Chapter 3
Dynamic Programming

Preview

Overview

Dynamic programming is similar to the divide-and-conquer approach in that an instance of a problem is divided into smaller instances. In dynamic programming, we solve the small instances first, store the results, and look them up when we need them instead of recomputing them.
Dynamic programming is a **bottom-up** approach since the solution is constructed from the bottom up in the array. There are two steps in the development of this approach. First, *establish* the recursive property that gives the solution to an instance of the problem. Second solve an instance of the problem in a *bottom-up* fashion by solving smaller instances first.

Lecture Notes

The Binomial Coefficient

The **binomial coefficient** is discussed in Section A.7 of Appendix A. A simple example of dynamic programming, binomial coefficient, is given by this formula:

$$\binom{n}{k} = n!/\,k!\,(n-k)! \qquad \text{for } 0 \le K \le n.$$

We cannot compute the binomial coefficient directly from this definition because *n*! is very large, even for moderate values of *n*.

We should eliminate the need to compute *n*! or *k*! by using the recursive property and we will get this equation:

[See Binomial Co-efficient equations on page 96]

The problem with the above recursive definition is that the same instances are solved in each recursive call. We can develop a more efficient algorithm using dynamic programming. Using the steps (for dynamic programming) defined above and the recursive definition we established. Take an array B such that:

$$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j] & \text{for } 0 < j < i \\ 1 & \text{for } j = 0 \text{ or } j = i \end{cases}$$

Here we *established* the recursive property, which is Step1 in dynamic programming approach. Step 2 should be to *solve* an instance of the problem in bottom-up approach by computing values for B [i] [j] starting from B [0] [0].

By using dynamic programming instead of divide-and-conquer, we have developed a much more efficient algorithm. In dynamic programming, we use the recursive property to iteratively solve the instances in sequence, starting with the smallest instance. In this way we solve the smallest instance just once. Dynamic programming is a good technique to try when divide and conquer leads to an inefficient algorithm.

Floyd’s Algorithm for Shortest Path

Air travelers encounter a common problem when they must fly from one city to another when a direct flight does not exist. Floyd’s algorithm addresses some of these problems.

Before starting with Floyd’s algorithm for determining shortest path, let us review some graph theory. In a pictorial representation of a graph, circles represent **vertices**, and a line from one circle to another represents an **edge** (sometimes also called an arc). If each edge has a direction associated with it, the graph is called a directed graph, or **digraph**. If the edges have values associated with them, the values are called **weights,** and the graph is called a **weighted graph.** A **path** is a sequence of adjacent vertices in a graph, while a **simple-path** is a path but with distinct vertices (that is you can not pass through the same vertex twice). A **cycle** is a simple path with three or more vertices such that the last is adjacent to the first. A graph is said to be **acyclic** if it has no cycles and **cyclic** if it has one or more cycles. A path is called **simple** if it never passes through the same vertex twice. The **length** of a path in a weighted graph is the sum of the weights on the path. In an **unweighted** graph, it is the number of edges in the path.

A problem that has many applications is finding the shortest path from each vertex to all the other vertices. The shortest path must be a **simple path**, or a path that does not pass through the same vertex twice**.** A *shortest path* must be simple path but there may be more than one simple path from one vertex to another we should choose the one with the minimum value.

[See Figure 3.2 on page 101]

The Shortest Paths problem is an **optimization problem**. There can be more than one candidate solution to an instance of an optimization problem. Each candidate solution has a value associated with it, and a solution to the instance is any candidate solution that has an optimal value. Depending on the problem, the optimal value is either the maximum or minimum of these lengths.

Because there can be more than one shortest path from one vertex to another, the problem is to find any one of the shortest paths. The obvious algorithm is to determine for each vertex. This algorithm is worse than exponential time. Using dynamic programming, we can create a cubic-time algorithm. First we develop an algorithm that determines the lengths of only the shortest paths. After that, we modify it to produce shortest paths as well.

Dynamic Programming and Optimization Problems

The first two steps in the development of a dynamic programming algorithm are the determination of the shortest paths and the construction of the shortest paths. The construction of an optimal solution is the third step. Specifically, the steps are as follows:

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

Although it may seem that any optimization problem can be solved using dynamic programming, it is not true. The **principle of optimality** must apply in the problem. This principle states that an optimal solution to an instance of a problem always contains optimal solutions to all **subinstances**. Optimal solutions to the subinstances can be any optimal solutions. The principle of optimality does not apply in every problem.

Quick Check

1. Dynamic programming is similar to divide and conquer in that we find a(n) ____ property.
   Answer: recursive
2. A graph with weighted edges is called ____ graph.
   Answer: weighted

3. An acyclic graph is a graph containing one or more cycles. (True or False)
   Answer: False
4. The value of the optimal solution is computed in a(n) ____ fashion.
   Answer: bottom-up

Chained Matrix Multiplication

If we want to apply a 2 X 2 matrix times a 3 X 4 matrix, the resultant matrix is a 2 X 4 matrix: It takes three elementary multiplications to compute each item in the product.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} * \begin{bmatrix} 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 29 & 35 & 41 & 38 \\ 74 & 89 & 104 & 83 \end{bmatrix}$$

Matrix multiplication is an **associative** operation, meaning that the order in which we multiply does not matter.
Considering the multiplication of these four matrices:

A	*	B	*	C	*	D
20 X 2		2 X 30		30 X 12		12 X 8

We have these elementary multiplications:
***A(B (CD))** requires 3680 multiplications. 30 * 12 * 8 + 2 * 30 * 8 + 20 * 2 *8 = 3, 680*
***(AB) (CD)** requires 8880multiplications. 20* 2 * 30 + 30 * 12 * 8 + 20 * 30 *8 = 8, 880*
A((BC)D) requires 1232 multiplications. *2* 30 * 12 + 2 *12 * 8 + 20 * 2 * 8 = 1, 232*
((AB)C)D requires 10320 multiplications. 20 * 2 * 30 + 20 * 30 * 12 + 20 * 12 * 8 = 10, 320
(A(BC))D requires 3120 multiplications. *2 * 30 * 12 + 20 * 2 * 12 + 20 * 12 * 8 = 3, 120*

The goal is to develop the optimal order for multiplying *n* matrices. The optimal order depends only on the dimensions of the matrices. The **brute-force** algorithm is to consider all possible orders and take the minimum. This is a very inefficient method.

Optimal Binary Search Tree

A **binary search tree** is a binary tree of items that come from an ordered set, such that
1. Each node contains one key.
2. The keys in the left sub-tree of a given node are less than or equal to the key in that node.
3. The keys in the right sub-tree of a given node are greater than or equal to the key in that node.

The **depth** or the **level** of a node in a tree is the number of edges in the unique path from the root to the node. The *depth of a tree* is the maximum depth of all nodes in the tree. A tree is said to be **balanced** if the depth of the two sub-trees of every node never differ by more than 1.

Ordinarily, a binary search tree contains records that are retrieved according to the values of the keys. The goal is to organize the keys so that the average time it takes to locate a tree is minimized. A tree that is organized in this fashion is said to be **optimal**. In an optimal tree, all the keys have the same probability of being the **search key**. The number of comparisons done by procedure search to locate a key is called the **search time**.

Our goal is to determine a tree for which the average search time is minimal.

Let Key_1, Key_2, . . . , Key_n be the n keys in order, and let pi be the probability that Key_i is the search key.

If c_i is the number of comparisons needed to find Key_i in a given tree, the average search time for that tree is:

$\sum$ cipi from i = 1 to i = n.

[See Figure 3.11 on page 124]

The fifth tree is optimal.

The Traveling Salesperson Problem

Suppose a salesperson is planning a sales trip that includes 20 cities. Each city is connected to some of the other cities by a road. To minimize travel time, we want to determine a shortest route that starts at the salesperson's home city, visits each of the cities once, and ends up at the home city. This problem of determining a shortest route is called the Traveling Salesperson problem.

An instance of this problem can be represented by a weighted graph, in which each vertex represents a city. We generalize the problem to include the case in which the weight (distance) going in one direction is different from the weight going in another direction. We assume that the weights are nonnegative numbers. A **tour** in a directed graph is a path from a vertex to itself that passes through each of the other vertices only once. An **optimal tour** in a weighted, directed graph is such a path of minimum length. The Traveling Salesperson problem is to find an optimal tour in a weighted directed graph when at least one tour exists. Dynamic programming can be applied when the principle of optimality applies.

No one has ever found an algorithm for the Traveling Salesperson problem whose worst-case time complexity is better than exponential. Yet, no one has ever proved that the algorithm is not possible.

Sequence Alignment

The text has an example of how to apply a dynamic program to molecular genetics. If necessary, refer to the text to review some basic concepts of genetics.

Consider the same DNA sequence in every individual of a population (species). In each generation, each site in the genetic sequence has a probability of undergoing a mutation in each gamete that produces an individual for the next generation. A possible result is substitution at a given site of one base by another base in most or all if the population. Another possibility is that a speciation event will occur, and the members will separate into two distinct species.

When we compare homologous sequences from two individuals in two different species, we must first align the sequences because one or both of the sequences may have undergone insertion and/or deletion mutations since they diverged.

When we include a dash (–) in an alignment, it is called inserting a gap. It indicates that either the sequence with the gap has undergone a deletion, or the other sequence has undergone an insertion. Once we specify for gaps and mismatches, we can determine the optimal alignment. Checking all possible alignments is a nearly impossible task.

To make the task less impossible, we can assign penalties. In the example, we assign a penalty of 1 for a mismatch, and a penalty of 2 for a gap. We call the sum of all penalties in the alignment the **cost.**

Once we specify the penalties for gaps and mismatches, it is possible to determine the optimal alignment. The algorithm in the text gives the cost of the optimal alignment; it does not produce an optimal alignment.

To solve this problem using dynamic programming, we create an *m*+1 by *n*+1 array. The extra character gives the upward iteration scheme a starting point.

[See Figure 3.19 on page 143]

The completed array shows the optimal alignment.

[See Figure 3.20 on page 145]

Quick Check

1. To multiply an i * j matrix times a j * k matrix using the standard method we should do ___ multiplications.
   Answer: i * j * k
2. The depth of a tree is the sum of depths of all nodes. (True or False)
   Answer: False
3. The number of comparisons done by procedure search to locate a key is called the ____.
   Answer: search time
4. A Hamiltonian circuit also stands for a(n) ____.
   Answer: tour
5. Dynamic programming applies in the Traveling salesman problem. (True or False)
   Answer: True

Classroom Discussion

- Matrix multiplication is associative. Discuss and give an example.
- Sub-trees of optimal trees are themselves optimal trees. Discuss

Homework

Assign Exercises 5, 12, 26, and 31.

Keywords

- **Acyclic graph –** a graph that has no cycles.
- **Adjacency matrix –** a kind of representation for a graph (discussed above).
- **Adjacent vertices –** if they are incident on same edge.
- **Associative –** the order of operations does not matter.
- **Balanced tree –** a tree such that the depth of the two sub-trees of every node never differ by more than one.
- **Binary search tree –** a binary of items (keys) with some conditions (discussed above).
- **Binomial coefficient –** a binomial raised to a large power**.** Check appendix A for the general equation.
- **Bottom-up –** the solution of the instance is in constructed from the bottom up in the array.
- **Brute-force algorithm –** an algorithm that tries every one of a range of possible solutions.
- **Candidate solution–** a path from one vertex to another.

- **Cost –** the sum of all penalties in an alignment.
- **Cycle –** a path from a vertex to itself.
- **Cyclic graph –** a graph that contains cycles.
- **Depth of a node –** is the number of edges in the unique path from the root to the node.
- **Depth of a tree –** maximum depth of all nodes in the tree.
- **Digraph –** a graph with directed edges.
- **Dynamic Programming –** a technique similar to divide and conquer approach in that instance of a problem is divided into smaller instances.
- **Edge (**also called an **arc)–** line from one circle to another in a graph (edge for digraph / arc for graph).
- **Left sub-tree –** sub-tree whose root is the left child of the node.
- **Length –** of a path in a weighted graph is the sum of weights on the path.
- **Level –** same meaning as depth.
- **Optimal tour –** in a weighted directed graph is a tour of optimal length.
- **Optimal tree –** a tree in which the keys are organized so that the average time it takes to locate a key in minimized.
- **Optimization problem –** is the shortest path problem.
- **Path –** sequence of vertices such that there is an edge/arc from each vertex to its successor.
- **Principle of optimality –** is applicable in a problem if an optimal solution to an instance of a problem always contains optimal solutions to all substances.
- **Right sub-tree –** sub-tree whose root is the right child of the node.
- **Search key** – the data used to identify the information we are searching for in a database.
- **Search time –** number of comparisons done by procedure search to locate a key.
- **Simple path –** a path that does not pass through the same vertex twice.
- **Subinstance –** a part of an instance.
- **Tour –** in a directed graph is a path from a vertex to itself that passes through each of the other vertices exactly once.
- **Unweighted graph –** a graph in which all edges are equal to 1.
- **Vertices –** circles or nodes in a graph.
- **Weights –** values associated with edges.
- **Weighted graph –** a graph that has weighted edges.

Important Links

- http://www.nist.gov/dads/ (Dictionary of Algorithms and Data Structures)
- www.sciencedirect.com/science/journal/01966774 (Journal of Algorithms)
- http://www.talkorigins.org/faqs/genalg/genalg.html (Genetic Algorithms and Evolutionary Computation)