Lecture Notes on Dynamic Programming

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 23
November 16, 2010

1 Introduction

In this lecture we introduce *dynamic programming*, which is a high-level computational thinking concept rather than a concrete algorithm. Perhaps a more descriptive title for the lecture would be *sharing*, because dynamic programming is about sharing computation. We have already seen earlier that sharing of space is also crucial: binary decision diagrams in which subtrees are shared are (in practice) much more efficient than binary decision trees in which there is no sharing.

In order to apply dynamic programming, we generally look for the following conditions:

1. The optimal solutions to a problem is composed of optimal solutions to subproblems, and
2. if there are several optimal solutions, we don't care which one we get.

The emphasis on optimality in these conditions dates back to the 1930's when dynamic programming was developed. The name also refers to programming in the sense of the operations research literature (like, for example, *integer programming*) and does not refer to programming the way we understand today.

Under the above conditions, the idea of dynamic programming is to build an exhaustive table with optimal solutions to subproblems. Then we combine these solutions according to properties of the specific problem we are addressing. The use of a table avoids recomputation—it *shares* computation by storing results and avoiding their recomputation.

We start with a simple example and the discuss the implementation of BDDs, which uses dynamic programming in several places.

2 Fibonacci Numbers

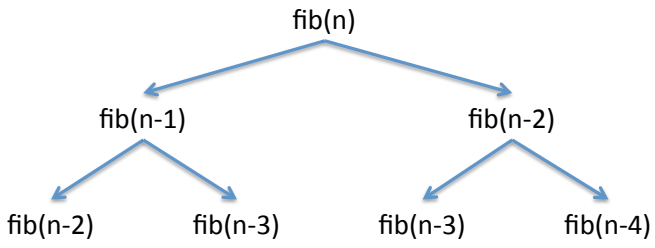
As a very simple example, we consider the computation of the Fibonacci numbers. They are defined by specifying, mathematically,

$$\begin{array}{lcll} f_0 & = & 0 & \\ f_1 & = & 1 & \\ f_{n+2} & = & f_{n+1} + f_n & (n \geq 0) \end{array}$$

A direct (and very inefficient) implementation is a recursive function

```
int fib0(int n) {
  REQUIRES(n >= 0);
  if (n == 0) return 0;
  else if (n == 1) return 1;
  else return fib0(n-1) + fib0(n-2);
}
```

When we draw the top part of a tree of the recursive calls that have to be made, we notice that values are computed multiple times.



Before we move on to improve the efficience of this program, did you notice that the program above is buggy in C, and the corresponding version would be questionable in C0? Think about it before reading on.

The problem is that addition can overflow. In C, the result is undefined, and arbitrary behavior by the code would be acceptable. In C0 the result would be defined, but it would be the result of computing modulo 2^{32} which could be clearly the wrong answer. We can fix this by explicitly checking for overflow.

```
#include <limits.h>

int safe_plus(int x, int y) {
  if (   (x > 0 && y > INT_MAX-x)
      || (x < 0 && y < INT_MIN-x) ) {
    fprintf(stderr, "integer overflow\n");
    abort();
  } else {
    return x+y;
  }
}

int fib1(int n) {
  REQUIRES(n >= 0);
  if (n == 0) return 0;
  else if (n == 1) return 1;
  else return safe_plus(fib1(n-1),fib1(n-2));
}
```

Now the program will abort in a well-defined manner upon overflow, instead of exhibiting undefined behavior which might silently give us the wrong result.

3 Top-Down Dynamic Programming

In *top-down dynamic programming* we store the values as we compute them recursively. Then, if we need to compute a value we just reuse the value if we have computed it already. A characteristic pattern for top-down dynamic programming is a top-level function that allocates an array or similar structure to save computed results, and a recursive function that maintains this array.

```
int fib2_rec(int n, int* A) {
  REQUIRES(n >= 0);
  if (n == 0) return 0;
  else if (n == 1) return 1;
  else if (A[n] > 0) return A[n];
  else {
    int result = safe_plus(fib2_rec(n-1,A), fib2_rec(n-2,A));
    A[n] = result; /* store A[n] == fib(n)  */
    return result;
  }
}

int fib2(int n) {
  REQUIRES(n >= 0);
  int* A = calloc(n+1, sizeof(int));
  if (A == NULL) { fprintf(stderr, "allocation failed\n"); abort(); }
  /* calloc initializes the array with 0s */
  int result = fib2_rec(n, A);
  free(A);
  return result;
}
```

We also call this programming technique *memoization*.

We might be tempted to improve this function slightly, by looking up the second value:

```
int fib2_rec(int n, int* A) {
  REQUIRES(n >= 0);
  if (n == 0) return 0;
  else if (n == 1) return 1;
  else if (A[n] > 0) return A[n];
  else {
    int result = safe_plus(fib2_rec(n-1,A), A[n-2]);
    A[n] = result; /* store A[n] == fib(n)  */
    return result;
  }
}
```

This would be incorrect, but why?

The problem is that C does not guarantee the order of evaluation except in some specific circumstances such as short-circuiting conjunction. So it might evaluate `A[n-2]` first, before `fib2_rec(n-1,A)`. At that point, $A[n-2]$ might still be 0 and we obtain an incorrect answer.

4 Bottom-Up Dynamic Programming

Top-down dynamic programming retains the structure of the original (inefficient) recursive function. Bottom-up dynamic programming inverts the order and starts from the bottom of the recursion, building up the table of values. In bottom-up dynamic programming, recursion is often profitably replaced by iteration.

In our example, we would like to compute $A[0], A[1], A[2], \ldots$ in this order.

```
int fib3(int n) {
  REQUIRES(n >= 0);
  int i;
  int* A = calloc(n+1, sizeof(int));
  if (A == NULL) { fprintf(stderr, "allocation failed\n"); abort(); }
  A[0] = 0; A[1] = 1;
  for (i = 2; i <= n; i++) {
    /* loop invariant: 2 <= i && i <= n+1; */
    /* loop invariant: A[i] = fib(i) for i in [0,i) */
    A[i] = safe_plus(A[i-1], A[i-2]);
  }
  ASSERT(i == n+1);
  int result = A[n];
  free(A);
  return result;
}
```

We have indicated the loop invariant here only informally, although we *could* refer to one of the earlier implementations if we wanted to, perhaps viewing `fib0` as a specification.

5 Implementing ROBDDs

In the implementation of ROBDDs, dynamic programming plays a pervasive role. Recall from Lecture 19 that ROBDDs are binary decision diagrams

(BDDs) satisfying two conditions:

Irredundancy: The low and high successors of every node are distinct.

Uniqueness: There are no two distinct nodes testing the same variable with the same successors.

These two conditions guarantee canonicity of the representation of boolean functions and also make the data structure efficient in many common cases.

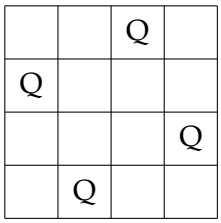
We use dynamic programming ideas in order to maintain the uniqueness invariant; irredundancy is simple in comparison. The idea is that whenever we are asked to construct a new node, given a low and high successor and a variable to test, we look in a hashtable if we have already constructed such a node. If so, we reuse it. If not, we construct it and enter it in the hashtable. The key to this hashtable consists of the variable and the (indices of) the low and high successors.

A similar idea applies when we apply a binary operation to two ROBDDs to create a new one. Here, we also check if the given operation has been applied to the same two ROBDDs before, and if so we reuse the previously stored results. If not, we compute it and store it to avoid its recomputation.

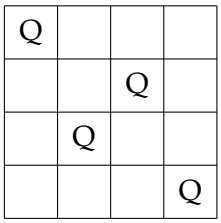
For more information on ROBDDs and the algorithms to construct and test them, we refer the reader to the code in robdd.c and the lectures notes by Henrik Reif Anderson at `http://www.itu.dk/~hra/notes-index.html`. The implementation referenced above is based quite closely and Anderson's notes.

6 Encoding the n-Queens Problem

The *n-queens problem* is to fill an $n * n$ chessboard with n queens such that none attacks any other. Queens in chess can move horizontally, vertically, and diagonally. For example, the following are examples and counterexamples of solutions on a $4 * 4$ board.



Solution



Non-solution

We would like to encode n-queens problems as ROBDDS. The idea is to assign a boolean variable to each square, where a value of 1 means that the square is occupied by a queen, and a 0 means that the square is empty. We write these variables as x_{ij} for the square at column i and row j.

x_{03}	x_{13}	x_{23}	x_{33}
x_{02}	x_{12}	x_{22}	x_{32}
x_{01}	x_{11}	x_{21}	x_{31}
x_{00}	x_{10}	x_{20}	x_{30}

Now we need to generate constraints on these boolean variables such that a correct solution will be evaluated as true (1) and an incorrect situation will be evaluated as false (0). For example, to encode that the *column* 0 *has at least one queen* on a $4 * 4$ board, we would write

$$x_{00} \vee x_{01} \vee x_{02} \vee x_{03}$$

Similarly, to encode that the main diagonal has no more than one queen we might write

$$\begin{array}{rl} & (x_{00} \supset (\neg x_{11} \wedge \neg x_{22} \wedge \neg x_{33})) \\ \wedge & (x_{11} \supset (\neg x_{00} \wedge \neg x_{22} \wedge \neg x_{33})) \\ \wedge & (x_{22} \supset (\neg x_{00} \wedge \neg x_{11} \wedge \neg x_{33})) \\ \wedge & (x_{33} \supset (\neg x_{00} \wedge \neg x_{11} \wedge \neg x_{22})) \end{array}$$

where $b \supset c$ (*b implies c*) is the same as $(\neg b) \vee c$ in boolean logic.

To see how this is programmed, we need to see the interface to the ROBDD package.

```
typedef struct bdd* bdd;
typedef int bdd_node;
bdd bdd_new(int k); /* k variables */
void bdd_free(bdd B);
int bdd_size(bdd B); /* total number of nodes */
bdd_node make(bdd B, int var, bdd_node low, bdd_node high);
bdd_node apply(bdd B, int (*func)(int b1, int b2), bdd_node u1, bdd_node u2);
int satcount(bdd B, bdd_node u);
void onesat(bdd B, bdd_node u);
void allsat(bdd B, bdd_node u);
```

The crucial functions here are `make` and `apply`.

`make(B, x, u, v)` takes a BDD B and a variable x and returns a node testing the variable x with low successor u and high successor v. Both u and v must be defined in B, and the result will be a node w also defined in B. We only use this to create variables and their negations, exploiting that the BDD nodes representing *false* and *true* are 0 and 1, respectively. The variable x_{ij} gets index $i + j * n + 1$, where 1 is added because the BDD library counts variables starting at 1. So we can obtain a BDD node representing just the BDD variable x_{33} on a $4 * 4$ board with

```
bdd B = bdd_new(4*4);
x33 = make(B, 3+3*4+1, 0, 1);
```

where 0 means that the low successor of `x33` will be 0 (false), and 1 means that the high successor of `x` will be 1 (true).

`apply(B, op, u, v)` takes two BDD nodes u and v and applies boolean operation `op` to them, returning a new node represention $u\ op\ v$. In the implementation this will be a function pointer, where the function implements the boolean operation on integers. It will be passed only 0 and 1 and must return either 0 or 1.

For example, the boolean expression

$$r = x_{00} \vee x_{01} \vee x_{02} \vee x_{03}$$

could be represented as

```
bdd B = bdd_new(4*4);
int x00 = make(B, 1, 0, 1);
int x01 = make(B, 2, 0, 1);
int x02 = make(B, 3, 0, 1);
int x03 = make(B, 4, 0, 1);
int r = apply(B, &or, x00, x01);
r = apply(B, &or, r, x02);
r = apply(B, &or, r, x03);
```

where we have previously defined

```
int or(int b1, int b2) {
  return b1 | b2;
}
```

Now it is pretty straightforward to encode, in general, that each column has a queen. We assume B holds a BDD of $n * n$ variables.

```
r = 1;
/* each column has a queen */
for (i = 0; i < n; i++) {
  u = 0; /* false */
  for (j = 0; j < n; j++) {
    x = make(B, i+j*n+1, 0, 1); /* x_ij */
    u = apply(B, &or, u, x);
  }
  r = apply(B, &and, r, u);
}
```

The outer loop (i) goes through each column building up the result BDD for r, while the inner loop (j) goes through each row in the column i and builds up u. Schematically we have

$$\begin{aligned} r &= 1 \wedge u_0 \wedge \cdots \wedge u_{n-1} \\ u_i &= 0 \vee x_{i0} \vee \cdots \vee x_{i(n-1)} \end{aligned}$$

This should be enough of a roadmap to be able to read the code in nqueens.c. We can count the number of solutions, using the `satcount` function, and obtain an enumeration of all solutions with `allsat`. For larger boards it may be infeasible to print all solutions, even if it is still feasible to calculate their number.