1.204 Lecture 13

Dynamic programming:
Method
Resource allocation

Introduction

• Divide and conquer starts with the entire problem, divides it into subproblems and then combines them into a solution
  – This is a top-down approach
• Dynamic programming starts with the smallest, simplest subproblems and combines them in stages to obtain solutions to larger subproblems until we get the solution to the original problem
  – This is a bottom-up approach
• Dynamic programming is used much more than divide and conquer
  – It is more flexible and controllable
  – It is more efficient on most problems since it must consider far fewer combinations
Principle of optimality

• “Principle of optimality”:  
  – In an optimal sequence of decisions or choices, each subsequence must also be optimal 
  – For some problems, an optimal sequence may be found by making decisions one at a time and never making a mistake 
    • True for greedy algorithms (except label correctors) 
  – For many problems it’s not possible to make stepwise decisions based only on local information so that the sequence of decisions is optimal. 
    • One way to solve such problems is to enumerate all possible decision sequences and choose the best 
    • Dynamic programming can drastically reduce the amount of computation by avoiding sequences that cannot be optimal by the “principle of optimality”

Project selection example

• Suppose we have: 
  – $4 million budget 
  – 3 possible projects (e.g. flood control) 
    • Each funded at $1 million increments from $0 to $4 million 
    • Each increment produces a different marginal benefit 
  – Dynamic programming problems are usually discrete, not continuous 
• We want to find the plan that produces the maximum benefit 
• Stages are the number of decisions to be made 
  – We have 3 stages, since we have 3 projects 
• States are the number of distinct possibilities 
  – At each stage there are 5 states ($0, 1, 2, 3, 4 million)
Project selection formulation

- We build a multistage graph to represent this problem:
  - Source node at start of graph, representing ‘null’ initial stage
  - Set of nodes at each stage for each state
  - Sink node at end of graph, which is a collapsed representation of the final state
- Each node characterized by $V(i,j)$:
  - $V(i,j)$ is value (benefit) obtained up to (but not including) stage $i$ by committing $j$ resources
  - Each node also stores its predecessor node in $P(i)$
- Each arc is characterized by $E(m,n)$:
  - $E(m,n)$ is value obtained by spending $n$ resources on project $m$

Project selection data

<table>
<thead>
<tr>
<th>Project 0</th>
<th>Project 1</th>
<th>Project 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Investment</td>
<td>Benefit</td>
<td>Investment</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>4</td>
</tr>
</tbody>
</table>

- In theory, projects could have dependencies, but in practice it’s an improbable model. In the example above:
  - Project 1’s benefits could depend on project 0 investment
    - But not on project 2 investment
  - Project 2’s benefits could depend on total project 0 and 1 investment
    - But not on either individually
- (There are some chip power management graphs with such dependencies)
Dynamic programming graph: feasible

Stage:
0 1 2 3
| Project 0 decisions | Project 1 decisions | Project 2 decisions |

0 1 2 3
| Project 0 decisions | Project 1 decisions | Project 2 decisions |

0 1 2 3
| Project 0 decisions | Project 1 decisions | Project 2 decisions |
Dynamic programming graph: feasible

Stage:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Project 0 decisions</th>
<th>Project 1 decisions</th>
<th>Project 2 decisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dynamic programming graph: feasible

Stage:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Project 0 decisions</th>
<th>Project 1 decisions</th>
<th>Project 2 decisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dynamic programming graph: feasible

Solution

- Generate graph in forward direction:
  - Start at source node
  - Compute $V(i,j)$ and $E(m,n)$ as graph is built
  - Keep track of predecessor $P(i)$ of each node that yields highest $V(i,j)$
    - This eliminates non-optimal subsequences ("pruning")
  - Eliminate infeasible arcs and nodes as graph is built
    - Rule is easy: Check budget constraint at each node; do not generate arcs or nodes that would violate it
  - End when sink node is reached from all nodes of previous stage
- Construct solution by tracing back from sink to source using predecessor variable
Multistage graph problem characteristics

• Multistage graph is the standard DP first example
  – Graph is reduced by applying feasibility constraint to eliminate many combinations
    • Can’t exceed resource limit
  – Each stage is independent of all previous stages
    • How you got to V(i,j) doesn’t matter
    • This limits the combinatorial aspect of the original problem
    • A naïve approach would have looked at all project combinations

• Principle of optimality:
  – “In an optimal sequence of decisions or choices, each subsequence must also be optimal”
  – In our example subsequences are optimal:
    • Node 0 to node 2 (trivially)
    • Node 0 to node 2 to node 10
    • Node 0 to node 2 to node 10 to node 11 (full sequence)

Complexity of multistage graph

• Complexity of well-behaved multistage graph:
  – M projects or stages
  – At each stage, there are ~n^2/2 comparisons to find V(i,j) from the incoming arcs
    • Where n is number of resource levels
  – This is O(Mn^2)
  – Horowitz and Sahni call it O(M+a)
    • Where a is number of arcs since they assume the graph has already been built and is available as input

• Complexity of worst case:
  – Worst case:
    • Different resource levels in each project, so number of nodes increases at each stage
    • High constraint (large resource limit), so no elimination of nodes
    • Number of nodes doubles in each stage
    • This is O(2^n)
  • Thus, complexity is O( min( Mn^2, 2^n ) )
Dynamic programming ‘curses’

- Dynamic programming (DP) isn’t natural for most problems
  - Most dynamic programming problems are \(O(2^n)\)
  - Stages and states have ‘curse of dimensionality’:
    - Stages and states can explode combinatorially
    - Challenge in DP formulation is to avoid or limit the curse...
  - Multistage graph is easiest
  - We’ll do a job scheduling DP next
    - Another example of using the multistage graph model
  - And then it gets harder…
    - We’ll do a set-based DP model for a knapsack problem
    - Sets are “standard model” for complex DP

Multistage graph Java implementation

- Build graph going forward
  - Don’t need graph data structure
    - Don’t need to create or store arcs
    - All information can be stored in nodes
    - Store predecessor of each node (implicit arc)
      - Source, next set of nodes and sink are special cases
  - Read off solution going backward from sink
    - Follow predecessors from sink to source
    - Subtract cumulative resources, profits at each step (arc) to
      recover the decision on each project
  - Allocate \(n+1\) nodes per stage if resource limit= \(n\)
    - If \(n=4\), need 5 nodes for resource level 0, 1, 2, 3, 4
  - Nested Node class holds profit, resource, predecessor
  - Java garbage collector will clean up Nodes not on optimal subsequences
    - No ‘predecessor’ will refer to them
public class MultiStageGraph {
    private static class Node {
        private int projNbr;  // Project number
        private int cumResource; // Resource allocated so far
        private int cumProfit; // Profit so far
        private Node predecessor; // Previous node in graph
        public Node(int proj, int res, int prof, Node p) {
            projNbr = proj;
            cumResource = res;
            cumProfit = prof;
            predecessor = p;
        }
    }
    private int numProj;   // No of projects
    private int n;        // Max units of resource + 1
    private Node root;   // First node in graph
    private Node sink; // Last node in graph
    public MultiStageGraph(int np, int n) {
        this.numProj = np;
        this.n = n;        // root, sink null initially
        // See download for get, set...}
}

public void buildGraph(int[][] p) { // Profit by project
    // Store previous stage nodes; need at next stage
    Node[] prevStage = new Node[n];
    // Store current stage nodes
    Node[] currStage = new Node[n];
    // Stage 0 start node, units so far 0, profit so far 0
    root = new Node(0, 0, 0, null);
    Node currentNode = null;
    // Project (stage) 1 start nodes as special case,
    // since they have single arcs back to root
    for (int i = 0; i < n; i++) {
        // Stage 1 start node has stage 0 profit
        currentNode = new Node(1, i, p[0][i], root);
        prevStage[i] = currentNode;
    }
MultistageGraph: buildGraph() 2

// Stage 2 start nodes thru stage M-1 start nodes
for (int i = 2; i < numProj; i++) {
   // Loop, giving 0-> n resources to project
   for (int j = 0; j < n; j++) {
      currentNode = new Node(i, j, 0, null);
      currStage[j] = currentNode;
      for (int k = 0; k <= j; k++) { // Arcs from prev stage
         Node pastNode = prevStage[j - k];
         int profit = p[i-1][k];
         int cumProfit = profit + pastNode.cumProfit;
         if (cumProfit >= currentNode.cumProfit) {
            currentNode.cumProfit = cumProfit;
            currentNode.predecessor = pastNode;
         }
      }
   }
   // Copy current node array into previous node array
   for (int j = 0; j < n; j++) {
      prevStage[j] = currStage[j];
   }
}
// End buildGraph()
MultistageGraph: backwardPass()

```java
public int backwardPass() {
    System.out.println("Problem solution:");
    System.out.println(" Total profit: " + sink.cumProfit);
    System.out.println(" Total units: " + sink.cumResource + "\n");
    Node next = sink;
    Node current = sink.predecessor;

    while (current != null) {
        System.out.println("Project: " + current.projNbr);
        // Difference in units is project units assigned
        int units = next.cumResource - current.cumResource;
        // Difference in profit is project profit
        int profit = next.cumProfit - current.cumProfit;
        System.out.println(" Units: " + units);
        System.out.println(" Profit: " + profit);
        next = current;
        current = current.predecessor;
    }
    return sink.cumProfit;
}
```

// Better implementation would return 2-D array of (resource, profit) for each project

MultistageGraph: main()

```java
public static void main(String[] args) {
    int numProjects = 3;
    int maxResource = 4;
    int[][] p2 = {{0, 6, 8, 8, 10}, // Project 0 profits
                  {0, 5, 11, 16, 17}, // Project 1 profits
                  {0, 1, 4, 5, 6}, }; // Project 2 profits
    // Increment maxResource: e.g., if maxResource=4,
    // we have 5 decision levels {0,1,2,3,4}
    MultiStageGraph g2 =
            new MultiStageGraph(numProjects, ++maxResource);
    g2.buildGraph(p2);
    int totalProfit = g2.backwardPass();
    System.out.println("Total profit: " + totalProfit);
}
Summary

• Dynamic programming key concepts
  – Stages: Decision points
  – States: Decision options
  – Principle of optimality
    • “In an optimal sequence of decisions or choices, each subsequence must also be optimal”
  – Solution approach: create solution graph
    • Eliminate infeasible combinations at each stage
    • Prune suboptimal combinations at each stage
    • Track predecessor of optimal subsequences to each stage
    • (Can generate graph going forward or backward)
  – In most problems, DP is a heuristic solution approach
    • Eliminate/prune unlikely combinations but not provably suboptimal