A BRIEF INTRODUCTION
TO
DYNAMIC PROGRAMMING (DP)

by
Amarnath Kasibhatla
Nanocad Lab
University of California, Los Angeles
04/21/2010

Overview

- What is DP?
- Characteristics of DP
- Formulation
- Examples
- Disadvantages of DP
- References

WHAT IS DP?

- *Dynamic Programming* (DP) is a commonly used method of optimally solving complex problems by breaking them down into simpler problems.

- Dynamic programming is both a mathematical optimization method and a computer programming method. It is applicable to both discrete and continuous domains.

- Richard Bellman pioneered the systematic study of dynamic programming in the 1950s.

Popular problems/applications that use DP (not an exhaustive list):

- *Knapsack (0/1, integer)*
- *Shortest path on a DAG*
- *Matrix Chain multiplication problem*
- *Longest common subsequence*
- *VLSI CAD problems, e.g., Gate sizing, Placement, Routing etc.*
- *Queuing theory, Control theory, Bioinformatics, Information theory, Operations Research etc.*
- *Multiple-class Mean Value Analysis (MVA) etc.*

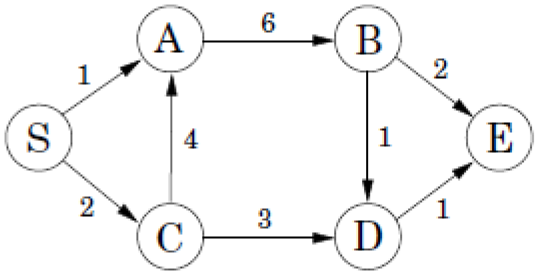
CHARACTERISTICS OF DP

- DP is applicable to problems that exhibit the properties of overlapping subproblems which are only slightly smaller and optimal substructure.

- *Optimal substructure (Shortest path example)* :

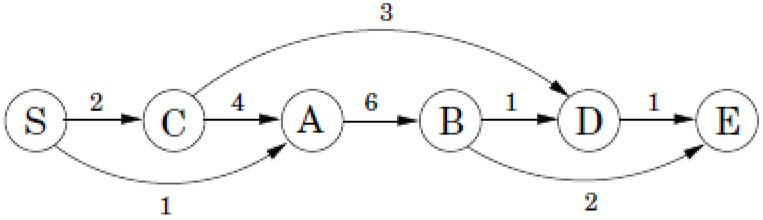
  ➢ Let's say we need to find the shortest distance from node S to node D. Predecessors of D are B and C.

  To find the shortest path to D:



dist(D) = *min*{ dist(B)+1, dist(C)+3}
dist(B) = dist(A) + 6
dist(C) = dist(S) + 2
dist(A) = min{ dist(S)+1, dist(C)+4}

initialize all $\mathtt{dist}(\cdot)$ values to ∞
$\mathtt{dist}(s) = 0$
for each $v \in V \backslash \{s\}$, in linearized order:
$\quad \mathtt{dist}(v) = \min_{(u,v)\in E}\{\mathtt{dist}(u) + l(u,v)\}$



  ➢ If the shortest path involves to D involves the path from S to D has the node C, then the shortest path from S to C and shortest path from C to D are the optimal subsolutions of the actual problem.

CHARACTERISTICS OF DP (Contd.)

- *Overlapping subproblems (Fibonacci series example)* :

Fibonacci series 0, 1, 1, 2, 3, 5, 8 …

- A naive implementation of finding n^{th} Fibonacci number is :

```
Function fib(n)
     if n =0 return 0
    else if n =1 return 1
    else     return fib(n-1) + fib (n-2)
```

But this involves repeated calculations – for higher numbers it leads to exponential time!!!
Eg. Fib(4) = fib(3) + fib(2), fib(3) = fib(2) + fib(1)

- Bottom-up approach of DP: Memorize and use solutions of previously solved subproblems

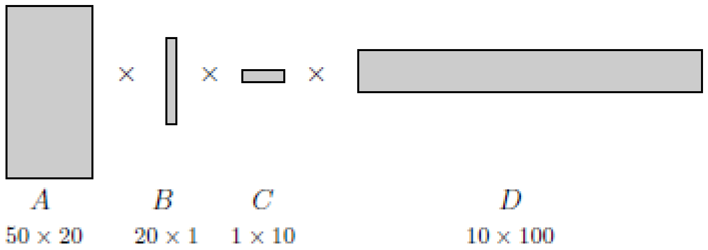
```
Function fib(n)
      var previousFib := 0, currentFib := 1
      if n =0 return 0
      else if n = 1 return 1
      repeat n-1 times
      var newFib := previousFib + currentFib
      previousFib := currentFib
      currentFib := newFib
      return currentFib
```

Example: fib(42) = fib(41) + fib(40)

- *O(n) is the time complexity and O(1) is space complexity, compared to exponential complexity of naïve method.*

EXAMPLE 1 – OPTIMAL MATRIX MULTIPLICATION ORDER

- Determine the optimal order of A *x* B x C *x* D
- An optimal multiplication order can reduce the computations by orders of magnitude.



- General problem: $A_1 \times A_2 \times A_3 \times \dots \times A_n$

Parenthesization	Cost computation	Cost
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	120,200
$(A \times (B \times C)) \times D$	$20 \cdot 1 \cdot 10 + 50 \cdot 20 \cdot 10 + 50 \cdot 10 \cdot 100$	60,200
$(A \times B) \times (C \times D)$	$50 \cdot 20 \cdot 1 + 1 \cdot 10 \cdot 100 + 50 \cdot 1 \cdot 100$	7,000

Subproblems : $A_i \times A_{i+1} \times \dots A_j$, *1<= I <= j <= n*

Define C(*i,j*) = minimum cost of multiplying $A_i \times A_{i+1} \times \dots A_j$

Solve a subproblem by the splitting into two pieces $A_i \times \dots \times A_k$, $A_{k+1} \times \dots \times A_j$ for *i<= k < j*
The cost of the subproblem is the cost of these two pieces and the cost of combining them $C(i,k) + C(j,k) + m_{i-1} * m_k * m_j$
For every subproblem we just need to find splitting point **k** such that

$$C(i,j) = \min_{i \le k < j} \{C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j\}$$

The pseudo code:

for $s = 1$ to $n-1$:
  for $i = 1$ to $n-s$:
    $j = i + s$
    $C(i,j) = \min\{C(i,k) + C(k+1,j) + m_{i-1} \cdot m_k \cdot m_j : i \le k < j\}$
return $C(1,n)$

The complexity of $O(n^3)$. The optimal order is obtained by tracing back the values of k for each subproblem.

EXAMPLE 2 – 0/1 KNAPSACK PROBLEM

- Mathematical Optimization – 0/1 Knapsack problem.
- Given n objects and a "knapsack".
- Item I weights $w_i > 0$ Kgs and has value $v_i > 0$.
- Knapsack has capacity of W Kgs.
- Goal: fill knapsack so as to maximize its total value.

- OPT(i,w) = max profit subset of items 1…i with weight limit w

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w),\ \ v_i + OPT(i-1,w-w_i)\} & \text{otherwise} \end{cases}$$

Pseudo Code to build the table:

```
Input: n, w1,…,wN, v1,…,vN

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}

return M[n, W]
```

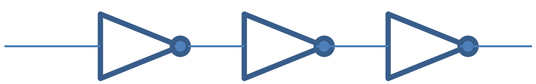
	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

This is a pseudo-polynomial time algorithm with complexity O(n*W)

EXAMPLE 3 – OPTIMAL SIZING OF AN INVERTER CHAIN

- **Optimally gate sizing an inverter chain for a given timing constraint using DP.**

E.g. problem: Minimize power for D_{max} = 8



DELAY TABLE FOR THE INVERTER ASSUMED IN NUMERICAL EXAMPLE

	Input cap	Leakage power	Delay	
			Load cap 3	Load cap 6
Size 1	3	5	3	4
Size 2	6	10	1	2

- Simple enumeration will take $O(k^N)$

- Time complexity in this case is O(k*B*N) for k gate sizes, delay budget of B and N number of inverters in the chain.

NUMERICAL EXAMPLE FOR A THREE-STAGE INVERTER CHAIN . THE FINAL OPTIMAL SIZING SOLUTION IS SHOWN IN **bold** FONT

Output cap	Stage 1			Stage 2			Stage 3		
	Budget	CP	OS	Budget	CP	OS	Budget	CP	OS
3	**1**	**10**	**2**	3	20	2			
3	2	10	2	**4**	**15**	**1**			
3	3	5	1	5	15	2			
3	4	5	1	6	10	1			
3	5	5	1	7	10	1			
3	6	5	1	8	10	1			
3	7	5	1						
3	8	5	1						
6	2	10	2	4	20	2	**8**	**20**	**1**
6	3	10	2	5	15	1			
6	4	5	1	6	15	2			
6	5	5	1	7	10	1			
6	6	5	1	8	10	1			
6	7	5	1						
6	8	5	1						

DISADVANTAGES OF DP

- “Curse of dimensionality” – Richard Bellman:
- Runtime is strongly dependent on the range of *state* variable (example the weight capacity W of the knapsack), so we cannot guarantee bounds on the runtime.
- Problems involving fractional state variable values can lead exponential increase in the iterations (time complexity).
- The storage space (space complexity) is strongly dependent on the state variable and can also be .
- Is only applicable to problems with identified overlapping subproblems and optimal substructures. Many problems use using dynamic programming locally to solve the larger problem.
- Establishing/identifying the optimal substructure and the DP recursion is not a trivial task for large problems.

REFERENCES

- R. Bellman, ***Dynamic Programming***. *Dover Publications, N.Y, 1957.*
- Bellman, R. and S. Dreyfus (1962) ***Applied Dynamic Programming*** *Princeton University Press* Princeton, New Jersey.
- Blackwell, D. (1962) **Discrete Dynamic Programming**. *Annals of Mathematical Statistics* ***33, 719-72*****6**.
- Chow, C.S. and Tsitsiklis, J.N. (1989) **The Complexity of Dynamic Programming**. *Journal of Complexity* **5 466.488.**
- Eric V. Denardo ***Dynamic programming: models and applications, 2003.***
- *www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf*
- *http://www2.fiu.edu/~thompsop/modeling/modeling_chapter5.pdf*
- *http://mat.gsia.cmu.edu/classes/***dynamic/dynamic***.html*