GB21802 - Programming Challenges

Week 4 - Dynamic Programming (Part II)

Claus Aranha
caranha@cs.tsukuba.ac.jp

College of Information Science

2015-05-20,23

Last updated May 19, 2016

Last Week Results

Special Notes

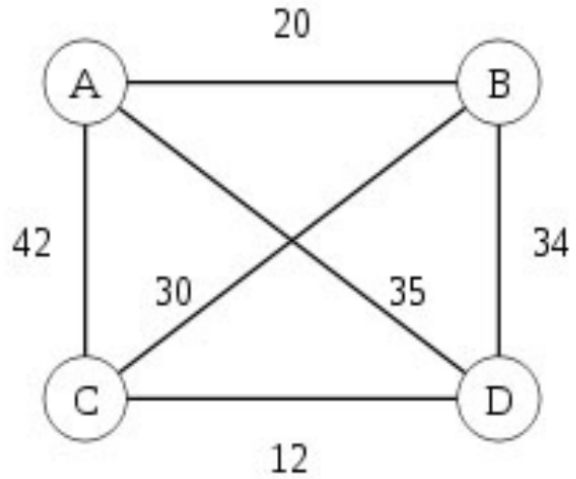
Outline

- Using DP on the Travelling Salesman Problem

Traveling Salesman Problem with DP

Problem Definition

Given *n* cities and their pairwise distances (*nxn* dist matrix), compute the cost of a tour that starts from any city *s*, visits all cities *once*, and returns to *s*.



	A	B	C	D
A	0	20	42	35
B	20	0	30	34
C	42	30	0	12
D	35	34	12	0

In the graph above, we have $n = 4$ cities and $n! = 24$ possible tours (permutations). One minimal tour is A-B-C-D-A, with cost $20 + 30 + 12 + 35 = 97$.

Image from Steven Halim – “Competitive Programming”

TSP – Complete Search Approach

A complete search for the TSP tests all city permutations. For each permutation, the complete path of the tour is calculated.

```
#include<algorithm>
int c[4] = {0,1,2,3};
do {
  cost = 0;
  for (i=0;i<4;i++)
    cost += pathcost[c[i]][c[(i+1)%4];
} while (next_permutation(c,c+4));
```

Overlapping subproblems

Complete search costs $O(n!n)$, so the biggest n we can realistically handle on a programming competition is 10 or 11.

However, many subpaths in the tour are repeated. For example:

- A-B-(n-2)
- B-A-(n-2)

Have $(n-2)!$ repeated subproblems. A DP approach should be possible!

TSP – DP Approach

Basic Recurrence

- If all cities are visited, return the cost from the last city to the first one.
- If not all cities are visited, try each unvisited city and select the one with the minimum cost.

State Table

Note that this recurrence requires the set of cities already visited and the city currently being visited.

This means that our state is *(visitset,city)*. Our table has size $2^n n$, and each state can be calculated in time $O(n)$, so our DP-TSP has complexity $O(n^2 2^n)$.

Not a huge improvement, but now we can solve problems up to size $n \leq 16$.

TSP – DP Code

```
int dp[n][1<<n] = -1
start = 0

visit(p,v):
   if (v == (1<<n) - 1):
      return cost[p][start]
   if dp[p][v] != -1
      return dp[p][v]

   tmp = MAXINT
   for i in n:
       if not(v && (1 << i):
           tmp = min(tmp,
                     cost[p][i] + visit(i, v | (1<<i)))

   dp[p][v] = tmp
   return tmp
```

UVA 10943 – How do you add?

Problem Description

Given an integer n, how many ways can you add K integers ($0 \leq i \leq n$) so that their sum is equal to n?

Example: $n = 20, K = 2$
$0 + 20, 1 + 19, 2 + 18, \ldots, 20 + 0$

What is the recurrence?

- When K = 1, there is only one way to add to n.
- When K = i > 1, we can test all numbers X between 0 and n, and our result will be the sum of all $(n - X, K - 1)$ sub problems.

How do you add? – Recurrence Example

Recurrence Example: $n = 10, K = 3$

ways(10,3) =	(0,ways(10,2)) +
	(1,ways(9,2)) +
	(2,ways(8,2)) +
	...
	(10,ways(0,2))
ways(8,2) =	(0,ways(8,1)) +
	(1,ways(7,1)) +
	...
	(8,ways(0,1))
=	9
ways (10,3) =	$11 + 10 + 9 + 8 + \ldots + 1 = 66$

How do you add? – Bottom-Up DP

```
dp[maxK][maxN]
tsum[2][maxN]

for (i in maxN):
   tsum[1][i] = i+1
   dp[1][i] = 1

for (i in K):
   tsum[0] = tsum[1]
   for (j in maxN):
      dp[i][j] = tsum[0][j]
      tsum[1][j] = tsum[1][j-1] + tsum[0][j]
```

Time complexity is $O(nK)$

How do you Add? – Mathematical Approach

This problem can also be seen as solving the recurrence/closed form of a binomial combination C. We will come back to recurrences in a later class.

Thinking About DP – 1

DP can come in many forms other than its classical problems. You can solve these problems following the procedure that we have seen so far:

1. Elaborate a recursive, full search solution.
2. Define the distinct states and the transitions
3. Write a program for memoizing the state table (top-down) or constructing the table (bottom up).

DP has an intrinsic relationship with Directed Acyclic Graph (DAG). States are mapped to vertexes, transitions to edges. We will explore this relationship more in the future.

Thinking About DP – 2

Common ways to imagine DP states:

Position in the problem state

- The original problem is an array of values: $\{x_1, x_2, \ldots, x_n\}$
- Subproblems based on position in the array:
  Suffix: $\{x_1, x_2, \ldots, x_{n-1}\} + x_n$
  Prefix: $x_1 + \{x_2, x_3, \ldots, x_n\}$
  Two sub-problems: $\{x_1, x_2, \ldots, x_i\} + \{x_{i+1}, x_{i+2}, \ldots, x_n\}$
- Can be generalized for 2D (x,y): Consider the size of the table!

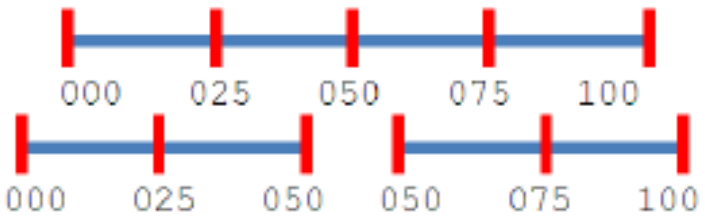
UVA 10003 – Cutting Sticks

Problem Description

Given a stick of length $1 \leq l \leq 1000$ and $1 \leq n \leq 50$ cuts (positions) to be made, the cost of a cut is given by the length of the stick being cut.

Find a cutting sequence that minimizes the cost of cutting the stick.

Example: $l = 100, n = 3, cuts = \{25, 50, 75\}$



- Sequence 1: 25, 50, 75. Cost: 100 + 75 + 50 = 225
- Sequence 2: 50, 25, 75. Cost: 100 + 50 + 50 = 200

What is the recurrence to find the minimum cost?

Cutting Sticks – Recurrence

Basic Idea: Price(start,end) = end-start + Price(start,cut) + Price(cut,end)

Problem: Using start/end, we have 1000*1000 states.

Solution: We use cut indexes instead! (50*50 states)

Recurrence using cut index:

- Price(i,i+1) = 0
- Price(i,j) = min(Size[j] - Size[i] + Price(i,k) + Price(k,j))
  (for all $i < k < j$)

Note that the recurrence costs $O(n)$, and the table size is $O(n^2)$, so the total cost is $O(n^3)$

DP on Math Problems

Many math problems can be implemented as DP:

- Combinatoric problems often have recursive formulas, and overlapping subproblems;
  - Fibonacci Number: $f(n) = f(n-1) + f(n-2)$
- Probability problems often require you to search the entire probability space (tree). These trees usually have overlapping branches.
- Maths problems on static data (sum, min, max)

DP on Strings

- edit distance, substring manipulation, etc.
- Usually, we don't send the string in the recurrence, but indexes on the string

DP Issues

- Many DP problems look like “non DP” problems.

Example

Select positions for a set of flags that cover a certain radius, in order to maximize area coverage.

The area calculation requires geometry, but the flag selection is usually DP.

- Some problem have sub-problems, but they are not overlapping. (In this case, DP will not work, but maybe Divide and Conquer?)

That's all for DP!

This is what we've seen for weeks 3 and 4:

- DP = Complete Search + State table;
- Good when many overlapping subproblems exist;
- Top-Down (recursive) and Bottom-Up (nested loops);
- Classical DP problems;
- (some!) non-classical DP problems;

For the next two weeks, the theme will be graph algorithms!

Problems for Week 4

- Collecting Beepers
- Shopping Trip
- Bar Codes
- Cutting Sticks
- String Popping
- Divisibility
- Marks Distribution
- Squares

World Finals Problem - 2016 Problem C

Problem Description – Ceiling Function

Given n arrays with K values each, each array is organized in a binary search tree in the order of input.

In other words, the first element is the root, the second is the left child of the root if smaller, right child if bigger, so on.

Count the number of different trees generated by the input data.