Dynamic Programming

Owomugisha Isaac
215004131
15/U/12351/PS
19th/March/2017

Abstract

This paper is about dynamic programming, an important algorithm design technique. This paper covers what dynamic programming is and shows some examples of how it can be used to solve some problems efficiently. This paper is a result of my fascination with the technique, which I first heard about while studying recursion (and memoization) in my data structures and algorithms class. I decided to delve deeper and I have found that it is indeed quite a useful and beautiful algorithm design technique. In this paper, I present some thoughts on what I have learned.

Most of the results and examples in this paper are borrowed from [1] and [2].

1. Introduction

Dynamic-programming is a general and powerful algorithm design technique. A large class of seemingly exponential problems have a polynomial solution only via dynamic-programming [2].

Just like the divide and conquer method, dynamic programming solves problems by combining solutions to sub problems, however, dynamic programming applies when the sub problems overlap, that is when sub problems share sub problems. When this happens, a divide-and-conquer algorithm would do more work than necessary. A dynamic-programming algorithm solves each sub problem and then saves its answer in a table, thereby avoiding re-computing the answer every time it solves each sub problem. [1]

Dynamic programming is typically applied to optimization problems, that is, problems that might have many possible solutions but we are looking for the optimal solution.

The following quote about the history of dynamic programming is from [John Rust 2006]: (*Richard Bellman (1920 - 1984) received the IEEE Medal of Honor, 1979).* "Bellman ... explained that he invented the name 'dynamic programming' to hide the fact that he was doing mathematical research at RAND under a Secretary of Defense who 'had a pathological fear and hatred of the term research'. He settled on the term 'dynamic programming' because it would be difficult to give a 'pejorative meaning' and because 'it was something not even a congressman could object to.'"

2. The Technique

Dynamic programming can be thought of as a combination of “recursion” and “memoization”.

There are four steps involved in developing a dynamic-programming algorithm [1]:

1. Characterize the structure of an optimal solution (this involves defining a base case and the sub problems)
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.
4. Construct an optimal solution from computed information.

The following section illustrates how these steps apply to computing the n^{th} Fibonacci number, and the dynamic-programming algorithm is compared with the naïve recursive algorithm.

3. Dynamic-Programming example: Computing Fibonacci numbers

Consider the Fibonacci numbers which are defined as follows:

$$F_1 = F_2 = 1$$

$$F_n = F_{n-1} + F_{n-2} \ for\, n > 2.$$

We want an algorithm that takes as input an integer n, and returns as output F_n.

Consider the following algorithm (implemented in python):

```
def fib(n):
        if n <= 2: f = 1
        else: f = fib(n - 1) + fib(n - 2)
        return f
```

The run time of this algorithm is $O(2^n)$. (See [2] for a proof). This is very inefficient. The major drawback of this algorithm is that it computes the same values many times. For example consider $fib(5)$:

$$fib(5) = fib(4) + fib(3)$$

$$fib(4) = fib(3) + fib(2)$$

$$fib(3) = fib(2) + fib(1)$$

It can be seen that $fib(3)$ is called twice (by $fib(4)$ and $fib(5)$). For higher values of n, some values can be computed very many times.

A way to eliminate this re-computation is by storing the values as they are generated in an array. This has the benefit of constant time access (since indexing into an array takes constant time). With this method, the values $fib(1)$ to $fib(n)$ are only computed once each, and the resulting dynamic-programming algorithm runs in $O(n)$ time (a big improvement from the exponential run time we had above).

The algorithm, implemented in python, is shown below:

```
memo = {} #dictionary with keys, k and values = fib (k)
def fib(n):
        if n in memo: return memo[n] #if the value for n is already in memo, return it.
        if n <= 2:f = 1
        else:f = fib(n - 1) + fib(n - 2)
        memo[n] = f
        return f
```

4. Conclusion

In this paper I have given a brief overview about dynamic programming and given an example of how it can improve an algorithm from exponential time to linear time. The topic of dynamic programming is wide and interesting. The reader is referred to [1] for a more thorough coverage.

5. **References**

[1] T.H Cormen, C.E Lieserson, R.L Rivest, C Stein, Introduction to Algorithms 3rd Edition, 2009

[2] Erik Demaine, 6.006 Introduction to algorithms, Fall 2011, Lecture 19: Dynamic Programming I: Memoization, Fibonacci, Shortest Paths, Guessing. Link: http://ocw.mit.edu/courses/elctrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011