Comparison of Two Approaches to Dynamic Programming

Pim van den Broek and Joost Noppen
Department of Computer Sciece
University of Twente
P.O. Box 217
7500 AE Enschede, the Netherlands
pimvdb@cs.utwente.nl, noppen@cs.utwente.nl

Abstract

Both in mathematics and in computer science Dynamic Programming is a well known concept. It is an algorithmic technique, which can be used to write efficient algorithms, based on the avoidance of multiple executions of identical subcomputations. Its definition in both disciplines is however quite different. The aim of this paper is to compare both definitions. It is shown that the computer science approach is more efficient, since it avoids the execution of unneeded subcomputations, whereas the mathematical approach has greater possibilities of reducing memory requirements.

1. INTRODUCTION

Dynamic Programming has been introduced in a mathematical context by Bellmann in [1] and [2]. A discussion of its present use in computer science can be found in [3]. It is an algorithmic technique, which can be used to write efficient algorithms, based on the avoidance of multiple executions of identical subcomputations. In mathematics, it is a technique which transforms optimization algorithms to a backwards iteration form; in computer science it is the technique of tabelation of results of subcomputations.

Let us illustrate this with a simple example.
Consider the Fibonacci sequence 1,1,2,3,5,8,13,21,.... where each number is the sum of its two predecessors. An obvious Java method to compute a Fibonacci number (FN) is

```
int fib (int n)
{  if (n==0) then return 1;
   if (n==1 then return 1;
   return fib (n-1) + fib (n-2);
}
```

The time which is needed by this method to compute the n-th FN is exponential in n. This means that the computation succeeds only for small values of n; it is impossible to calculate fib(100), for example (even if your programming language has no upper bound on integers). The reason for this inefficiency is that all calls of fib() are evaluated from scratch, and so there are multiple calculations of the same FN.

Dynamic Programming will reduce the execution time of the method to a time which is linear in n. In the computer science approach (DP-CS), a table for FNs is maintained. Each time a FN is calculated, it is stored in the table; each time a FN is needed, it is retrieved from the

table if it has been calculated before, otherwise it will be calculated. The Fibonacci method is transformed into

```
int fib (int n)
{  int[] fiblist = new int[n+1];
   fiblist[0] = 1;
   fiblist[1] = 1;
   return fib(n,fiblist);
}

int fib (int n, int[] list)
{  int lookup = list[n];
   if (lookup != 0) return lookup;
   int result = fib (n-1,list) + fib (n-2, list);
   list[n] = result;
   return result;
}
```

In the mathematics approach (DP-M), fib(n) is calculated by a backwards iteration, calculating fib(1), fib(2), fib(3),.....,fib(n-1), fib(n), in this order. To calculate a FN in this sequence, the results of the previously calculated FNs are available.

```
int fib (int n)
{  int[] fiblist = new int[n+1];
   fiblist[0] = 1;
   fiblist[1] = 1;
   for (int i = 2; i<=n; i++)
     fiblist[i] = fiblist[i-1] + fiblist[i-2];
   return fiblist[n];
}
```

Both methods above have linear execution time. Of course, their memory usage can be optimized further.

The remainder of this paper is organised as follows. In section 2 we briefly review DP-CS, and in section 3 we briefly review DP-M. In section 4 DP-CS and DP-M are compared, and in section 5 we discuss the applicability of DP-CS and DP-M in the general situation of a computation with reusable subcomputations. Our conclusions are drawn in section 5

2. DYNAMIC PROGRAMMING IN COMPUTER SCIENCE

In DP-CS, results of method calls are stored in a table; if the method is called more than once with the same argument(s), only the first call will be evaluated, while the other calls retrieve their value from the table. Suppose we have a method like

```
type f (argtype x)
{  type result == initial value;
   calculate result;
   return result;
}
```

Applying DP-CS, this method is transformed into

```
type f (argtype x)
{  if (table contains result for x)
      return retrieved value;
   type result == initial value;
   calculate result;
   store result for x in table;
   return result;
}
```

A typical situation where this transformation is useful is where the method contains multiple recursive calls to itself (as is the case in the Fibonacci example). The total number of recursive calls then is exponential in the recursion depth. The transformed method will be called recursively at most once for each value of the input. In the example of Fibonacci numbers, the computational complexity of the method is turned from non-polynomial to linear.

3. DYNAMIC PROGRAMMING IN MATHEMATICS

In mathematics, Dynamic Programming is commonly used in optimization problems. Consider the following multi-stage decision problem. We are given a finite number of states, one of which is the start state, denoted by s_0. We have to take N consecutive decisions. Each decision takes us from the present state to a next state. Each state carries a numerical value; the value of state s is denoted by v(s). The problem is to find $P(s_0,N,v)$, the maximum of the values of the states which are reachable from s_0 by taking N consecutive decisions. The key insight to apply DP-M here is that

$$P(s,N,v) = P(s,M,v'), \text{ where } v'(s) = P(s,N-M,v), \quad \text{for } 0<M<N \tag{1}$$

This equation says that the optimal value of v after N decisions is equal to the optimal value after M decision of the optimal value of v after the remaining N-M decisions.
Computation of $P(s_0,N,v)$ with DP-M proceeds as follows. Eq. (1) with M=N-1 reads

$$P(s,N,v) = P(s,N-1,v'), \text{ where } v'(s) = P(s,1,v) \tag{2}$$

First v'(s) is calculated for each state s. The remaining problem, the computation of P(s,N-1,v'), is attacked in the same way, by computing v''(s) = P(s,1,v') = P(s,2,v). The problem will be solved after N-1 iterations. In the i-th iteration, P(s,i,v) is solved for all states s, using the result of the (i-1)-th iteration.

4. COMPARISON OF BOTH APPROACHES

We will compare DP-CS and DP-M by applying DP-CS to the optimization example from the previous section. An immediate recursive definition of P is:

$$P(s,n,v) = \max \{P(f(s,d),n-1,v) \mid \text{all decisions } d\}, \text{ if } n>0 \tag{3a}$$
$$P(s,0,v) = v(s) \tag{3b}$$

where f is the state transition function: in state s with decision d the next state is f(s,d). In order to avoid recomputation, a table is set up to contain the values of P(s,n,v) for all states s and all n with 1<=n<=N. Then DP-CS computes $P(s_0,N,v)$ from eq. (3), using the table as explained in section 2.
In DP-M, the values P(s,i,v) are computed in the i-th iteration and passed to the next iteration. Therefore these values have to be stored. Here we see the main similarities between DP-M and DB-CS:

1. in both approaches values P(s,n,v) have to be stored.

2. in both approaches values P(s,n,v) are computed as the maximum of a set of values as in eq. (3a).

There are differences, however. We list a number of them:
1. Where in DP-CS the values of P(s,n,v) have to be stored for all n with 0<=n<=N, in DP-M the table only needs space for n = i and n = i-1, where i is the current iteration, since the results of iteration (i-1) are only used in iteration i. By reusing memory, DB-M needs less memory than DB-CS.

2. DP-M uses eager computation: all values P(s,n,v) are calculated. DP-CS uses lazy computation: only the values P(s,n,v) are calculated which are actually needed to compute the desired result $P(s_0,N,v)$. So DP-CS can be more time-efficient than DP-M.

3.DP-M uses iteration, whereas DP-CS uses recursion. In an imperative or object-oriented programming language, iteration is more efficient than recursion. In a logic or functional programming language, recursion is the natural programming style.

4. In DP-M, needed values are always present in the table. In DP-CS, needed values may not be present in the table. Each time a table entry is needed, the table is consulted to see whether the entry is present. If it is not, it is computed, stored, and subsequently used.

5. GENERALISATION

As we have seen in the previous sections, the main virtue of Dynamic Programming is that it avoids subcomputations to be executed more than once. In this section we will explore the general situation of a computation with reusable subcomputations. We model a computation and its subcomputations as a rooted acyclic graph. The root of the graph corresponds to the main computation; each other node of the graph corresponds to a subcomputation. The descendants of a node correspond to subcomputations of its computation. We assume it is known at the outset which subcomputations a computation possibly has; however, it is not known in general which subcomputations are needed No computation can be completed before all its needed subcomputations are completed. In case there is no sharing of subcomputations, the graph is a tree, and Dynamic Programming is not appropriate. In case there is sharing of subcomputations, there are nodes with multiple ancestors, and Dynamic Programming can be used to avoid multiple executions of subcomputations.

In DP-CS, we may proceed in the usual way: we set up a table for the results of each subcomputation, and when a result of a subcomputation is needed, it is obtained from the table if it is present, otherwise the subcomputation is carried out and its result is stored.

In DP-M, there is a subtle difference with the approach we described earlier. In the general case, there need not be a subdivision into stages of the nodes of the graph such that each step of the backward iteration computes the results for a stage from the results of the previous stage, calculated in the previous step. Instead, the subcomputations have to be carried out in an order in which each subcomputation starts after its subcomputations all have finished. This means that all subcomputations should be stored in a single table, like in DP-CS, and the advantage of lower memory usage of DP-M (difference 1 in the previous section) is lost. On the other hand, in DP-M, subcomputations which are not needed are all executed (difference 2 in the previous section).
We will illustrate this with a well known example. The problem is to compute, given two strings s1 and s2, their longest common subsequence, LCS(s1,s2). It is easy to formulate a recursive algorithm which solves this problem:

LCS(s1, s2) = Ø, if s1= Ø or s2= Ø (4a)
LCS(a:s1,a:s2) = a:LCS(s1,s2) (4b)
LCS(a:s1,s2) = LCS(s1,s2), if a does not occur in s2 (4c)
LCS(a:s1,s2) = longest(LCS(s1,s2), a:LCS(s1,suffix a s2)), if a occurs in s2 (4d)

Here the string a:s is the character a followed by string s, suffix(a,s) is the suffix of s which starts after the first occurrence of a in s, and longest returns the longest of its two arguments.

This algorithm recomputes the LCS of suffices of the input strings more than once, resulting in a worst case complexity which is exponential in the length of the input strings. Using Dynamic Programming the worst case complexity is reduced from exponential to quadratic.
Suppose s1 = "abcd" and s2 = "aefb".
We identify as subcomputations the computation of the LCS of each pair of non-empty suffices of the input strings, and we set up a table to store the resultsof these subcomputations. When the table is filled completely, it is given by

	"aebf"	" ebf"	"bf"	"f"
"abcd"	**"ab"**	"b"	"b"	Ø
"bcd"	"b"	**"b"**	"b"	Ø
"cd"	Ø	**Ø**	Ø	**Ø**
"d"	Ø	**Ø**	Ø	**Ø**

which shows that the result is "ab".

In DP-M, the rows of the table are calculated from below, and each row from right to left. With this order, each time an entity is calculated, the LCS's for all non-empty suffices of the arguments of the entry are already present in the table. Note that it so happens that to calculate a row, only the present and the previous row are needed, so only storage for two rows is needed.
The DP-CS variant of the algorithm proceeds as follows:
The computation of LCS("abcd","aebf") needs the computation of LCS("bcd","ebf"), using eq. (4b). The computation of LCS("bcd","ebf") needs the computation of LCS("cd","ebf") and LCS("cd","f"), using eq. (4d). The computation of LCS("cd","ebf") needs the computation of LCS("d","ebf") and the computation of LCS("cd","f") needs the computation of LCS("d","f"), using eq. (4c).

We find that only the bold entries in the table will be calculated, the remaining positions in the table will remain empty.

6. CONCLUSION

In order to avoid the repeated execution of subcalculations, both DP-M and DP-CS can be used. In both approaches the results of subcalculations are stored. Due to the lazy evaluation strategy of DP-CS, no subcalculations will be executed which are not needed for the final result, whereas DP-M runs the risk of performing unneeded subcalculations. Due to the chosen evaluation order, DP-M has greater possibilities for optimization of memory requirements.

7. REFERENCES

[1] Bellman, R.E., (1957) Dynamic Programming, Princeton University Press

[2] Bellman, R.E. and Dreyfus, S.E., (1962) Applied Dynamic Programming, Princeton University Press

[3] Baase, S. and Van Gelder, A., (2000) Computer Algorithms, Introduction to Design and Analysis, 3th edition, Addison-Wesley