Formal Derivation of Dynamic Programming Algorithms

Leila Mofarah-Fathi and Theodore S. Norvell
Electrical and Computer Engineering
Faculty of Engineering and Applied Science
Memorial University
{*mofarah, theo*}*@engr.mun.ca*

***Abstract*—Dynamic programming is a recursive approach to solving optimization problems. It works by finding solutions to subproblems and combining those solutions. By storing solutions to solved problems, dynamic programming can be particularly efficient.**

In this paper we will discuss an approach to solving optimization problems based on specialization of an abstract dynamic programming algorithm. This provides not only reuse of the algorithm, but also reuse of its proof.

Application of dynamic programming includes Matrix chain multiplication and Largest black square.

***Index Terms*—formal methods, dynamic programming, divide and conquer, predicative style**

I. Introduction

ALGORITHM design approaches, such as greedy algorithms, dynamic programming, divide and conquer, and binary search, are generally taught and understood as informal ideas. Can we capture each algorithmic approach formally?

We are investigating how abstract specifications can be proved to be implemented by abstract algorithms. By applying a transformation that maps the abstract specification onto a concrete specification, we can derive a concrete algorithm from the abstract algorithm. This allows the abstract algorithm to be reused, along with its proof, to implement multiple concrete problems. The approach is summarized as follows. Suppose we know that an abstract specification P is implemented by an abstract algorithm Q, then if we need an algorithm for a problem $R = T(P)$, where T is a data transform, we can implement R with $T(Q)$.

Dynamic programming is a recursive approach to solving optimization and other problems [1], [3]. Like the divide-and-conquer method, it works by finding solutions to subinstances and combining those solutions. Unlike divide-and-conquer, dynamic programming saves the solutions to subinstances. There are two approaches to implementing dynamic programming: top-down approach, and bottom-up.

In this paper we will discuss an approach to solving problems based on concretization of top-down and bottom up abstract dynamic-programming algorithms. Along the way, we also formalize the closely related divide-and-conquer approach.

First let's consider two concrete problems to which we can apply our techniques.

A. The Matrix Chain Multiplication

Matrix Chain Multiplication is a problem of finding the minimum cost of calculating the product of a sequence of matrices $A_1 A_{2\ldots} A_n$[2]. Each matrix A_i has dimensions d_{i-1} by d_i .

The cost of multiplying one single matrix is zero, and the cost of multiplying two matrices $A_i A_{i+1}$is $d_{i-1} \times d_i \times d_{i+1}$. The cost of any matrix chain multiplication, consisting more than two matrices, depends on how the chain is split and how the two subchains are multiplied. Consider the following matrix chain example of four: $A_1 A_2 A_3 A_4$. A feasible solution is the parenthesization $((A_1 A_2)(A_3 A_4))$. Its corresponding cost is the sum of the following three parts:

(a) the cost of first subproduct $(A_1 A_2)$, $d_0 \times d_1 \times d_2$

(b) the cost of the second subproduct $(A_3 A_4)$, $d_2 \times d_3 \times d_4$

(c) the cost of multiplying the two matrices resulted from the subproducts $A_{1..2}$ and $A_{3..4}$, $d_0 \times d_2 \times d_4$.

Thus, the optimal cost of the product $A_i A_{i+1} ... A_j$, where $i < j$, is the minimum, over all k such that $i < k < j$, of the sum of

(a) the optimal cost of calculating $A_{i..k}$,

(b) the cost of calculating $A_{k..j}$, and

(c) $d_i \times d_k \times d_j$.

For the matrix chain problem $A_{1..n,}$the problem instance space is set of chain indices: $\mathbb{N}^n$ The problem asks for the minimum cost to do the multiplication.

B. The Largest Black Square

The problem is to find the size of the largest black square in a black and white image. We will represent the image with a constant boolean array $M \in A \times A \to \mathbb{B}$ where $A = \{0, ..N\}$[1] and $N \in \mathbb{N}$ is a constant. Each black pixel is represented by *true* and each white pixel by *false*. Let $B = A \cup \{N\}$. Pixels are indexed by A while corner points are indexed by B. The problem is to find[2]

$$\max \{(p, q) \in B \times B \cdot lsea(p, q)\}$$

where lsea stands for 'largest black square ending at' and is defined for each corner by

$$lsea(p, q) = \max \{r \in \{0, .. \min\{p, q\}\} \mid square(p, q, r)\}$$

[1] $\{0, ..N\}$ is the set of all values i such that $0 \leq i < n$

[2] $max\,\{x \in S \mid P \cdot E\}$ is the maximum over the set of all values of E where x is a value of S such that P is true

where

$$square(p,q,r) = (\forall i \in \{p-r,..p\}, j \in \{q-r,..q\} \cdot M(i,j))$$

As illustrated in Figure 1, we can find the largest square ending at a corner point (p,q) (marked as ★ in the figure), if we know the sizes of the largest squares ending at each of its three neighbors to the north, west, and north west (marked as ♦ in the figure).

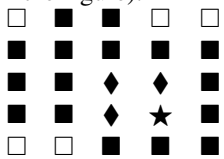

Figure 1. Black and white image

In this paper we will use the Matrix Chain Multiplication and the Largest Black Square problems as example problems to be solved by dynamic programming.

II. Divide and Conquer

A. The idea of divide and conquer

Divide and conquer is a recursive approach to solving problems [2], [3]. This method divides the problem instance to number of subinstances in order to determine a solution by combining the subsolutions. The divide and conquer method proceeds in three steps: *divide*, *conquer*, and *combine*.

Initially, the instance is divided to subinstances. Each of the subinstances are solved making a set of subsolutions. The subsolutions will be combined to create a solution to the original instance.

B. Formal divide and conquer

Consider a space of problem instances P, a space of solutions S, and a function $f : P \to S$. Formally, given a problem instance p we need to compute $f(p)$. Each specification can be interpreted as a boolean expression relating the initial state to final state [4]. The specification of a problem can be written in the following definition in SIMPLE [5], [6].

```
Definition EvaluateFunction(p) ::=
  slot P : set
  slot S : set
  slot f : P → S
  ensure s' = f(p)
```

In order for the divide and conquer strategy to be applicable, we need to define the following entities:

- PB and PL are sets such that $PB \cup PL = P$.
- $divide : PB \to 2^P$ is a function.
- $combine : PB \times 2^{(P \times S)} \to S$ is a function

We assume that leaf instances PL are easy to solve and branch problems PBwill be solved recursively. We require:

$f(p) = combine(p, A)$ provided for all $q \in divide(p)$, $(q, f(q)) \in A$

and that $divide$ induces a well-founded order on P.

The abstract divide-and-conquer algorithm can be written formally as a functional program DC that refines f.

```
Function DC(p)
  if p ∈ PL then return f(p)
  else let D = divide(p)
       let A = {q ∈ D · (q, DC(q))}
       return combine(p, A)
```

III. Top-Down dynamic programming

One of the approaches to implement dynamic programming is top-down approach [2]. The proposed formal divide and conquer definition is used as the basis for a top-down dynamic programming algorithm. We regard the top-down dynamic programming approach to be simply divide-and-conquer combined with *memoization*, that is, the storing of solutions to instances.

To apply memoization to the existing divide and conquer definition, a variable A is used as a table to store calculated results. It stores a set of pairs (p, s) that satisfy $s = f(p)$. We write $A(p)$ to mean that solution that is paired with p in A. As an invariant A represents a partial function. We can get a top-down algorithm using the refinement of function $F(p)$.

```
Definition DynamicTD(p) ::=
  inv ∀(q,t) ∈ A · t = f(q)
  var A : P → S := ∅
  proc Solve(p : P)
    if ∃s. (p,s) ∈ A then A(p)
    else if p ∈ PL then(
      let s := f(p)
      A := A ∪ {(p,s)})
    else (
      let D := divide(p)
      for q ∈ D · Solve(q)
      let s = combine(p, A))
  solve(p)
  s := A(p)
}
```

That this algorithm refines the dynamic programming problem is expressed in SIMPLE as a theorem [5], [6].

```
Theorem DynamicProgramming(p) ⊑ Solve(p)
        where DynamicProgramming(p) =
              (∀(q,t) ∈ A · t = f(q)) ⇒
              (∀(q,t) ∈ A' · t = f(q)) ∧
                  (∃s · (p,s) ∈ A')
```

[[p or q?]]
We have then that

```
Theorem s' := f(p) ⊑ DynamicTD(p)
```

IV. BOTTOM-UP DYNAMIC PROGRAMMING

The other approach to implement dynamic programming is bottom-up approach [2]. The same proposed formal divide and conquer definition is used to derive bottom-up program to dynamic programming. However, memoization applies to the bottom-up approach is slightly different. In this method, there could be some subinstances that are solved but never used, but this does not happen in top-down approach. All possible subinstances are solved, stored, and combined to build a solution to the main problem. The bottom-up approach avoids the memory and time overhead of recursive calls. There is no need to use a divide function because the structure of bottom-up approach already knows subinstances and makes a new level subinstance in each step. Thus, to get a bottom-up algorithm, we need, for each problem p a sequence of problems $p(i)$ so that $p = p(i)$ for some n and so that each i, either $p(i)$ is a leaf or all problems in $divide(p(i))$ are in $\{p(0),\ p(1), \ldots,\ p(i-1)\}$.

We can get a bottom-up algorithm using the refinement of function $F(p)$.

```
Definition DynamicBU(p) ::=
  inv ∀(q,t) ∈ A · t = f(q)
  var A : P → S := ∅
  proc Solve(p : P)
    let n | p = p(n)
    for i : 0 to n (
      if p(i) ∈ PL then(
        let s := f(p)
        A := A ∪ {(p(i), s)})
      else (
        let s = combine (p (i) , A)
        A := A ∪ {(p(i), s)})
  s := A(p)
}
```

That this algorithm refines the dynamic programming problem is expressed in SIMPLE as a theorem.

```
Theorem DynamicProgramming(p) ⊑ Solve(p)
        where DynamicProgramming(p) =
                (∀(q,t) ∈ A · t = f(q)) ⇒
                (∀(q,t) ∈ A' · t = f(q)) ∧
                    (∃s · (p,s) ∈ A')
```

And thus

```
Theorem s' := f(p) ⊑ DynamicBU(p)
```

V. APPLICATION

A. Matrix chain Multiplication

To understand the Matrix Chain Multiplication problem as an instance of the general EvaluateFunction specification, we need to fill in the three slots of the specification.

- Define P to be the set of all finite sequences of natural numbers with length at least 2, $d_{0..n} ::= (d_0, d_1, \ldots, d_n)$.
- Define S to be the set of natural numbers, $n \in \mathbb{N}$.
- Define f to be the function that maps the sequence of natural numbers to a natural number that is the minimum cost of the, product of the corresponding matrix, sequence: We define f recursively as

$$\begin{aligned} f(d_{0..1}) &= 0 \\ f(d_{0..n}) &= \min_{k\in\{1,..n\}} f(d_{0..k}) + f(d_{k..n}) + d_0 \times d_k \times d_n, \\ &\quad \text{if } n > 1 \end{aligned}$$

Filling the three slots S, P, and f, with these definitions adapts the problem.

To adapt the top-down dynamic we need to determine PL, PB, $divide$, and $combine$ slots.

- Define PL to be the set of all sequences of natural numbers with length 2.

$$PL = \mathbb{N}^2$$

- Define PB to be the set of all finite sequences of natural numbers with length greater than 2,

$$PB = \bigcup_{n\in\mathbb{N}|n>2} \mathbb{N}^n$$

- Define $divide$ to be the function that generates all the subsequences of the sequence $(d_0, d_2, \ldots, d_n)$ that are required in the process of producing the result,

$$\begin{aligned} divide(d_{0..n}) &= \{k \mid 0 < k < n \cdot d_{0..k}\} \\ &\quad \cup \{k \mid 0 < k < n \cdot d_{k..n}\}. \end{aligned}$$

- Define $combine$ to be the function that calculates the cost of a problem instance using the solved sun-instances stored in space A.

$$combine(d_{0..n}, A) = \min_{k\in\{1,..n\}} \begin{pmatrix} A(d_{0..k}) + A(d_{k..n}) \\ + \quad d_0 \times d_k \times d_n \end{pmatrix}$$

The space A that stores the pair of problem instances and their corresponding cost that is used by combine function.

Filling these slots adapts both top-down and bottom-up algorithms, but we just derive the top-down algorithm of this example in the following algorithm.

```
Definition MCMTD(p) ::=
  inv ∀(q,t) ∈ A · t = f(q)
  var A : P → S := ∅
  proc Solve(d_{0..n} : P)
    if ∃s · (d_{0..n}, s) ∈ A then A(d_{i..j})
    else if n = 2 then
                A := A ∪ {(d_{0..n}, 0)}
          else (
                for k : 1 to n − 1(
                  Solve(d_{0..k});
                  Solve(d_{k..n}) );
                var s := ∞·
                for k : 1 to n − 1
                  s := s min (A(d_{0..k}) + A(d_{k..n})
                        +d_0 × d_k × d_n))
```

$$A := A \cup \{(d_{0..n}, s)\}$$
$solve(p)$
$s := A(p)$
}

The optimization problem of matrix chain multiplication returns the minimum cost. In order to get the process of doing the multiplication we can store the breakpoints k of each sequence longer than 2.

B. Largest Black Square

To understand the Largest Black square problem as an instance of the general EvaluateFunction specification, we need to fill in the three slots of the specification.

- Define P to be the set of all corner points $(p, q) \in B \times B$, where $B = \{0, .., N\}$
- Define S to be the set of numbers r such that $0 \leq r \leq N$.
- Define f to be

$$f(p, q) = lsea(p, q)$$

where

$$lsea(p, q) = max\{r \in \{0, ..min\{p, q\}\} \mid square(p, q, r)\}$$

$$square(p, q, r) = (\forall i \in \{p - r, ..p\}, j \in \{q - r, ..q\} \cdot M(i, j))$$

Filling the three slots S, P, and f, adapts the problem.

To adapt the bottom-up dynamic programming algorithm we need to determine PL, PB, $divide$, and $combine$ slots.

- Define PL to be the set of all pairs (p, q) that are located on the most top row or the most left column,

$$PL = \{(p, q) \in B \times B \mid p = 0 \vee q = 0\}.$$

- Define PB to be all other points

$$PB = \{(p, q) \in B \times B \mid p \neq 0 \wedge q \neq 0\}$$

- Define $divide$ to be the function that generates the set of three neighbors of a point (p, q) to the north, west, and north west,

$$divide(p, q) = \{(p - 1, q), (p - 1, q - 1), (p, q - 1)\}.$$

Note that these three neighbors are lexicographically prior to (p, q).

- Define $combine$ to be the function that finds the size of the largest black square using the solved neighbor points stored in space A. The idea is that if a square is black, the largest square at (p, q) can not be larger than 1 plus the largest square ending at any of the neighbors generated by *divide*. On the other hand, there is a square ending at (p, q) that is of size 1 plus the minimum of the squares ending at the three neighbors.

$$\begin{aligned} & combine((p, q), A) \\ = \; & \text{if } \neg M(p - 1, q - 1) \text{ then } 0 \\ & \text{else } 1 + min\{A(p - 1, q), A(p, q - 1), A(p - 1, q - 1)\} \end{aligned}$$

These slots adapt both top-down and bottom-up algorithms, but we just derive the bottom-up algorithm of this example in the following algorithm. For the bottom-up algorithm the other decision that needs to be made is the ordering of the instances so that subinstances are solved before superinstances. For this problem, instances can be ordered lexicographically.

```
Definition LEASBU ::=
  inv ∀(q, t) ∈ A · t = f(q)
  var A : P → S := ∅
  proc Solve
    for i : 0 to N
      for j : 0 to N (
        if i = 0 ∨ j = 0 then
          A := A ∪ {((i, j), 0)}
        else (
          var s·
          if M(i − 1, j − 1)
            s := 0
          else
            s := 1 + min{A(i − 1, j), A(i, j − 1),
                         A(i − 1, j − 1)};
          A := A ∪ {((i, j), s)})
  Solve
}
```

This algorithm serves to calculate the $lsea$ function for each intersection point and store the result in the A table. To find the largest square is now just a matter of looking for the largest value in the table.

VI. Conclusion

Abstract algorithm of dynamic programming has been formally developed using abstract specification. This specification includes the problem space, solution space, and a function mapping them. This abstract algorithm is both presented in top-down and bottom-up approaches. Application of dynamic programming such as Matrix chain multiplication and Largest black square represents how this abstract algorithm can be implemented in concrete algorithms.

References

[1] A. Lew and H. Mauch, *Dynamic Programming. A computational tool,* Springer, 2007

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms,* 2nd Edition, McGraw-Hill, 2001

[3] J. Edmonds, "How to think about Algorithms. Loop invariants and recursion," version 0.12, Jan. 2007; http://www.cse.yorku.ca/~jeff/notes/3101/TheNotes.pdf

[4] E. C.R. Hehner, *A Practical Theory of Programming,* Springer-Verlag, 1993

[5] T. S. Norvell and Z. Ding, "An environment for proving and programming," in *Newfoundland Electrical and Computer Engineering Conference,* October 1999

[6] T. S. Norvell, "Faster search by elimination," in *Newfoundland Electrical and Computer Engineering Conference,* Nov. 2005