Dynamic programming formulation

• To formulate a problem as a dynamic program:
  – Sort by a criterion that will allow infeasible combinations to be eliminated efficiently
  – Choose granularity (integer scale or precision) that allows dominated subsequences to be pruned
    • Choose coarsest granularity that works for your problem
  – Use dynamic programming in fairly constrained problems with tight budgets and bounds
    • If problem is not highly constrained, you will need to apply heuristic constraints to limit the search space
  – Choose between multistage graph, set or custom implementation
    • Decide if a sentinel is helpful in set implementation
  – Experiment
    • Every problem is a special case, since DP is $O(2^n)$
    • Can you find special structure that makes your DP fast?
DP examples

- This lecture shows another example
  - Job scheduling, using multistage graph
    - Example of sorting, feasibility, pruning used effectively
    - Example of good software implementation
      - No graph data structure built; solution tree built directly
      - Good but not ideal representation of tree/graph nodes; some nodes are created but not used
      - We don't even consider 2-D arrays, linked lists, etc., which do not scale at all, but which are popular in many texts. Crazy
    - Good DP codes are somewhat hard to write; there is much detail to handle and many lurking inefficiencies to combat
      - We will not dwell on the code details, but they are important
  - Knapsack problem in next lecture, using sets
    - Example of sorting, feasibility, pruning in different framework
    - Multistage graph doesn't work: too many nodes per stage
    - Object oriented design is big improvement over past codes
      - Be careful: many texts have zillions of inefficient, tiny objects

Job scheduling dynamic program

- Each job to be scheduled is treated as a project with a profit, time required, and deadline
  - We have a single machine over a given time (resource)
  - Use multistage graph formulation from last lecture
- Algorithm pseudocode:
  - Sort jobs in deadline order (not profit order as in greedy)
  - Build source node for job 0
  - Consider each job in deadline order:
    - Build set of nodes for next stage (job) for each state (time spent)
    - For current job:
      - Build arc with no time assigned to job
      - If time so far + current job time <= job deadline, build arc with job done
  - Build sink node for artificial last job
  - Trace back solution using predecessor nodes
Job scheduling algorithm

- We will label every node in the graph that we encounter with its profit and time used
  - If we find a better path to that node, we update its profit and time labels
  - This is exactly the same as the shortest path label correcting algorithm
    • We know this algorithm runs fast
- The issue is then: how big is the graph?
  • A smart formulation keeps the graph size at some polynomial bound in the problem size
  • Otherwise, the graph becomes exponentially large and this is why dynamic programming worst case is exponential
- If our model is good, we also need a good implementation
  • A bad implementation can make a good model run very slowly
  • (A good implementation can't really speed up a bad model...)

Job scheduling example

<table>
<thead>
<tr>
<th>Job</th>
<th>Deadline</th>
<th>Profit</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>39</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>90</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>88</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>37</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>25</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>70</td>
<td>1</td>
</tr>
</tbody>
</table>

4 time units of machine time available.
Job scheduling graph: forward

Stage:
0 1 2 3
<table>
<thead>
<tr>
<th>Job 0 decision</th>
<th>Job 1 decision</th>
<th>Job 2 decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profit: 39</td>
<td>Profit: 90</td>
<td>Profit: 88</td>
</tr>
<tr>
<td>Time: 1</td>
<td>Time: 1</td>
<td>Time: 2</td>
</tr>
<tr>
<td>Deadline: 1</td>
<td>Deadline: 2</td>
<td>Deadline: 2</td>
</tr>
</tbody>
</table>

Job scheduling graph: backward

Stage:
0 1 2 3
<table>
<thead>
<tr>
<th>Job 0 decision</th>
<th>Job 1 decision</th>
<th>Job 2 decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profit: 39</td>
<td>Profit: 90</td>
<td>Profit: 88</td>
</tr>
<tr>
<td>Time: 1</td>
<td>Time: 1</td>
<td>Time: 2</td>
</tr>
<tr>
<td>Deadline: 1</td>
<td>Deadline: 2</td>
<td>Deadline: 2</td>
</tr>
</tbody>
</table>
**Job class**

```java
public class Job implements Comparable {
    int jobNbr; // Package access
    int deadline; // Package access
    int profit; // Package access
    int time; // Package access

    public Job(int j, int d, int p, int t) {
        jobNbr = j;
        deadline = d;
        profit = p;
        time = t;
    }

    public int compareTo(Object other) {
        Job o = (Job) other;
        if (deadline < o.deadline)
            return -1;
        else if (deadline > o.deadline)
            return 1;
        else
            return 0;
    }

    public String toString() {
        return "J: " + jobNbr + " D: " + deadline + " P: " + profit + " T: " + time;
    }
}
```

**JobScheduler**

```java
public class JobScheduler {
    private Job[] jobs; // Input set of jobs to schedule
    private int nJobs; // Number of input jobs
    private int endTime; // Latest end time of job (=max resource)
    private int[] path; // List of nodes in the optimal solution
    private int jobsDone; // Output: total number of jobs
    private int totalProfit; // Output

    private int nodes; // Nodes generated in DP graph
    private int[] nodeProfit; // Profit of jobs prior to this node
    private int[] nodeTime; // Time spent on jobs prior to node
    private int[] pred; // Predecessor node with best profit
    private int stageNodes; // Difference in node numbers from
                            // one stage to next
```
JobScheduler constructor, jsd()

```java
public JobScheduler(Job[], int e) {
    jobs = j;
    endTime = e;
    nbrJobs = jobs.length;
    path = new int[nbrJobs+1];
    nodes = (nbrJobs-1)*(endTime+1)+2;
    // nodes = stages*states + source, sink
    nodeProfit = new int[nodes];
    nodeTime = new int[nodes];
    pred = new int[nodes];
    for (int i = 0; i < nodes; i++)
        pred[i] = -1;
    stageNodes = endTime+1;
}

public void jsd() {
    buildSource();
    buildCenter();
    buildSink();
    backPath();
}
```

buildSource()

```java
private void buildSource() {
    nodeProfit[0] = 0; // Source is node 0
    nodeTime[0] = 0;
    // Treat stage 0 as special case because it has only 1 node
    // If job not in solution set (0 time and profit).
    nodeProfit[1] = 0;
    nodeTime[1] = 0;
    pred[1] = 0;
    // If job feasible
    if (jobs[0].time <= jobs[0].deadline) {
        int toNode = 1 + jobs[0].time;
        nodeProfit[toNode] = jobs[0].profit;
        nodeTime[toNode] = jobs[0].time;
        pred[toNode] = 0;
    }
}
```
private void buildCenter() {
    for (int stage = 1; stage < nbrJobs - 1; stage++) {
        // Generate virtual arcs
        for (int node = (stage - 1) * stageNodes + 1; node <= stage * stageNodes; node++) {
            if (pred[node] >= 0) {
                // If job not in solution, build arc if it is on optimal sequence
                if (nodeProfit[node] >= nodeProfit[node + stageNodes]) {
                    nodeProfit[node + stageNodes] = nodeProfit[node];
                    nodeTime[node + stageNodes] = nodeTime[node];
                    pred[node + stageNodes] = node;
                }
                // If job feasible build virtual arc if it is on optimal sequence
                if (nodeTime[node] + jobs[stage].time <= jobs[stage].deadline) {
                    int nextNode = node + stageNodes + jobs[stage].time;
                    if (nodeProfit[node] + jobs[stage].profit >= nodeProfit[nextNode]) {
                        nodeProfit[nextNode] = nodeProfit[node] + jobs[stage].profit;
                        nodeTime[nextNode] = nodeTime[node] + jobs[stage].time;
                        pred[nextNode] = node;
                    }
                }
            }
        }
    }
}

private void buildSink() {
    int stage = nbrJobs - 1;
    int sinkNode = (nbrJobs - 1) * stageNodes + 1;
    for (int node = (stage - 1) * stageNodes + 1; node <= stage * stageNodes; node++) {
        if (pred[node] >= 0) {
            // Generate only single best virtual arc from previous node
            // Job feasible
            if (nodeTime[node] + jobs[stage].time <= jobs[stage].deadline) {
                // Job in solution
                if (nodeProfit[node] + jobs[stage].profit >= nodeProfit[sinkNode]) {
                    nodeProfit[sinkNode] = nodeProfit[node] + jobs[stage].profit;
                    nodeTime[sinkNode] = nodeTime[node] + jobs[stage].time;
                    pred[sinkNode] = node;
                }
            }
            // Job not in solution
            if (nodeProfit[node] >= nodeProfit[sinkNode]) {
                nodeProfit[sinkNode] = nodeProfit[node];
                nodeTime[sinkNode] = nodeTime[node];
                pred[sinkNode] = node;
            }
        }
    }
}
**backPath(), outputJobs()**

```java
private void backPath() {
    // Trace back predecessor nodes from sink to source
    path[nbrJobs] = (nbrJobs-1)*stageNodes + 1;  // Sink node
    for (int stage = nbrJobs-1; stage >= 1; stage--)  
        path[stage] = pred[path[stage+1]];
}

public void outputJobs() {
    System.out.println("Jobs done:");
    for (int stage = 0; stage < nbrJobs; stage++) {
        if (nodeProfit[path[stage]] != nodeProfit[path[stage+1]]) {
            System.out.println(jobs[stage]);
            jobsDone++;
            totalProfit += jobs[stage].profit;
        }
    }
    System.out.println("\nJobs done: " + jobsDone + " Total profit: " + totalProfit);
}
```

**main()**

```java
public static void main(String[] args) {
    Job[] jobs = new Job[7];
    jobs[0] = new Job(0, 1, 39, 1);
    jobs[1] = new Job(1, 2, 90, 1);
    jobs[2] = new Job(2, 2, 88, 2);
    jobs[3] = new Job(3, 2, 20, 1);
    jobs[4] = new Job(4, 3, 37, 3);
    jobs[5] = new Job(5, 3, 25, 2);
    jobs[6] = new Job(6, 4, 70, 1);
    int endTime = 4;
    Arrays.sort(jobs);  // In deadline order
    JobScheduler j = new JobScheduler(jobs, endTime);
    j.jsd();
    j.outputJobs();
}
```
Job scheduling DP complexity

- Complexity is minimum of:
  - $O(nM)$, where
    - $n$ is number of jobs (stages)
    - $M$ is $\min(\sum p_i, \sum t_i, d_j)$
  - $O(2^n)$
- Intuitively, if no pruning occurs and the time resource is large, the number of nodes can double at each stage (job)
  - This leads to $O(2^n)$ complexity
- If the times, deadlines or profits are constrained, many fewer nodes are generated
  - This leads to $O(nM)$ complexity

Example uses of job scheduling

- Transportation vehicle fleet maintenance
  - Many vehicles, many jobs with priority and benefit
    - Routine or scheduled maintenance
    - Accident repair
    - Upgrades
- Manufacturing facility scheduling
  - Marketing requirement for production of products with expected profit margins, deadlines and times
  - Facilities making a range of products (e.g., in China)
- Robotics: control of real-time tasks
- Taxi dispatch (with extensions)
Other dynamic programming examples

- Most resource allocation problems are solved with linear programming
  - Sophisticated solutions use integer programming now
  - DP is used with nonlinear costs or outputs, often in process industries (chemical, etc.) with continuous but complex and expensive output
  - DP for resource allocation has ‘dimensionality curse’ when there is more than one resource:
    - Have triplets of (cost, time, profit) for example, instead of pair of (cost, profit)
    - Our job scheduling DP is a nice exception

- Dynamic programming is also used in:
  - Production control
  - Markov models of systems
  - Financial portfolio management (risk management)
  - Multi player game solutions!

Reliability design

Multiple devices are used at each stage. Monitors determine which devices are functioning properly. We wish to obtain maximum reliability within a cost constraint
Reliability design formulation

• If a stage $i$ contains $m_i$ devices $D_i$:
  – Probability that all have a fault = $(1-r_i)^{m_i}$
  – Reliability of stage $y_i= 1 – (1-r_i)^{m_i}$

• We want to maximize reliability:
  – Subject to a cost constraint:
  
  $$\prod_{i=1}^{n} (1 – (1-r_i)^{m_i})$$

  
  $$\sum_{i=1}^{n} c_i m_i \leq C$$
  
  $$m_i \geq 1, \text{ and integer}$$

  – We need a more flexible representation: sets (but of a different sort that our Set class)

Reliability design example

<table>
<thead>
<tr>
<th>Device</th>
<th>D0</th>
<th>D1</th>
<th>D2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device type</td>
<td>Input buffer</td>
<td>Processor</td>
<td>Output buffer</td>
</tr>
<tr>
<td>Cost</td>
<td>$300$</td>
<td>$150$</td>
<td>$200$</td>
</tr>
<tr>
<td>Reliability</td>
<td>0.9</td>
<td>0.8</td>
<td>0.5</td>
</tr>
</tbody>
</table>

• Maximum cost= $1050
Reliability design conceptual graph

Stage:

0 1 2

<table>
<thead>
<tr>
<th>D 0 decision</th>
<th>D 1 decision</th>
<th>D 2 decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost: $300</td>
<td>Cost: $150</td>
<td>Cost: $200</td>
</tr>
<tr>
<td>Reliability: 0.9</td>
<td>Reliability: 0.8</td>
<td>Reliability: 0.5</td>
</tr>
</tbody>
</table>

Reliability design conceptual graph
Reliability design conceptual graph

Stage:

- D 0 decision
  - Cost: $300
  - Reliability: 0.9

- D 1 decision
  - Cost: $150
  - Reliability: 0.8

- D 2 decision
  - Cost: $200
  - Reliability: 0.5

Infeasible
Max = $700
Must have at least 1 D1, D2

Dominated-label correction
Infeasible
Max = $850
Must have at least 1 D1, D2

Rel Pred
### Reliability design conceptual graph

![Reliability Design Conceptual Graph](image)

**Infeasible**
- Max = $700
- Must have at least 1 D1, D2

**Dominated**
- Max = $850
- Must have at least 1 D1, D2

**Stage:**
- D 0 decision
  - Cost: $300
  - Reliability: 0.9
- D 1 decision
  - Cost: $150
  - Reliability: 0.8
- D 2 decision
  - Cost: $200
  - Reliability: 0.5

**Rel Pred**

<table>
<thead>
<tr>
<th>Stage</th>
<th>Decision</th>
<th>Cost</th>
<th>Reliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>D 0</td>
<td>$300</td>
<td>0.9</td>
</tr>
<tr>
<td>1</td>
<td>D 1</td>
<td>$150</td>
<td>0.8</td>
</tr>
<tr>
<td>2</td>
<td>D 2</td>
<td>$200</td>
<td>0.5</td>
</tr>
</tbody>
</table>

### Reliability DP needs different data structure

- **Explosion of number of nodes**
- **Dominance is a more general concept than label correction**
  - Nodes with lower reliability but higher cost are pruned
  - This is necessary to prune most dynamic programming graphs
    - Heuristics are usually used
- **Asymptotic analysis is not very helpful any more**
  - Most problems are $O(\min(\text{graph size}, 2^n))$
  - Approaches with similar worst cases can have very different actual running times. Must experiment.
- **Next time we'll cover the set implementation for dynamic programming**
  - We get rid of the nodes as well as the graph!
Things to notice in this formulation

• Sorting
  – We didn’t sort in the example, but in real problem it’s always
    worth doing
  – Almost always sort by benefit/cost ratio to get dominance
    • In this problem, sort by failure probability/cost
    • Having redundancy in cheap components with high failure rate is
      likely to be the most effective strategy
  – Sorting replaces many ad-hoc heuristics, gives same effect
• There is no sink node
  – There are tricks to avoid solving the last stage—see text
• Heuristics
  – Prune at each stage based on benefit/cost ratio. Eliminate the states with
    small improvements over the preceding state
  – Load ‘obvious’ solution elements into the source node via heuristic
  – E.g in knapsack, load first 50 of expected 100 items in profit/weight order
  – If you need to do these things, branch and bound is better approach
1.204 Computer Algorithms in Systems Engineering
Spring 2010

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.