Dynamic programming

Victor Eijkhout

August 2004

Dynamic programming is an optimization technique, that is, a way of solving problems where a yield function is to be maximized, or a cost function minimized, under certain constraints. Certain optimization problems can be solved using calculus – unconstrained optimization being a prime example – and others by such linear algebra techniques as the simplex method. However, such continuous methods have a hard time dealing with integer constraints. For instance, in computing the yield of a car factory, the number of cars has to be integer.

The type of problems that dynamic programming is suited for is the type where the problem can be formulated as a series of decisions. For instance, in work allocation problems, there is a separate decision for the amount of work to be allocated in each month.

We will start with some examples that show the essential features of the dynamic programming approach.

1 Some examples

1.1 Decision timing

Our first example concerns the problem of when to make a one-time decision, giving a number of opportunities to do so. This example illustrates the concept of a series of decisions, and of starting at the final stage and working backward from that.

The surprise menu in a new restaurant works as follows. You will be shown 5 dishes in sequence, and you can pick one that you like, but if you turn one down, you can not reconsider. Let us say that each dish scores between 0 and 1 on your attractiveness scale. How do you maximize your choice of dish?

Call the scores you give the dishes x_i, and N the number of dishes.

- If you wait till the last dish, you have no choice.
- The last dish can be anything, so the best you can say is that it will have an expected attractiveness of 0.5. Therefore, if $x_{N-1} > 0.5$, you take that one, since it is better than what you can expect in the next step.
- Now, you will take dish $N-1$ in half the cases, giving you on average a $.75$ score, and the other half of the cases you take dish N, with a score of $.5$. Therefore, you are expecting to score $.625$, and you will take dish $N-2$ if it scores more than that.

- In .375 of the cases, dish $N-3$ will score higher than that.
- Et cetera.

From this example we see some characteristics:

Stages The optimization problem involves a sequence of stages, each involving a choice.

Principle of optimality Once you arrive at a certain stage, the optimal solution for the rest of the path is independent of how you got to that stage.

Stepwise solution The solution (here: solution strategy) is arrived at by starting at the final and working backward. We will also examples that are solved forward; in general it is a characteristic that dynamic programming problems are solved stage-by-stage.

Often, the fact that the problem is solved starting at the last stage is considered an essential feature. We will not do so: many dynamic programming problems can also be solved forward.

> **Exercise 1.** For this example, draw up the recurrence relation between the expected scores. Prove that the relation is monotonically increasing (for decreasing index), and bounded above by 1, in fact with limit 1. Bonus: solve this relation explicitly.

1.2 A manufacturing problem

Suppose a factory has N months time to produce a quantity S of their product, which we will for now assume to be bulk. Because of seasonal variations, in month k the cost of producing an amount p_k is $w_k p_k^2$. The problem is to produce the requested amount in the given time, at minimal cost, that is

$$\min_{\sum p_k = S} \sum w_k p_k^2.$$

We break down the problem by looking at the cost for producing the remaining amount in the remaining time. Define the minimum cost as

$$v(s|n) = \min_{\sum_{k>N-n} p_k = s} \sum w_k p_k^2$$

and $p(s|n)$ as the work that has to be done n months from the end, given that s work is left, then, by splitting this into the cost this month plus the cost for the then remaining time, we get

$$\begin{aligned} v(s|n) &= \min_{p_n \le s} \left\{ w_n p_n^2 + \sum_{\substack{k>N-n+1 \\ \sum p_k = s-p_n}} w_k p_k^2 \right\} \\ &= \min_{p_n \le s} \left\{ w_n p_n^2 + v(s-p_n|n-1) \right\} \end{aligned}$$

That is, we get a recurrence where the remaining work for n months is expressed in that for $n-1$ months.

Starting off is easy: $p(s|1) = s$, and $v(s|1) = w_1 s^2$. By the above rule then

$$v(s|2) = \min_{p_2} \{ w_2 p_2^2 + v(s-p_2|1) \} = \min_{p_2} c(s, p_2)$$

where $c(s, p_2) = w_2 p_2^2 + w_1 (s-p_2)^2$. We find the minimum by taking $\delta c(s,p_2)/\delta p_2 = 0$, which gives us $p(s|2) = w_1 s/(w_1+w_2)$ and $v(s|2) = w_1 w_2 s^2/(w_1+w_2)$.

Solving one or two more steps like this, we notice the general form:

$$p(s|n) = \frac{1/w_n}{\sum_{i=1}^{n} 1/w_i} s, \qquad v(s|n) = s^2 \sum_{i=1}^{n} 1/w_i.$$

This solution can in fact be derived by a variational approach to the constraint minimization problem

$$\sum_k w_k p_k^2 + \lambda(\sum_k p_k - S)$$

for which we set the derivatives to both p_n and λ to zero.

This problem shows another characteristic of dynamic programming:

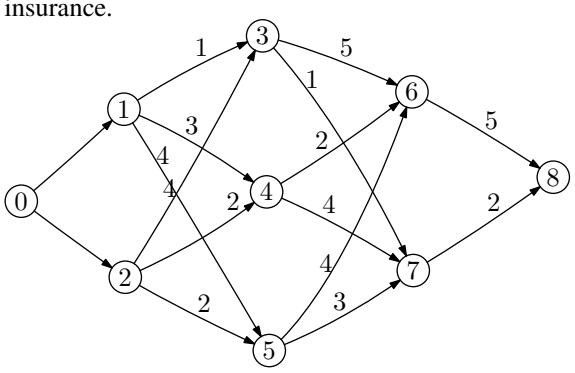
State The cost (yield) function that we define as a function of the stage, also has a state parameter. It typically describes the amount of some constrained quantity that is still left. In this case, that is the amount still to be produced.

We see that for this problem there is both an analytical solution, and one found by dynamic programming, using analytical techniques for local optimization. However, these techniques would become increasingly cumbersome, if we imposed restrictions such as that the factory does not have unlimited production capacity, or even impossible, if the product can only be made in discrete units, meaning that the p_n have to be integers.

The next problem is a good example of the sort of discrete choices that dynamic programming is well suited for.

1.3 The stagecoach problem

A business man in the Old West needs to travel from city 0 to city 8. For this, he has to go through 3 other cities, but in each stage of the trip there are choices. This being the Wild West, he decides to get travel insurance. However, not all routes are equally safe, so the cost of insurance varies. The problem is to find the trip that minimizes the total cost of the insurance.



We will look at various ways of solving this problem. First let us define the data.

```
table = [ [0, 5, 4, 0, 0, 0, 0, 0, 0], # first stage: 0
          [0, 0, 0, 1, 3, 4, 0, 0, 0], # second: 1 & #2
          [0, 0, 0, 4, 2, 2, 0, 0, 0],
```

```
          [0, 0, 0, 0, 0, 0, 5, 1, 0], # third: 3, #4, #5
          [0, 0, 0, 0, 0, 0, 2, 4, 0],
          [0, 0, 0, 0, 0, 0, 4, 3, 0],
          [0, 0, 0, 0, 0, 0, 0, 0, 5], # fourth: 6 & #7
          [0, 0, 0, 0, 0, 0, 0, 0, 2]
          ]
final = len(table);
```

1.3.1 The wrong way to solve this

The solution to the stage coach problem is easy to formulate recursively, given that you are in some city:

- If you are in the final city, the cost is zero;
- Otherwise it is the minimum – over all cities reachable from here – of the cost of the next leg plus the minimum cost from that city.

```
# the wrong way
def cost_from(n):
    # if you're at the end, it's free
    if n==final: return 0
    # otherwise range over cities you can reach
    # and keep minimum value
    val = 0
    for m in range(n+1,final+1):
        # for all later cities
        local_cost = table[n][m]
        if local_cost==0: continue
        # if there is a connection from here,
        # compute the minimum cost
        local_cost += cost_from(m)
        if val==0 or local_cost<val:
            val = local_cost
    return val
print "recursive minimum cost is",cost_from(0)
```

If there are N cities, divided into S stages of L cities in each stage, and assuming that in each stage all cities of the next stage are reachable, the cost of this algorithm is $O(L^S)$.

1.3.2 Dynamic programming solution

The implementation strategy of the previous section is wasteful. Consider some city 1, and cities 2 and 3 in the stage before that. The cost computations from 2 and 3 both compute the cost from 1, which is clearly redundant. We call this characteristic

Overlapping subproblems: A straightforward (recursive) solution to the problem would revisit a subproblem, meaning that different solution attempts have a common subproblem.

Recognizing this leads us to a better solution strategy: once the minimum cost from one city has been computed, we store it for future reference.

An equivalent interpretation of this strategy is that we compute in each stage a cost function $f_n(x_n)$, describing the cost of reaching the end starting at the nth step, giving that we start there in city x_n. This is in fact a dynamic programming solution, with x_n the state variable in the nth stage.

Formally, $f_k(s)$ is the minimum cost for traveling the from stage k given that your are in city s in that stage. Then

$$f_{k-1}(s) = \min_t\{c_{st} + f_k(t)$$

where c_{st} is the cost of traveling from city s to t.

Initially, the cost from every city till the final one (in particular from the final one itself) is zero:

```
# initialization
cost = (final+1)*[0]
```

Now we loop backwards over the stages, computing in each stage the cost of all city we can go through. These two loops – the stages, and the cities in each stage – can actually be collapsed into one loop:

```
# compute cost backwards
for t in range(final-1,-1,-1):
    # computing cost from t
```

For each city `t` we consider the ones `i` that are reachable from it. By induction, for these later ones we already know the cost of going from them to the final one, so the cost from `t` to the final one is the cost from `t` to `i` plus the cost from `i`:

```
    for i in range(final+1):
        local_cost = table[t][i]
        if local_cost==0: continue
        local_cost += cost[i]
```

If there was no cost yet associated with `t`, we set it now, otherwise we compare the cost from `t` over `i` with the cost of an earlier evaluated route:

```
        if cost[t]==0 or local_cost<cost[t]:
            cost[t] = local_cost
```

In the end, the minimum cost is given in `cost[0]`.

We see that the main difference between this solution and the recursive one given above, is that the recursive function call has been replaced by a lookup in a table.

The running time of this algorithm is $O(N \cdot L)$ or $O(L^2S)$, which is a considerable improvement over L^S for the straightforward implementation. This solution carries an extra cost of N memory locations; on the other hand, it does not build up a recursion stack.

1.3.3 Forward dynamic programming solution

This problem was solved by starting at the end point. We can also work our way to the solution forwards, with a code that is essentially the same. Instead of computing the cost

of reaching the final city from an intermediate, we now compute the cost of reaching the intermediate city from the initial one.

We loop over all cities and all their connections:

```
cost = (final+1)*[0]
for t in range(final):
    for i in range(final+1):
        local_cost = table[t][i]
        if local_cost == 0: continue
```

Now we can compute the cost to the connecting city as the transition cost, plus the known minumum cost to get where we are:

```
        cost_to_here = cost[t]
        newcost = cost_to_here+local_cost
        if cost[i]==0 or newcost<cost[i]:
            cost[i] = newcost
```

The result is now in `cost[final]`.

The minimization problem corresponding to this algorithm concerns $f_k s$, the cost to get to city s in stage k. Then

$$f_{k+1}(t) = \min_s \{c_{st} + f_k(s)$$

which is equivalent to the earlier problem.

Exercise 2. A 'sparse matrix' is a matrix where a number of matrix elements are zero, sufficiently many that you do not want to store them. To compute the matrix vector product $y = Ax$ you then do not compute the full sum $y_i = \sum_j a_{ij}x_j$, but only those terms for which $a_{ij} \neq 0$. This sort of operation is easy enough to code, but is pretty inefficient in execution.

Suppose that for small k the product with k consecutive matrix elements (that is $a_{ij}x_j + a_{ij+}x_{j+1} + \cdots + a_{ij+k-1}x_{j+k-1}$ can be executed more efficiently than doing it as k separate operations. For instance, suppose that with $k = 3$ the time per $a_{i\cdot}x_{\cdot}$ reduced to $.4$ of the normal multiply time, that is, doing three consecutive multiplications as a block takes time 1.2, while doing them separately takes time 3.

Now, if $a_{11} \neq 0$, $a_{12} = 0$, $a_{13} \neq 0$, the multiplication $a_{11}x_1 + a_{13}x_3$ takes time 2, but if we store the intermediate zero at a_{12}, the size 3 block multiplication takes time 1.2. This means that doing some superfluous operations (multiplying by zero) we can actually speed up the matrix-vector product.

Let a pattern of nonzeros and reduction factors be given. The pattern stands for the locations of the nonzeros in a matrix row, for instance

```
row = [1,0,0,1,0,1,1,0,0,1,0,0,0,1,1,0,1,1]
redux = [1, .75, .4, .28]
```

Formulate a principle of optimality for this problem, and write a dynamic programming solution for finding the covering of the sparse row that will lead to the shortest multiplication time. Tinker with the `redux` times (but make sure the n-th is more than $1/n$ in base-1 indexing) and see how the solution changes.

1.4 Traveling salesman

The above problems all had dynamic programming solutions with a cost slightly more than linear in the input problem size. Dynamic programming does not always give that low a complexity.

The traveling salesman problem looks a bit like the stagecoach problem above. However, now the traveler does not only need to go from a starting to a final city, he also has to visit every city on his travel.

This problem can be solved by dynamic programming, but the concept of stage is now more complicated. We can no longer map the cities into a linear ordered set of stages since they can be visited in any sequence.

The solution is to let stage n correspond to picking the nth city, and to define the current state as the last visited city, plus the set of the remaining ones. Initially we loop over all possible last cities, so the cost is the sum of the single leg trip to the end point, plus the minimum cost through remaining cities. unvisited cities.

To be precise: a state is a pair (S, f) of a set of cities left to be visited, and the current city $f \in S$.

We can now construct a cost function that depends on the stage and the current state.

$$\begin{aligned} C(\{1\}, 1) &= 0 \\ C(\{f\}, f) &= a_{1f} \quad \text{for } f = 2, 3, \ldots \\ C(S, f) &= \min_{m \in S-f} [C(S-f, m)] + a_{mf}] \end{aligned}$$

This is easily enough implemented:

```
def shortest_path(start,through,lev):
    if len(through)==0:
        return table[start][0]
    l = 0
    for dest in through:
        left = through[:]; left.remove(dest)
        ll = table[start][dest]+shortest_path(dest,left,lev+1)
        if l==0 or ll<l:
            l = ll
    return l
to_visit = range(1,ntowns);
s = shortest_path(0,to_visit,0)
```

This solution has factorial complexity.

2 Discussion

In the example above we saw various properties that a problem can have that make it amenable to dynamic programming.

Stages The optimization problem involves a sequence of stages, each involving a choice, or a discrete or continuous parameter to be determined.

Stepwise solution The solution is arrived at by solving the subproblems in the stages one by one. Often this is done starting at the final stage and working backward.

State The cost (yield) function that we define as a function of the stage, also has a state parameter. It typically describes the amount of some constrained quantity that is still left to be consumed or produced.

Overlapping subproblems This is the property that a straightforward (recursive) solution to the problem would revisit a subproblem, meaning that different solution attempts have a common subproblem.

Principle of optimality This is the property that the restriction of a global solution to a subset of the stages is also an optimal solution for that subproblem.

The principle of optimality is often phrased thus:

> An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must be an optimal policy with regard to the state resulting from the first decision.

It is easy to show that the principle of optimality holds for certain functions and constraints. For example, consider the problem of maximizing $\sum_i^N g_i(x_i)$ under the constraint $\sum_i x_i = X$ where $x_i \geq 0$. Call this maximal sum $f_N(X)$, then

$$\begin{aligned} f_N(X) &= \max_{\sum_i^N x_i = X} \sum_i^N g_i(x_i) \\ &= \max_{x_N < X} \left\{ g_N(x_N) + \max_{\sum_i^{N-1} x_i = X - x_N} \sum_i^{N-1} g_i(x_i) \right\} \\ &= \max_{x_N < X} \{ g_N(x_N) + f_{N-1}(X - x_N) \} \end{aligned}$$

We see here that the form of the g_i functions does not enter the argument, but the fact that the total utility is a sum of g_is does. Utility functions that are not symmetric in the component g_i functions clearly can not straightforwardly be solved with dynamic programming.

```
"Diglio A. Simoni" <diglio@simoni.org> wrote in message
news:ouFTc.27970$Kt5.1192@twister.nyroc.rr.com...
> Suppose C(m,dist[d]) is the optimal solution to travel distance d using m
> gas stations

That won't do, because:

> Two cases arise
>     a) Buy gas at station m, in which case:
>
>             cost = C(m-1,dist[m]) + price[m] * ( (dist[m] - dist[j] ) /
10 )
> + D( dist[m] - dist[j] )

You can't do that.  The problem says you have to fill your tank, and since
```

the parameters of the cost function don't capture how much gas you have left, you don't know how much filling your tank will cost.

In fact, the number of gas stations you've used at any point is irrelevant. If you could put in however much gas you liked, you wouldn't need to use dynamic programming to find a solution.

Since your prof asked for complexity analysis in terms of distance, I can tell that he expects you to use something like C(n,i) = the minimum price paid to get to dist[i] with n gallons of gas left in the tank, which works just fine.

You'll probably get extra marks if you expain how you can use a different cost function, or just a little finesse in the implementation, to get a better complexity result in terms of the number of gas stations.

Contents