Dynamic Programming

Lecture Algorithm Design

Stefano Leonardi, Rebecca Reiffenhäuser

October 23, 2020

Part 1

The Chinese Sticks Problem

The Chinese Sticks Problem

(Based on the description at this website.)

Assume you have a long stick that needs to be cut into shorter pieces. The cuts must be done in certain parts of the stick. You must cut the stick in all marked places. For example, consider a stick of size 10 and 3 places marked in it (positions 2, 4 and 7) where the cuts must be made.

The cost to make a cut is the same as the size of the stick. For instance, if you have a stick of size 10 and you need to make one cut (anywhere) in it, that will cost you 10.

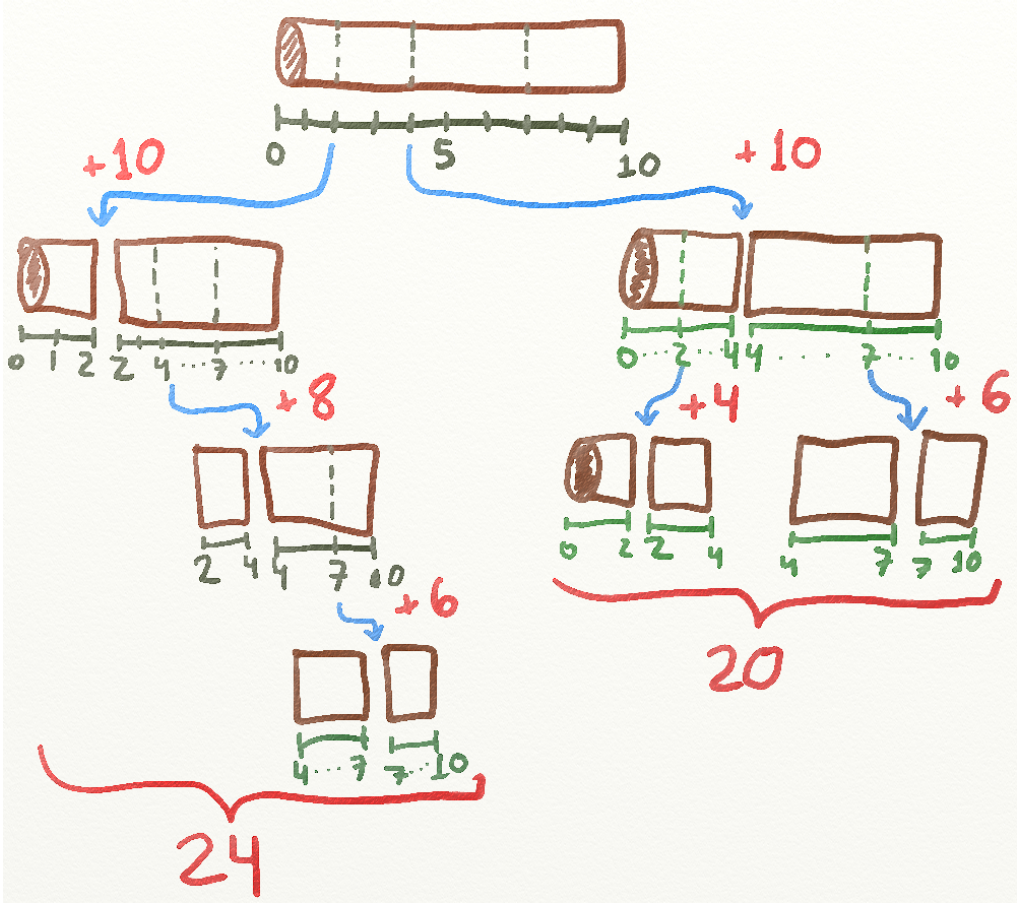
Goal: Find a cheapest way of making all necessary cuts.

Example

Cutting on positions 2, 4 and 7 (one after the other), the total cost will be 10+8+6=24.

A better way to cut that stick would be:

Start on position 4, then 2 then 7. Now the cost is 10+4+6=20.



More Formal Problem Definition

Given a stick of length N, and cutting places $k_1, k_2, \ldots, k_n$, find an ordering $S = (k_{i_1}, k_{i_2}, \ldots, k_{i_n})$ of all k_i such that the total cost

$$cost(S) = \sum_{j=1}^{n} \text{Current length of the stick when cutting at } k_{i_j}$$

is minimized.

Key Observation

Whenever we cut the sticks, we are left with *two* shorter ones that might have to be cut again. The optimal cost for cutting in all the marked places of the original stick equals the optimal costs from the two smaller sticks, plus the length of the original stick.

optimalCost(initialStickPosition, endStickPosition) =
optimalCost(initialStickPosition, k) +
optimalCost(k, endStickPosition) +
sizeOfTheCurrentStick

Recursive Formula

We can replace the stick size by the distance between beginning and end of the stick.

optimalCost(initialStickPosition, endStickPosition) =
optimalCost(initialStickPosition, k) +
optimalCost(k, endStickPosition) +
(endStickPosition - initialStickPosition)

Recursive Algorithm

```
1   int bestCut = INF
2   for(int k=indexOfFirstCut, k<=indexOfLastCut, k=nextCutIndex()){
3     int subCutCost = optimalCost(0, N, k);
4     if(subCutCost < bestCut){
5         bestCut = subCutCost;
6     }
7   }
8
9   int optimalCost(int initialStickPosition,
10                  int endStickPosition,
11                  int k){
12    return optimalCost(initialStickPosition, k) +
13           optimalCost(k, endStickPosition) +
14           (endStickPosition - initialStickPosition)
15  }
```

Problem: Computation Time

Sadly, a lot of possibilities are calculated over and over again.

But, we are only interested in the *best-possible* solution for each sub-stick!

$\rightarrow$ Store this optimum value for every sub-stick so we can just look it up!

Making the Algorithm more Efficient

```
1  int optimalCost(int initialStickPosition,
2                  int endStickPosition,
3                  int k){
4
5    int memorizedResult = m[initialStickPosition][endStickPosition];
6    if(memorizedResult != INF){
7      return memorizedResult;
8    }
9    int bestCost = optimalCost(initialStickPosition, k) +
10                  optimalCost(k, endStickPosition) +
11                  (endStickPosition - initialStickPosition)
12
13   m[initialStickPosition][endStickPosition] = bestCost;
14   return bestCost;
15 }
```

Running Time / Space

At worst, we compute an optimum cutting for every combination of initial and end- stick position.

Those are chosen from the n cutting positions, plus $\{0, N\}$.

$\rightarrow$ we have $O(n^2)$ values to compute and store.

Each actual computation of a value in the matrix m requires to compare all up to n possibilities of cutting this stick in two (except that for shorter sticks, it will of course be less than n).

Part 2

Maximum Weighted Independent Set on a Path

Maximum Weighted Independent Set on a Path

Given are a set V of vertices on a path, and weights $w(v) \geq 0$ for each vertex $v \in V$.
We define a subset $I \subseteq V$ to be an *Independent Set* if for any two vertices $v_1, v_2 \in I$, there exists *no edge* between v_1 and v_2.

Goal: Find a maximum-weight independent set in V.

Problem Structure

An optimal solution can have one of two properties:

- Either, the last element in the path is not part of the maximum weighted independent set
  (then, we know that the solution is equally valid for the subgraph V' formed by deleting the last vertex from the path)
- Or the last element is part of the set
  (then, we know that the predecessor cannot be part of the set, and the solution minus the last vertex is equally valid for the subgraph V' formed by deleting the last two vertices from the path)

Time Considerations

If we knew which of the above options is true (last vertex in or out?), we could walk backwards in linear number of steps!
As previously for Chinese Sticks, doing this the naive way is costly: Trying out both options recursively results in exponential running time.

Solution: Again, remember the result of past computations and look it up instead of doing it all over again!

Algorithm Structure

- Walk through the path from left to right, and use calculated weights to decide whether or not the last vertex is in the set.
- **Base Case:**
  - For empty sets, their weight is zero.
  - For any set $S = \{v\}$, $v \in V$, the weight of the set is $w(v)$. Store $w(v)$ in matrix entry $A[0]$.
- **General Case** (for the i-th vertex v_i):
  - We can either choose the old solution up to the last point, i.e. $A[i-1]$, or the $A[i-2]$ solution (that can never pick the vertex next to the i-th), together with the i-th vertex.
  - Store in $A[i]$ the maximum over $A[i-1]$ and $A[i-2] + w(v_i)$.

Algorithm Properties

The algorithm runs in linear time: it picks a maximum between two values for every vertex on the path.

Although this gives us only the weight of a maximum weight independent set, not the set itself, we can easily adjust the algorithm:

- For every step from left to right along the path, we could also store the selected vertices.
- Or, we go backwards along the steps of the original algorithm to *reconstruct* the solution.

The time complexity is preserved also when the goal is to find the actual independent set.

Algorithm for Reconstruction of the Independent Set

In pseudocode, we could get the set like this:

$S = \emptyset$

v_i = last element in A

While $i \geq 1$:

  If $A[i] = A[i-1]$ // Last vertex not in set S

  $i--$

  Else // Last vertex is in set S

  $S = S \cup v_i$

  $i = i - 2$

  End

End

Return S.

Segmented Least Squares Problem

Slides are the ones belonging to the book, and can be found here.