

Dynamic Programming

Introduction

- The name dynamic programming was coined in 1957 by Richard Bellman to describe a type of optimal control problem.
- The term programming is used to mean a “series of choices” like programming your VCR and has nothing to do with programming a computer.
- The term dynamic conveys the idea that choices may depend on the current state, rather than being decided ahead of time.
- In dynamic programming we do all the possible subproblems early and store them somewhere.
 - Thus, when we have a need for a subproblem, we don’t run the risk of having to recompute it.

Making Change

- Consider change making: if we try to make change for 63 cents using a greedy algorithm, it takes 6 coins (2 quarters, 1 dime, 3 pennies).
- If have a 21 cent coin, the greedy method fails to find optimum solution. (3 21-cent coins.)
- If we try to do divide and conquer, we end up solving similar problems repeatedly.
- What if we try to find the least number of coins for each amount?
- We’ve seen this before – can do greedy or exhaustive. Greedy is exponential as for each coin consider both “use it” and “don’t use it.”
- The problem is known to be NP complete.
- We save the results of smaller problems in a table so we can reuse the result of the smaller problem.
- This table is shown in Figure 1.
 - Columns – amount to make change for.
 - Rows – use only that coin and smaller.
- To find the value of an element i, j , we take the minimum value in the array element $[i-1, j]$ (number if we didn’t use the current coin) and $1 + \text{element}[i, j - \text{value}[i]]$ (number if we use the current coin).
- In the following table consider $[10, 11]$ and $[7, 12]$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14
5	1	2	3	4	1	2	3	4	5	2	3	4	5	6
7	1	2	3	4	1	2	1	2	3	2	3	2	3	2
10	1	2	3	4	1	2	1	2	3	1	2	2	3	2
25	1	2	3	4	1	2	1	2	3	1	2	2	3	2

Figure 1 Reuse Table for Change

0/1 Knapsack Problem

- You have a series of items of a given weight. You want to select a number of items whose sum is less than or equal to the capacity of a knapsack.
- But we want the value maximized.
- Greedy techniques are likely not optimal – and you have to try all combinations.

```

int value[MAX]; // value of each item
int weight[MAX]; // weight of each item

//You can use item "item" or items with lower number
//The maximum weight you can have is maxWeight
int bestValue(int item, int maxWeight)
{ if (item == n)
  return (maxWeight < weight[item]) ? 0 : value[item];
  if (maxWeight < weight[item])
  // current item can't be used, skip it
  return bestValue(item-1, maxWeight);

  useIt = bestValue(item-1, maxWeight - weight[item])
    + value[item]
  dontUseIt = bestValue(item-1, maxWeight);
  return max (useIt, dontUseIt);
} // bestValue

```

Figure 2 Solution for Knap Sack Problem

- You could try them exhaustively.
- Possible code is shown in Figure 5.
- If we have a capacity of 10, consider a tree in which each level corresponds to considering each item (in order). The weights of the items are: 2, 2, 6, 5, 4 and the values are 6, 3, 5, 4, 6.
- This repetition can be seen in the call tree of Figure 3. The initial call is bestValue(1, 10).
 - Notice, in this simple example about a third of the calls are duplicates.
 - Also notice that we didn't even need to look at value in the call of trees, if we try everything.

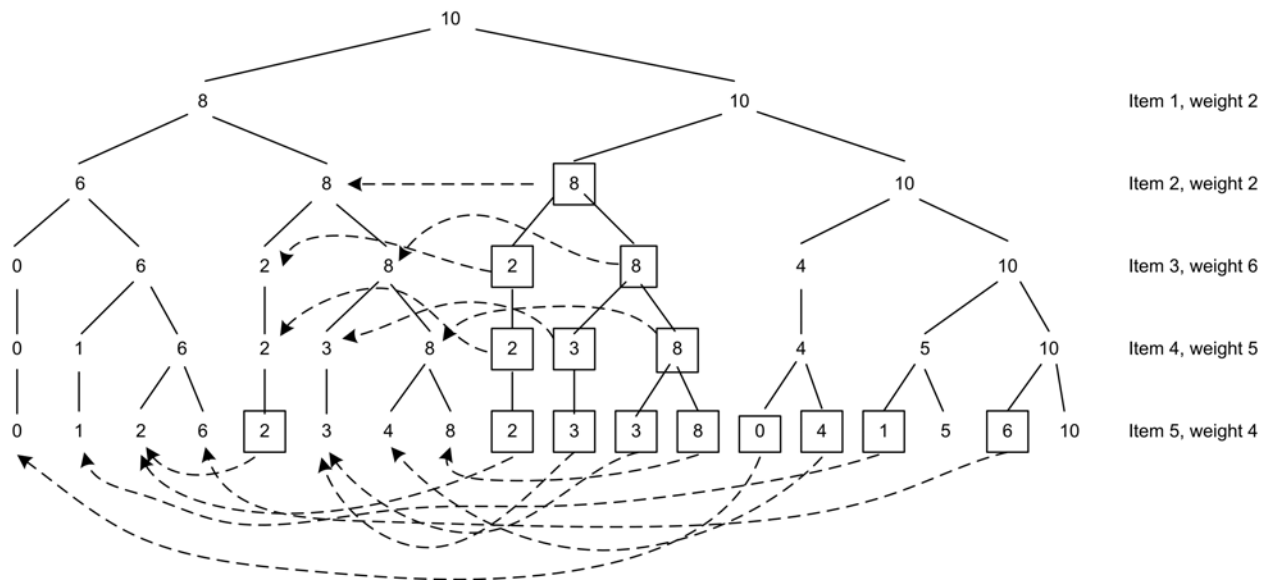


Figure 3 Knap Sack Repetition

- If we stored the best solution found for all subproblems, it would look something like: $best[item][max]$ = given the current item (or earlier in the list) and max value, which is the best value.
- If we just consider weights and **not** values, we get the table shown in Figure 4.

	1	2	3	4	5	6	7	8	9	10
2	0	2	2	2	2	2	2	2	2	2
2	0	2	2	4	4	4	4	4	4	4
6	0	2	2	4	4	6	6	8	8	10
5	0	2	2	4	5	6	7	8	9	10
4	0	2	2	4	5	6	7	8	9	10

Figure 4 Weights for 2, 2, 6, 5, 4

- We could compute such a table in an iterative fashion as shown in Figure 5.

```

int getBest( i,w )
{   if ( i <0 || i > itemMax || w<0 || w > maxWeight) return 0;
    return best[i][w];
} // if
// Setup, do first row
for ( int maxWeight = 1; maxWeight <= capacity; maxWeight++ )
    if ( weight[1] > maxWeight ) best[1][maxWeight] = 0;
    else best[1][maxWeight] = value[1];
for (int item = 1; item <= itemMax; item++)
    for (int maxWeight = 1; maxWeight <= capacity; maxWeight++)
        {   useIt = getBest(item-1, maxWeight-weight[item]) + value[item]
            dontUseIt = getBest(item-1, maxWeight)
            best[item][maxWeight] = max(useIt, dontUseIt)
        } // for

```

Figure 5 Iterative Computation of Best

- Let's compare the two strategies:
 - Normal / forgetful – wait until you are asked for a value before computing it. You may have to do some things twice, but you will never do anything you don't need.
 - Compulsive/elephant – you do everything before you are asked to do it. You may compute things you never need, but you will never compute anything twice (as you never forget).
- Which is better? At first it seems better to wait until you need something, but in large recursions, almost everything is needed somewhere and many things are computed *lots* of times.
- Consider the complexity for a max capacity of M and N different items.
 - Normal – for each item, try it two ways (useIt or dontUseIt) – $O(2^N)$
 - Compulsive – fill in array – $O(MN)$
- Which is better depends on values of M and N . Notice, the two complexities depend on different variables.

Sequence Comparisons -Edit Distance

- Problems in molecular biology involve finding the minimum number of edit steps that are required to change one string into another.
- Three types of edit steps:
 - insert
 - delete
 - replace
- Example: abbc → babb
 - abbc → bbc → bbb → babb (3 steps – delete, replace, insert)
 - abbc → babbc → babb (2 steps – insert, delete)
- We are trying to minimize the number of steps.
- Idea – look at making just one position right. Find all the ways you could use. Count how long each would take and recursively figure total cost.

- Orderly way of limiting the exponential number of combinations to think about.
- For ease in coding, we make the last character right (rather than any other). (Then I can tell the routine the beginning address and number of positions without having the string begin in a different location. In C it would be no problem, but other languages use a different technique.)
- $C(n, m)$ is the cost of changing the first n of str1 to the first m of str2. There are four possibilities (pick the cheapest):
 1. If we **delete** a_n , we need to change $A(n-1)$ to $B(m)$. The cost is $C(n, m) = 1 + C(n-1, m)$.
 2. If we **insert** a new value at the end of $A(n)$ to match b_m , we would still have to change $A(n)$ to $B(m-1)$. The cost is $C(n, m) = 1 + C(n, m-1)$.
 3. If we **replace** a_n with b_m , we still have to change $A(n-1)$ to $B(m-1)$. The cost is $C(n, m) = 1 + C(n-1, m-1)$.
 4. If we **match** a_n with b_m , we still have to change $A(n-1)$ to $B(m-1)$. The cost is $C(n, m) = C(n-1, m-1)$.
- We could define $c(i, j)$ to be either 0 (if $a_i = b_j$) or 1 (otherwise) to combine the last cases.
- Thus we have the following rules:

$$C(n, m) = \min \begin{cases} C(n-1, m) + 1 & (\text{deleting } a_n) \\ C(n, m-1) + 1 & (\text{inserting } b_m) \\ C(n-1, m-1) + c(n, m) & (\text{replacing } \vee \text{ matching } a_n) \end{cases}$$

- We have turned one problem into three problems - just slightly smaller.
- Bad situation - unless we can reuse results. Dynamic Programming.
- We store the results of $C(i, j)$ for $i = 1, n$ and $j = 1, m$.
- If we need to reconstruct how we would achieve the change, we also store $M(i, j)$ which indicates which of the four decisions lead to the best result.
- The code to do this is shown in Figure 6

```

M[i,j] = d(s1[1..i], s2[1..j])
M[0][0] = 0;
M[i][0] = i, i = 1..|s1|;
M[0][j] = j = 1..|s2|
for ( i = 1; i < |s1|; i++ )
for ( j = 1; j < |s2|; j++ )
M[i][j] = min(
M[i-1][j-1] + s1[i] == s2[j] ? 0 else 1,
M[i-1][j] + 1,
M[i][j-1] + 1)

```

Figure 6 Code for edit distance

- Complexity: $O(mn)$ - but needs $O(mn)$ space as well.
- Consider changing do to redo. This is shown in Figure .

	*	r	e	d	o
*	←I0	←I1	←I2	←I3	←I4
d	↑D1	R1	↖R2	↖M2	(I3
o	↑D2	↖R2	↖R2	↖R3	↖M2

Figure 7 M(i,j) for do to redo

- Consider changing mane to mean. This is shown in Figure 8.

	*	m	e	a	n
*	←I0	←I1	←I2	←I3	←I4
m	↑D1	↖M0	←I1	←I2	←I3
a	↑D2	↑D1	↖R1	↖M1	←I2
n	↑D3	↑D2	↖R2	↑D2	↖M1
e	↑D4	↑D3	↖M2	↑D3	↑D2

Figure 8 Change mane to mean

Applications

- File revision – Linux `diff f1 f2` that finds the different between files `f1` and `f2` producing an edit script to convert `f1` into `f2`.
- Remote screen update – If a computer program on machine 1 is being used by someone from a screen on a (distant) machine 2, e.g. via `rlogin`, then machine 1 may need to update the screen on machine 2 as the computations proceeds.
- Spelling correctors – if a text contains a word, `w`, that is not in the dictionary, a ‘close’ word, i.e., one with a small edit distance to `w`, may be suggested as a correction.
- Plagiarism detection – provide an indication of similarity that might be too close in some situations.
- Molecular biology – give an indication of how close tow string are. Similar measures are used to compute a distance between DNA sequence (strings of {A,C,G,T}).

Shortest Paths

- Try to find the shortest paths (in a digraph) from a node to all other nodes.
- There exists a fast algorithm: Dijkstra’s algorithm. It is a dynamic programming algorithm. Although it obviously generates no improper paths, it is not evident that all paths are established.
- Ours differs from that given previously in that it is an all-points problem rather than from a given vertex.
- $path_k[i,j]$: reflects paths from i to j such that only points between 1 and k are used as intermediate values.
- When done for all k , a complete transitive closure is found $O(V^3)$.
- The code for this is shown in Figure 7.

```

for k = 1 to v
  for i = 1 to v
    if path[i,k] != INF
      for j = 1 to v
        path[i,j] = min(path[i,j], path[i,k]+path[k,j])

```

Figure 7 Code for Transitive Closure

- Note this is dynamic programming in that all subproblems are solved (getting from i to j only going through k) and those solutions helped solve higher level problems.

Noncrossing Subset of Nets

- Suppose people are coming out of N doors and going into N doors as determined by a routing. The text calls this a Net. Each item i (the door he is at) is paired with a connection door, C_i . What is the maximum number of people that can be moved at the same time?
- Can we state this problem in terms of smaller instances of itself – which can be stored?
- Suppose we have a matrix $size[a][b]$ which - is maximum number of nets that can be routed together if the maximum in door is “a” and the maximum outdoor is “b”.
- Consider the example (from the text) in Figure 8.

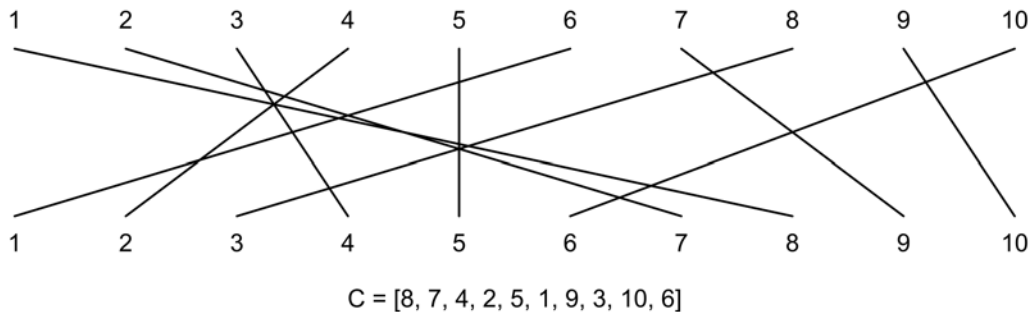


Figure 8 A Wiring Instance

- Figure 9 shows the MNS for Figure 8

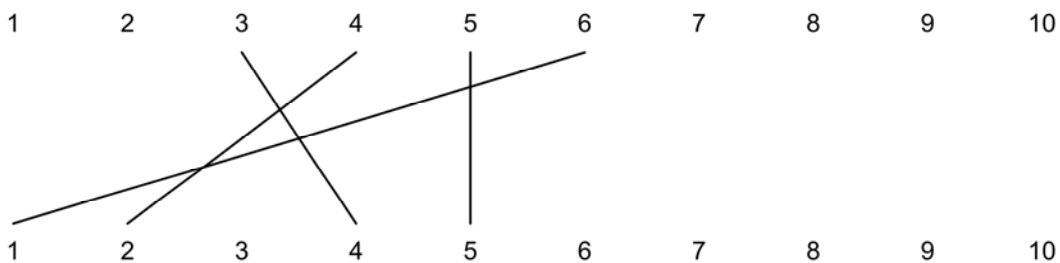


Figure 9 Possible Nets for size(7,6)

- This can be solved with dynamic programming using the following idea.

- For each new net, either use it or don't.
- If you don't use it, you have the size you had before.
- If you do use it, you need to look up the best size you can use.
- This is shown in Figure 10.

```

for (int j = 1; j < MAX; j++) // first item
  if (j < C[1]) size[1][j]=0
  else size[1][j]=1;

for (int i = 2; i < MAX; i++)
  for (int j = 1; j < MAX; j++)
    if (j < C[i]) size[i][j] = size[i-1][j] // current net cannot be a member
    else
      { includelt = size[i-1][C[i]-1] + 1; // add it to the best you can do with it
        dontIncludelt = size[i-1][j]; // best without it
        size[i][j]=max(includelt, dontIncludelt);
      }

```

Figure 10 Code for MNS

- Figure 11 shows size[i][j] for the net shown in Figure 8.

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	1	1	1
2	0	0	0	0	0	0	1	1	1	1
3	0	0	0	1	1	1	1	1	1	1
4	0	1	1	1	1	1	1	1	1	1
5	0	1	1	1	2	2	2	2	2	2
6	1	1	1	1	2	2	2	2	2	2
7	1	1	1	1	2	2	2	2	3	3
8	1	1	2	2	2	2	2	2	3	3
9	1	1	2	2	2	2	2	2	3	4
10	1	1	2	2	2	3	3	3	3	4

Figure 11 Size(i,j)