

One possible (non-optimal) solution



If the robot reached position (x,y) with k apples, it does not matter how it did it. So the path from (x,y) to the goal is independent from the parent path.
This smells like DP! Let's solve it with a bottom up approach.

Example 2: Apple Field – Bottom up DP

- tables: We want to maximize the number of apples at the goal. Let's try a m*n position table. At each cell, we want to store the maximum number of apples achievable at that cell.

  We also want a second table, *path*, which will store the path used.

- initial condition: We can fill the top row (x,1) of the table with the sum of apples from 1 to x. And we can fill the Left column with the sum of apples from 1 to y.

  Alternatively, we can add an additional dummy "-1" column/line and fill it with zeros.

- transition: For each cell x,y, max(x,y) is either max(x-1,y) + apple(x,y) or max(x,y-1) + apple(x,y)

Example 2: Apple Field – Let's see some code

```
// Assume apple[0,] and apple[,0] are all zeroes.
int apple[m][n]// Input.
int sum[m][n]// DP table, set to 0 using memset
int parent[m][n][2]// Path table, set to 0 using memset

for i in (1:m):
  for j in (1:n):
    sum[m][n] = apple[m][n] + max(sum[m][n-1],sum[m-1][n])
    if (sum[m][n-1] > sum[m-1][n]):
       parent[m][n][0] = m, parent[m][n][1] = n-1
    else:
       parent[m][n][0] = m-1, parent[m][n][1] = n
```

Summary

Dynamic Programming is a smart way to do complete search, when a large part of the search space is overlapping.

- Top-down DP: recursive complete search with memoization;
- Bottom-up DP: state table with iterative completion recurrency;

Next Class

- DP on non-classical problems
- Relationship between DP and DAG
- Other “cool” DP techniques.

Read more about DP

- http://people.csail.mit.edu/bdean/6.046/dp/
- http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=dynProg

This Week's Problems

- Jill Rides Again
- Maximum Sum on a Torus
- Strategic Defense Initiative
- Is bigger smarter?
- Ferry Loading
- Unidirectional TSP
- Flight Planner
- e-Coins