Dynamic Programming

`Fri Oct 11 10:47:28 AM PDT 2024`

Introduction

Dynamic programming is an algorithm paradigm in which an instance of the problem consists of *subinstances* (Usually called *subproblems*) The subproblems form a directed acyclic graph, which must be solved in topological order. Each subproblem reads the input of the instance together with the outputs of its predecessor subproblems, and computes its own output, which can be read by its successor subproblems. The last subproblem worked computes the overall output of the instance.

There are many problems which can be reduced to dynamic programming, and many classic algorithms are dynamic programming algorithms when suitably formulated.

Weighted Acyclic Directed Graphs

The simplest kind of dynamic program is finding the minimum (or maximum) weight path from a given start vertex s of a weighted acyclic directed graph, say G, to a given other vertex, say t, for the single pair version, or to all other vertices of G, for the single source version. Negative arc weights are allowed. Since finding a maximum is same as finding a minimum after all weights are negated, those two goals are equivalent.

Topological Order

Suppose that there are n vertices of a weighted acyclic graph G. which we write in topological order: v[0], v[1], . . . v[n−1], where v[0] is the source vertex. Let S[i] be the smallest total weight of any path (that means, directed path) from v[0] to v[i]. We assume that, for each i, there at least one path from v[0] to v[i]. Let W[i,j] be the weight of the arc from v[i] to v[j] if such an arc exists, and let Pred[j] be the set of i < j such that there is an arc from v[i] to v[j]. The following dynamic program solves the subproblems in topological order.

```
S[0] = 0
for all j from 1 to n-1
   S[j] = "infinity"      // There is no such number as infinity! We use
   for all i in Pred[j]   // that term to indicate that no value is defined.
      temp = S[i]+W[i,j]  // Predecessors of j, i.e., there's an arc j->i
      if(temp < S[j])
         S[j] = temp
         back[j] = i
```

Lines 4,5,6,7 of the code could be compressed to:

```
   S[j] = min{S[i]+W[i,j] : i in Pred[j]}
```

Coin Row Problems

For a coin row problem, we are given a row of n coins, each of which has some value. Our goal is to select a maximum weight *legal* subset of the coins, where the definition of legality is different for each version.

For each problem, let the coins be numbered $1 \ldots n$, and let $V[i]$ be the value of the i^{th} coin. For simplicity, we assume that $V[i] > 0$ for each i.

The Simple Version

Define a subset to be legal if it contains no two consecutive coins of the row. We give two dynamic programs.

Best subsequence up to some point

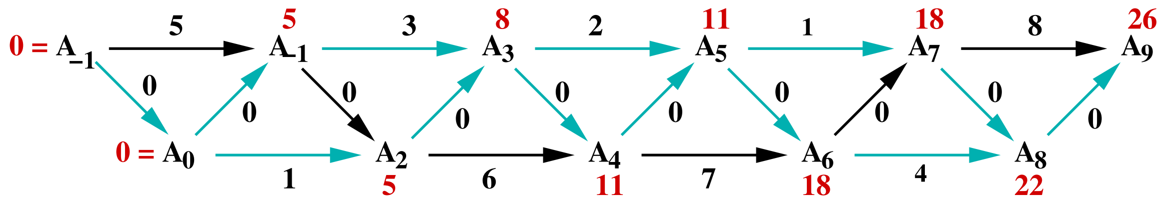
In our first program, the subproblems are to compute $A[i]$, the maximum weight of any legal subsequence of the first i coins. This subsequence need not end at i. For convenience we introduce dummy values to make our code more uniform; We compute $A[i]$ for $-1 \leq i \leq n$. as follows.

```
Program 1:

A[-1] = 0; // dummy value
A[0] = 0; // dummy value
for(int i = 1; i <= n; i++)
 A[i] = max(A[i-1],A[i-2]+V[i]);
return A[n];
```

I leave it up to you to modify the code so as to compute backpointers: back[i] must be equal to either i−1 or i−2 for each i.

The problem reduces to finding a maximum weight path in a weighted directed graph. We use an explicit example sequence of the $\{V[i]\}$: 5, 1, 3, 6, 2, 7, 1, 4, 8. The maximum total legal subsequence is 5, 6, 7, 8.



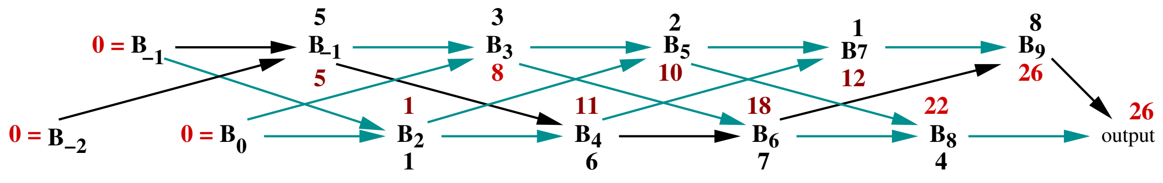
The Graph Corresponding to Program 1

Best subsequence ending at some point

Our second algorithm solves the same problem, but we compute subproblems B[i], defined to be the maximum weight of any legal subsequence ending at the i^{th} coin.

```
Program 2:
```

```
B[-2] = 0; // dummy value
B[-1] = 0; // dummy value
B[0] = 0; // dummy value
for(int i = 1; i <= n; i++)
 B[i] = V[i] + max(B[i-2],B[i-3]);
return max(B[n],B[n-1]);
```



The Graph Corresponding to Program 2

Another Version

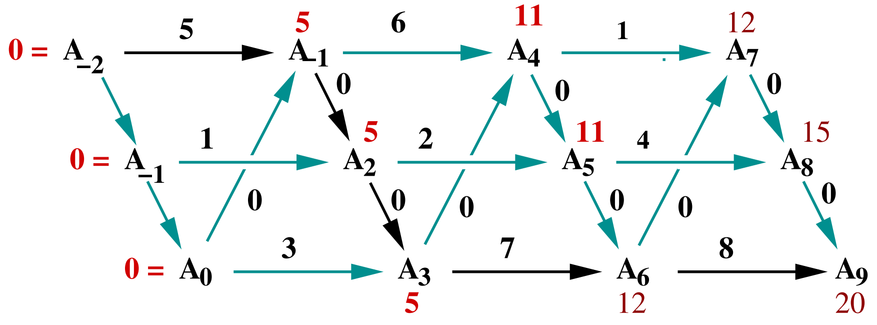
We now consider a more complicated coin row problem. A subset is legal if any two coins in the set must have at least two coins between them in the original row. Again, we use the sequence: 5, 1, 3, 6, 2, 7, 1, 4, 8. As before, we have two dynamic programs.

Best subsequence up to some point

Let $A[i]$ be the maximum weight of any legal subsequence of the first i coins.

```
Program 3

A[-2] = 0; // dummy value
A[-1] = 0; // dummy value
A[0] = 0; // dummy value
for(int i = 1; i <= n; i++)
 A[i] = max(A[i-1],A[i-3]+V[i]);
return A[n];
```



The Graph Corresponding to Program 3

Best subsequence ending at some point

Let $B[i]$ be the maximum weight of any legal subsequence which ends at the i^{th} coin.

```
Program 4

B[-4] = 0; // dummy value
B[-3] = 0; // dummy value
B[-2] = 0; // dummy value
B[-1] = 0; // dummy value
B[0] = 0; // dummy value
for(int i = 1; i <= n; i++)
 B[i] = V[i] + max(B[i-3],B[i-4],B[i-5]);
return max(B[n],B[n-1],B[n-2]);
```

The maximum total legal subsequence is 5, 7, 8.

Do you understand these programs?

The graph for Program 4 is more complex than the others. Can you draw it?

A Harder Version

The coin row problem gets harder as we pick more and more complex definitions of legality. For a harder version, we define a subset to be legal if it does not have any three consecutive coins of the original row.

Writing the dynamic program for this version of the coin row problem is considerably harder than for the previous two. Can you do it?

Bellman Ford Algorithm

We are given a weighted directed graph G with n vertices and m arcs. Negative weights are allowed, but as always, no negative cycle may exist. The source vertex is v_0, and we need to find the minimum weight path from v_0 to v_i for each i.

The time complexity of the Bellman Ford algorithm is $O(n^3)$ in the worst case, but that worst case will seldom occur in practice. Using the code I gave you in class, the time complexity is reduced to $O(n^2 d)$, where d is the maximum number of arcs in any shortest path. However, the time can be reduced even further, to $O(md)$, where m is the number of arcss, a vast improvement if G is sparse.

Bellman Ford is a dynamic programming algorithm, but it is may not be obvious what the subproblems are. Note that G can have cycles, but the graph of subproblems cannot.

There are at most n^2 subproblems. For any $0 \le k \le n$ and any $0 \le h \le n$, the subproblem $S[h,k]$ is defined to be the minimum weight of any path from v_0 to v_k consisting of at most h hops (arcs). The directed acyclic graph of subproblems contains an arc from $S[d,k]$ to $S[d+1,\ell]$ for any k and ℓ provided there is an arc of G from v_k to v_ℓ. The source vertex of the graph of subproblems is $S[0,0]$.

Floyd Warshall Algorithm

We are given a directed weighted graph G of n vertices. The time complexity of the Floyd Warshall algorithm is $\Theta(n^3)$. There are approximately n^3 subproblems, namely $S[i,j,k]$ for all i,j,k from 1 to n. The value of that subproblem is the minimum weight of any path from v_i to v_k whose interior does not contain any vertex of index greater than j. Note, however, that either i or k, or both, could be larger than j. The subproblem $S[i,j,k]$ has three predecessors in the subproblem graph, namely $S[i,j-1,k]$, $S[i,j-1,j]$ and $S[j,j-1,k]$. Do you see why the graph of subproblems is acyclic, even if G is cyclic?

When we encode the Floyd Warshall algrotihm, we do not explicitly work the subproblems $S[i,j,k]$. We instead continually update the value of $S[i,k]$, which is the minimum value of $S[i,j,k]$ for all j up to that point, and hence is (possibly) updated during each iteration of the j-loop. In order to make that work, the outer loop must have iterator j, and the two inner loops iterators i and k.

```
for all i and k // shortcut notation.  Sorry!
   S[i,k] = W[i,k] // possibly infinity
for all i
   S[i,i] = 0
for all j
   for all i and k
      temp = S[i,j] + S[j,k]
      S[i,k] = min(S[i,k],temp)
```