CS270 Combinatorial Algorithms & Data Structures **Spring 2005**

Lecture 11: 2.22.05

Lecturer: Christos *Scribe: S. Mubaraq Mishra*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

11.1 Dynamic Programming

We will attempt to explain Dynamic Programming with a example. Consider the problem of finding the shortest path in a DAG. Figure 11.1(a) shows the graph while (b) shows the linearized graph. All edges in the linearized graph go from left to right. We wish to find the shortest path from node A to node F.

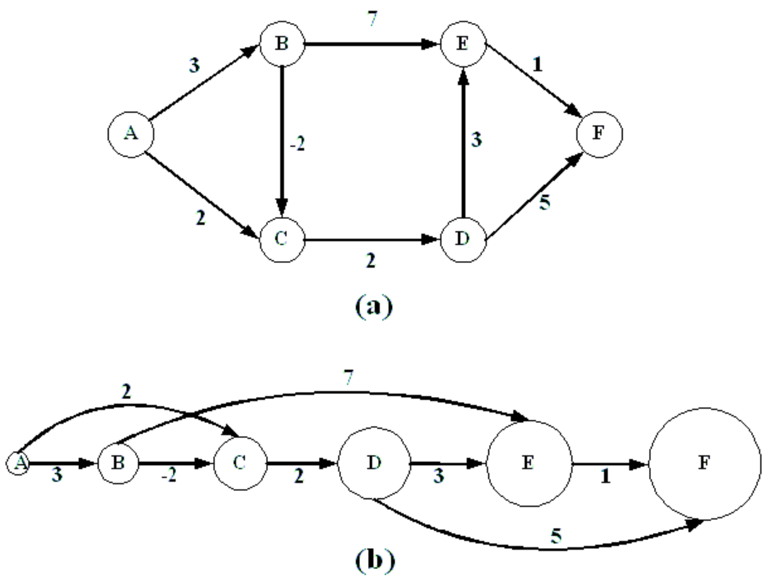

Figure 11.1: (a) DAG (b) Linearized DAG

We can update shortest paths in the order shown in Figure 11.1(b) i.e. (A,B,C,D,E,F).

Algorithm 1 DP algorithm for calculating shortest path in a DAG

```
$d(A) = 0, d(v \neq A) = \infty$ {Initialize the distance}
for $i = A$ to $F$ do
    $d(v) = \min_{(u,v) \in E} (d(u) + l(u,v))$
end for
```

The running time of the above algorithm is O($|E|$). Why does this algorithm work ? The key lies in the property that *any part of the shortest path is also shortest*. The problem of finding the shortest path from node A to node F can be decomposed into finding the shortest path from A to all other nodes i.e. a problem can be solved from the results of similar (hopefully smaller) subproblems. We can write $d(F)$ as

$$d(F) = min(d(E) + 1, d(D) + 5)$$

and we can write the remaining equations as:

$$\begin{aligned} d(E) &= min(d(B) + 7, d(D) + 3) \\ d(D) &= d(C) + 3 \\ d(C) &= min(d(A) + 2, d(B) - 2) \\ d(B) &= d(A) + 3 \end{aligned}$$

Based one the above discussion, we can review some of the salient features of dynamic programming:

Subproblem definition The key in Dynamic Programming is in finding appropoiate subproblems, the solution to which can be used to form the solution of the larger problem. Solution to each subproblem is optimal in itself. In the above example, $d(F)$ is the main problem and $d(E) \ldots d(A)$ are subproblems used to form the solution for d(F).

Dynamic Programming versus Divide and Conquer Dynamic programming is similar to Divide and Conquer in that the solution to the large problem depends on previously obtained solutions to easier problems. The significant difference, however, is that dynamic programming allows the solution of a subproblem to be used in the solution of two different subproblems. In contrast, the divide and conquer approach creates separate, independent problems.Figure 11.2 illustrates these differences. The problem to the solved is representated by the root of the tree, where the children are easier subproblems. Subproblems in divide and conquer do not interact while in D.P. they do. In the above problem, the subproblem $d(B)$ is used for the solution of $d(C)$ as well as $d(E)$.

Algorithmic Sledgehammer Together with linear programming, dynamic programming are powerful algorithmic tools that can be applied to a vast number of problems. This differentiates them from 'shortest path' kind of algorithms which can be applied to very few problems.

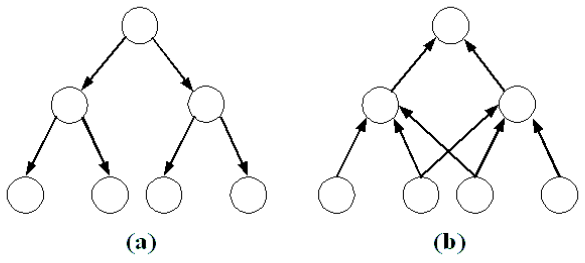

Figure 11.2: (a) Divide and Conquer (b) Dynamic Programming

11.2 Knapsack problem

Definition Given n items of different values and volumes, find the most valuable set of items that fit in a knapsack of fixed volume V. Fractions are not allowed and there is an unlimited supply of objects of each type.

Example Consider a knapsack of volume 10 units and a list of items:

Item	Volume v_i	Value s_i
a	6	\$30
b	3	\$14
c	4	\$16
d	2	\$9

Recursion This problem can be modelled as a D.P. by making the observation that a knapsack of volume v can be formed from a knapsack of volume $v - v_i$ by adding a item of volume v_i. Formally,

$$C(v) = \max_{i:(v-v_i)\geq 0} (C(v-v_i) + s_i)$$

Algorithm 2 DP algorithm for the Knapsack problem

```
$c(0) = 0$ {Initialize the Value}
{V is the target volume of the knapsack}
for $i = 1$ to $V$ do
    $C(v) = \max_{i:(v-v_i)\geq 0} (C(v-v_i) + s_i)$
end for
```

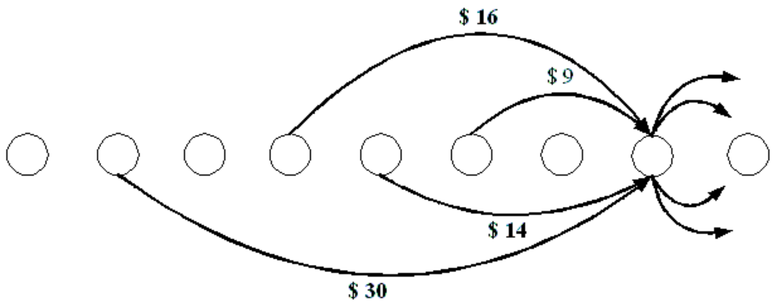

Figure 11.3: Linearized graph for the knapsack problem

Running Time Figure 11.3 shows the knapsack problem as a linearized DAG. The number of nodes in this graph is $|V|$ and each node has indegree n. Hence the running time is O(Vn).

11.3 Knapsack problem with a single unit per item

Definition This problem is the same as the one before, except that, we are restricted to one unit per item.

Recursion When defining the recursion for this problem we need to take into account the number of remaining items. Define $C(v, i)$ as the maximum value obtainable from the first i items and volume v. This could be achieved in the following ways:

- No Use Case: We do not use item $i - 1$. In this case $C(v, i) = C(v, i - 1)$.
- Use Case: In this case we add the i^{th} item to the knapsack. Then $C(v, i) = C(v - v_i, i - 1) + s_i$

Trivially, a knapsack cannot be filled by 0 items and a knapsack of 0 volume cannot be filled either. Hence we have the two trivial subproblems:

$$C(v, 0) \leftarrow 0$$
$$C(0, i) \leftarrow 0$$

Formally, we can write:

$$C(v, i) = \min \begin{cases} C(v, i - 1) \\ C(v - v_i, i) + s_i \end{cases}$$

Algorithm 3 DP algorithm for Knapsack of volume V to be filled with n single unit items)

```
for $i = 0$ to $V$ do
  $C(v, 0) = 0$
end for
for $j = 0$ to $n$ do
  $C(0, j) = 0$
end for
for $v = 1$ to $V$ do
  for $i = 1$ to $n$ do
    $C(v, i) = min(C(v, i - 1), C(v - v_i, i) + s_i)$
  end for
end for
return $C(V, n)$
```

Running Time From Figure 11.4 we see that there are $V \times n$ nodes in the linearized DAG and the indegree of each node is 2. Hence the running time of the program is O($2Vn$) or O(Vn).

11.4 Edit Distance

Definition We wish to modify one string and convert it into another string through a series of character edits (insertion, deletion, overwrite). We would like to minimize the number of edits required to make this transformation. In other words, we would like to find the minimum edit distance between two strings. The cost of all edits is the same.

Example Consider the problem of modifying the word POLYNOMIAL into EXPONENTIAL. To solve this as a D.P., we have to formulate the appropriate subproblem. In this case the subproblem is finding the edit distance between a prefix of EXPONENTIAL and a substring of POLYNOMIAL. Formally, lets define $ed(i, j)$ as the edit distance between POLYNOMIAL$[0 \ldots i]$ and EXPONENTIAL $[0 \ldots j]$.Trivially, we need to make i deletions to get a string of length 0. Hence

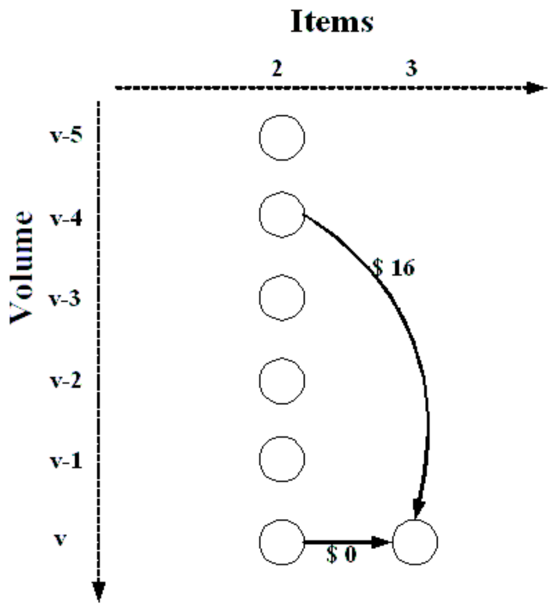

Figure 11.4: Knapsack problem with single unit per item

$$ed(i, 0) \leftarrow i$$

and we need to make j insertions to get a string of length j from an empty string. Hence

$$ed(0, j) \leftarrow j$$

Recursion Figure 11.5 shows the recursion. Imagine we are trying to evaluate $ed(7, 7)$ i.e. modifying POLYNOM into EXPONEN. This could be achieved in the following ways:

- Overwrite Case: We have already edited POLYNO to EXPONE and we can get EXPONEN from POLYNOM by replacing 'm' by 'n'. In this case we are getting $ed(7, 7)$ from $ed(6, 6)$.
- Insert Case: We have managed to obtain EXPONE from POLYNOM and we need to add a 'n' to get EXPONEN. In this case we are getting $ed(7, 7)$ from $ed(7, 6)$.
- Delete Case: We have managed to obtain EXPONEN from POLYNO and we need to delete 'm' from POLYNOM. In this case we are getting $ed(7, 7)$ from $ed(6, 7)$.

Formally, we can write:

$$ed(i, j) = \min \begin{cases} ed(i, j-1) + 1 \\ ed(i-1, j) + 1 \\ ed(i-1, j-1) + POLYNOMIAL(i) \neq EXPONENTIAL(j) \end{cases}$$

Running Time From Figure 11.5 we see that there are $|A| \times |B|$ nodes in the linearized DAG with each node having indegree 3. Hence the running time of the algorithm is O($3|A||B|$) or O($|A||B|$).

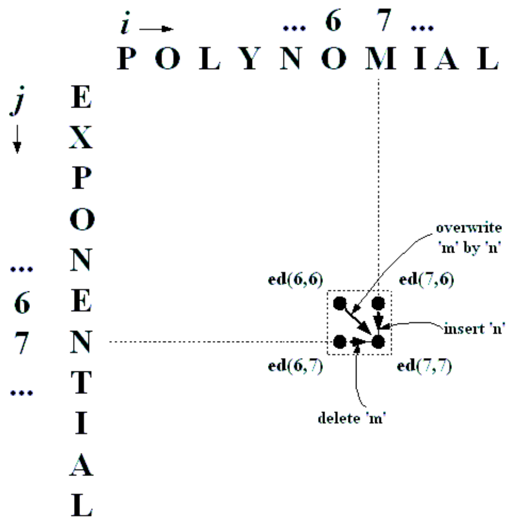

Figure 11.5: Edit Distance

Algorithm 4 DP algorithm for obtain Edit Distance from string A to B)

```
for $i = 0$ to $|A|$ do
  $ed(i, 0) = i$
end for
for $j = 0$ to $|B|$ do
  $ed(0, j) = j$
end for
for $i = 1$ to $|A|$ do
  for $j = 1$ to $|B|$ do
    $ed(i, j) = min(ed(i, j - 1) + 1, ed(i - 1, j) + 1, ed(i - 1, j - 1) + a(i) \neq b(j))$
  end for
end for
return $ed(i, |B|)$
```