CAD of VLSI

Tutorial #11
Dynamic programming
Introduction – Algorithmic paradigms

• **Greed.** Build up a solution incrementally, optimizing some local criterion.

• **Divide-and-conquer.** Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

• **Dynamic Programming (DP).** Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.
  – **Like** divide-and-conquer, DP solves problems by combining solutions to sub-problems.
  – **Unlike** divide-and-conquer, sub-problems are not non-overlapping.
    • Sub-problems may share sub-sub-problems, etc.
Simple example - Fibonacci numbers

• Fibonacci numbers:  \( a_1 = 1; \ a_2 = 1; \ a_n = a_{n-1} + a_{n-2} \)

• Recursive solution (brute force):

```c
int fibo(int n) {
    if ((n==1) || (n==2)) return 1;
    else return fibo(n-1) + fibo(n-2); }
```

• Run-time complexity – \( O(2^n) \)

• Why so bad?
  – Recomputing results instead of reusing results

• Instead, compute bottom up and store
  – Actually one needs to only store the previous two results for this problem

• The above is a simple example of dynamic programming
Example from CAD
Tree covering for technology mapping

• The problem:
  – Find covering of subject graph (tree) with minimal area
  • There is an extension for min-delay covering

Or:

Based on Susan Lysecky slides
Tree covering
Brute force approach

• Brute force solution, using back tracking:
  *Start from root, try all possible patterns*
  *For each root pattern, try all possible patterns for its descendants, etc.*

• Run-time:
  – For each node try all possible patterns
  – For tree with $n$ nodes and $p$ patterns in library run time is $\sim O(n^p)$

• Should look for another approach
Tree covering
Dynamic programming

- Revealing optimal substructure:
  - If pattern P is min cost match at some node of subject tree...
  - then it must be that each *leaf* of pattern tree is also the *root* of some min cost matching pattern

- Assume three different patterns match at root of subject tree
  - Pattern P1 has 2 leaf nodes: a and b
  - Pattern P2 has 3 leaf nodes: x, y and z
  - Pattern P3 has 4 leaf nodes: j, k, l and m
  - Which is the cheapest pattern if we know cost of each pattern?
Min cost tree cover

- Cheapest cover of root of subject tree is $\text{mincost}(\text{root})^* = \min(\text{patterncost}(P_1) + m.c-(a) + m.c-(b), \text{patterncost}(P_2) + m.c-(x) + m.c-(y) + m.c-(z), \text{patterncost}(P_3) + m.c-(j) + m.c-(k) + m.c-(l) + m.c-(m))$

- Each rectangle means recursive call to $\text{mincost}$ (subtree)*

- This shows optimal substructure of covering problem

Based on R. Rutenbar slides
Tree covering
Dynamic programming

- **Revealing overlapping sub-problems**
  - Assume we calculate tree cost top-down
  - For picture below: node “y” in the subject tree will get its mincost cover computed \( \text{mincost}(y) \) when we put P2 at the root of the subject tree
    - … and again, when we put P3 at the root
  - Instead, calculate tree cost **bottom-up**
    - Will have to calculate \( \text{mincost}(y) \) only once and store — memoization
  - This reminds Fibonacci numbers example …

Based on R. Rutenbar slides
Tree covering
Dynamic programming

• Assume table[node] = ∞ for all nodes at the beginning
• The algorithm:

```java
mincost(treenode) {
  if (table[treenode] < ∞) // was already calculated
    then return(table[treenode]);

  cost = ∞
  foreach(pattern P matching at subject treenode) {
    let L = {nodes in subject tree corresponding to leaf nodes in P
             when P is placed with its root at treenode }
    newcost = patterncost(P)
    foreach(node n in L) {
      newcost = newcost + mincost(n);
    }
    if (newcost < cost)
      then {
        cost = newcost;
        table[treenode] = newcost
        treenode.selected = P;
      }
  }
}
```

Cell Library

Based on R. Rutenbar slides

Numbers in parenthesis represent cost
Tree covering
Dynamic programming

- The algorithm works bottom up
- For each node, checks all possible patterns that can be rooted at this node and combines each pattern’s cost with optimal solutions of sub-trees rooted at leafs of the pattern
- Only optimal solution is saved in each node
- Stage: solution for the sub-tree
- State expansion: check all possible patterns that can be rooted at the node
- State pruning: save only the solution with minimal cost
- In this case:
  - number of stages: \( n = \) number of nodes
  - state expansion: check \( p \) patterns for each node, i.e. each state produces \( p \) states in the next stage
  - state pruning: save only the best case, i.e. prune \( p-1 \) states, leave only \( 1 \)
  - i.e. number of states growths in \( O(1) \) from stage to stage
  - Therefore, the run time complexity is \( O(pn) \), space complexity is \( O(n) \)
Tree covering
Dynamic programming - Example

• Cover following circuit using DP approach:

Based on Susan Lysecky slides
Tree covering
Dynamic programming - Example

- NAND2 is only match for node a
Tree covering
Dynamic programming - Example

NAND2 [a] (2*)
 INV [b] (1*)

NAND2 [c] (5*) = 2_{nand2} + 2^* + 1^*

- NAND2 is only match for node c
- INV is only match for node b

Based on Susan Lysecky slides
Tree covering
Dynamic programming - Example

- NAND2 is only match for node e
- INV is only match for node d
- INV is possible match for node f
- AOI21 is possible match for node f too
- NAND2 is possible match for node g
Tree covering
Dynamic programming - Example

-INV is only match for node h
-NAND2 is possible match for node i
-NAND3 is possible match for node i too

Based on Susan Lysecky slides
Tree covering
Dynamic programming - Example

-NAND2 is only match for node j
-NAND3 is possible match for node j too
Tree covering
Dynamic programming - Example

• Now backtrace to reveal optimal cover
Tree covering

Dynamic programming - Example

Based on Susan Lysecky slides
Dynamic Programming principles:

1 – principle of optimality

• The principle of optimality holds if the problem has optimal substructure: an optimal solution to a problem is built from optimal solutions to all sub-problems

• It means that usually optimal solution can be described recursively in terms of optimal solutions to sub-problems

• The principle of optimality does not say
  – If you have optimal solutions to all sub-problems...
  – ...then all you need to do to get an optimal solution is to combine (some of / any combination of) them.
Example – principle of optimality

- Finding shortest path in a graph

\[ \text{DP can be used to find the shortest path in a DAG.} \]

- The shortest paths \(A \rightarrow D, A \rightarrow E, A \rightarrow F\) are solutions of sub-problems of the shortest path problem \(A \rightarrow G\) and

- The shortest paths \(G \rightarrow H, G \rightarrow J, G \rightarrow K\) are solutions of sub-problems of the shortest path problem \(G \rightarrow L\) but

- The shortest path from \(A\) to \(L\) is not necessarily the shortest path from \(A\) to \(G + \) the shortest path from \(G\) to \(L\) \(\text{ (because perhaps we have } C \rightarrow K)\)
  - However, there is some way to divide path \(A \rightarrow L\) into sub-problems with optimal solutions, that will give an optimal solution for \(A \rightarrow L\)
  - Hence, the principle of optimality holds for this problem
Dynamic Programming principles:

2 – overlapping sub-problems

• A recursive solution contains “small” (polynomial) number of distinct sub-problems repeated many times

• Memoization: after computing a solution to a sub-problem, store it in a table. Subsequent calls check the table to avoid redoing work.

• The optimal solution to a sub-problem is obtained usually by pruning

Example: in computation of the shortest path A → H:

• Optimal solution to sub-problem A → G is obtained by min: (A → D, A → E, A → F), i.e. pruning of two sub-optimal solutions

• Optimal solution to sub-problem A → G is then memoized (saved) and used later for solving A → H (no need to store A → D, A → E, A → F any longer)

(just store {G, Cost_G} and the A → G)
Dynamic Programming principles: 3 – *bottom-up fashion*

- Compute the value of optimal solution in bottom up fashion
- In our examples:
  - In Fibonacci number calculation: use stored values for numbers 1, 2, …, n-1
    - Calculation in $O(n)$ time instead of $O(2^n)$ by recursive algorithm
  - In the shortest path calculation: calculate shortest path from node 0 to node n by
    \[
    L_{0\rightarrow n} = \min_{\text{over all nodes } i \text{ connected by edge to } n} \left( L_{0\rightarrow i} + d_{i\rightarrow n} \right)
    \]
Stage – state representation of Dynamic Programming

- **Stage** is the set of all possible solutions for the sub-problem of size k (called the $k^{th}$ stage)
- **State** is the single solution in the stage
- Each state in the $k^{th}$ stage leads to number of states in the $(k+1)^{th}$ stage
  - This is called state expansion
- After generation of all states in the next stage only the non-redundant states are saved
  - Memoization is exactly this
  - This operation called state pruning
- Non-redundant states generate $(k+1)^{th}$ stage

Stage k → Expansion → Pruning → Stage k+1
Stage – state representation of Dynamic Programming

The route with \( \rightarrow \) was pruned and so did the route with \( \rightarrow \rightarrow \).
To analyze complexity of DP algorithm:
- Should know total number of stages (sub-problems)
- How states are expanded & pruned, i.e.
- How number of states grows from stage $i$ to stage $i+1$

In order to ensure that dynamic programming solves the problem in polynomial time and space:
- the number of sub-problems (stages) should be polynomial in the input size
- the number of states after expansion should be polynomial in the input size
- the number of states after pruning should be polynomial in the input size
  - Usually it is linear in the input size
Examples from CAD

• The problems that can be solved by DP:
  • Min area tree covering for technology mapping
  • Min-area floorplanning for slicing-floorplan
  • Min-delay (or min-power) interconnect tree sizing
  • Min-delay (or min-power) bus sizing
  • Shortest path finding

• The problems that cannot be solved by DP:
  • Min-area DAG covering for technology mapping
  • Min-length multiport routing
  • Min-area placement

Based on Susan Lysecky slides
Summary

- Keep in mind 3 principles of Dynamic Programming:
  - **Optimal substructure:** an optimal solution to a problem contains optimal solutions to all sub-problems
  - **Overlapping sub-problems:** A recursive solution contains “small” (polynomial) number of distinct sub-problems repeated many times
  - **Bottom-up fashion:** Compute the value of optimal solution in bottom up fashion

- State-stage representation:
  - Stage
  - State
  - State expansion
  - State pruning

- Examples of uses in CAD:
  - Technology mapping
  - Floorplanning
  - Interconnect tree sizing
  - Bus sizing