CS472 Module 6 Part B - Optimization Problems

Athens State University

March 31, 2014

Outline

Contents

1 What do we mean?

Optimization Problem

- Multiple candidate solutions
- Candidate solution has a value associated with it
- Solution to the problem is a candidate solution with an optimal value
- Minimum or Maximum

All-pairs Shortest Path Problem

- Optimization problem
- Candidate Solution: path from one vertex to another
- Value of candidate solution: length of the path
- Optimal value: Minimum length
- Possibility of multiple shortest paths

Three steps to a dynamic programming algorithm

1. *Establish* a recursive property that gives the optimal solution to an instance of the problem
2. Compute the value of an optimal solution in a `bottom-up` fashion
3. Construct an optimal solution in a `bottom-up` fashion

Can we use dynamic programming for all optimization problems?

The Principle of Optimality

The *principle of optimality* applies to a problem if an optimal solution to an instance of the problem always contains optimal solutions to all sub-instances of the problem.

Consider the shortest paths problem

If v_k is a node on an optimal path from v_i to v_j, then the sub-paths v_i to v_k and v_k to v_j are also optimal paths.

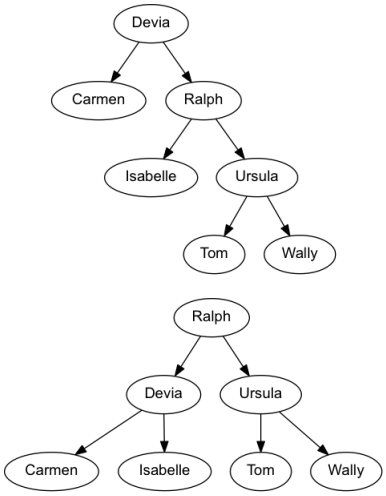
2 Case Study: Optimal Binary Search Trees

Binary Search Trees

Binary search tree A binary tree of items (called keys) that come from an ordered set such that

- Each node contains one key
- Keys in the left sub-tree of a given node are less than or equal to the key in that node
- The keys in the right sub-tree of a given node are greater than or equal to the key in that node

BSTs- Which is better? Why?



Optimal BSTs

Search time Number of comparisons required to locate a key in a BST

- Assuming a set K of n keys in order and p_i being the probability of K_i being the key for which we search, then the average search time for finding a key in a given tree will be

$$\sum_{i=1}^{n} c_i p_i$$

  where c_i is the number of comparisons for K_i

- Our problem is to find the best search tree; i.e., the BST that minimizes

$$\sum_{i=1}^{n} c_i p_i$$

Optimal BSTs and the Principle of Optimality

- The issue with this problem is that the number of possible trees on a set of n elements is exponential in n
- But it can be shown that if a tree is optimal for a given probability distribution, then its left and right sub-trees will be optimal for subsets of that probability distribution
- We can take advantage of this fact to write an algorithm for finding the optimal BST for a set of keys and associated probabilities of finding each of those keys

Algorithm for finding an optimal BST

Algorithm 1: Optimal Binary Search Tree Algorithm

Input: n: the number of keys, and $p(i)$: probability of searching for the ith key
Output: A value with the average search time for the optimal BST and a 2-d array that can be used to construct the optimal tree

```
for i ← [1..n] do
    A[i, i-1] ← 0; A[i, i] ← p[i];
    R[i, i] ← i; R[i, i-1] ← 0;
A[n+1][n] ← 0;
R[n+1][n] ← 0;
for d ← [1..(n-1)] do
    for i ← [1..(n-d)] do
        j ← i + d; A[i,j] ← min_{i≤k≤j}(A[i, k − 1] + A[k + 1, j]) + Σ_{m=i}^{j} p_m;
        R[i,j] = the value of k that produced the minimum;
```

Algorithm 2: Build Optimal Search Tree

Input: Array K containing n keys in order, and the 2-d array from the previous algorithm
Output: A tree containing the optimal BST

```
keyIndex ← R[i,j];
if keyIndex = 0 then
    return (an empty tree);
else
    p ← a new tree node;
    p->key ← K[keyIndex];
    p->left ← optimal_tree (i, keyIndex - 1);
    p->right ← optimal_tree (keyIndex + 1, j) return (p);
```

3 Case Study: Traveling Salesman Problem

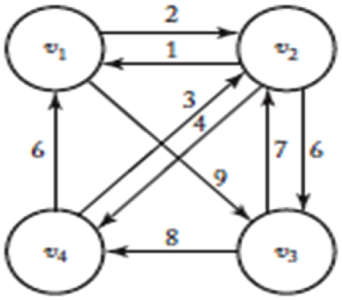
The Traveling Salesman Problem

- You want to make a trip that includes a number of cities
- Each city is connected by a direct flight that has a certain cost
- You want to find the combination of flights that allows you to visit each city once and return home with the best total cost

Representation

- We can represent this problem as a weighted graph with each node representing a city
- The weight of each edge between two nodes represents the cost
- The cost of going from node u to node v can be different than the cost of going from node v to node u
- We define a *tour* (also known as a *Hamiltonian circuit*) as being a path through the graph that passes through each node only once
- An `optimal tour` is a tour with minimum length

Example



$$length[v_1, v_2, v_3, v_4, v_1] = 22$$
$$length[v_1, v_3, v_2, v_4, v_1] = 26$$
$$length[v_1, v_3, v_4, v_2, v_1] = 22$$

- For all tours, the 2nd node can be any of $n-1$ nodes
- ... the 3rd node can be any of $n-2$ nodes
- ... and the nth node can only be one node
- So the total number of tours is:

$$(n-1) * (n-2) * \ldots * 1 = (n-1)!$$

Can we use dynamic programming to solve the TSP?

Observe: If v_k is the 1st node after v_1 on an optimal tour, the sub-path from v_k to v_1 must be the shortest path that passed through each of the other nodes exactly once.

- Represent the weighted graph in an adjacency matrix and let

W	The adj. matrix representation of our graph
L	Length of an optimal tour
V	set of all nodes
A	a subset of V
$D[v_i][A]$	length of shortest path through v_i to v_1 that passes through each node in A exactly once

A dynamic programming recurrence for the TSP

- Consider the set $V_{int} = V - \{v_1, v_j\}$ be set of nodes without our starting point and destination
  - Note this set contains all nodes except for v_1 and v_j
  - And remember the principle of optimality applies
- So,

$$L = \min_{2 \leq j \leq n} (W[1, j] + D) + D[v_j][V_{int}]$$

- And in general

$$D[v_i][A] = \min_{j:v_j \in A}(W[i, j] + D[v_j][A - v_j]), \text{if } A \neq \emptyset$$
$$D[v_i][\emptyset] = W[i, 1]$$

The DP algorithm for the TSP

Algorithm 3: DP algorithm for the TSP

Input: An adj. matrix W for a weighted directed graph and the number n of nodes in the graph
Output: The length of a minimal tour and a 2-d graph showing how to construct that tour

```
for i ← [2..n] do
    D[i][∅] ← W[i,1];
for k ← [1..(n-2)] do
    for all subsets of A ⊆ V − {v1} of size k do
        for i s.t. i ≠ 1 and v_i is not in A do
            D[i][A] ← min_{j:v_j∈A}(W[i, j] + D[j][A − {v_j}]);
            P[i][A] ← value of j that gave the minimum;
D[1][V − {v1}] ← min_{2≤j≤n}(W[1, j] + D[j][V − {v1, v_j}]);
P[1][V − {v1}] ← value of j that gave the minimum;
minlength ← D[1][V − {v1}];
```

OK, so what's the point?

- Remember that we have claimed this problem is NP-complete
- Problem: this algorithm is $T(n) = (n-1)(n-2)2^{n-3} \in \Theta(n^2 * 2^n)$
- ... and it requires $\Theta(n * 2^n)$ space
- But, recall that the brute force algorithm is factorial

OK, so what's the point?

- Suppose that we're trying to solve find a 20 node tour in a graph that has a connection from each node to every other node
- If the basic operation in either the brute force or dynamic programming algorithm takes 1 microsecond to execute:

  Brute force: 19! microseconds or 3857 years

  DP algorithm: $(20-1) * (20-2) * 2^{20-3}$ microseconds or 45 seconds
- The memory required for the DP algorithm would $20 * 20^{20}$ or 20,971,520 array slots

4 Key points

Key Points

- What is an optimization problem?
- What is the principle of optimality?
- Use of DP to solve the optimal BST problem
- What is the Traveling Salesman Problem?
- How can use DP to solve the TSP?