1.204 Lecture 14

Dynamic programming:
Job scheduling

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

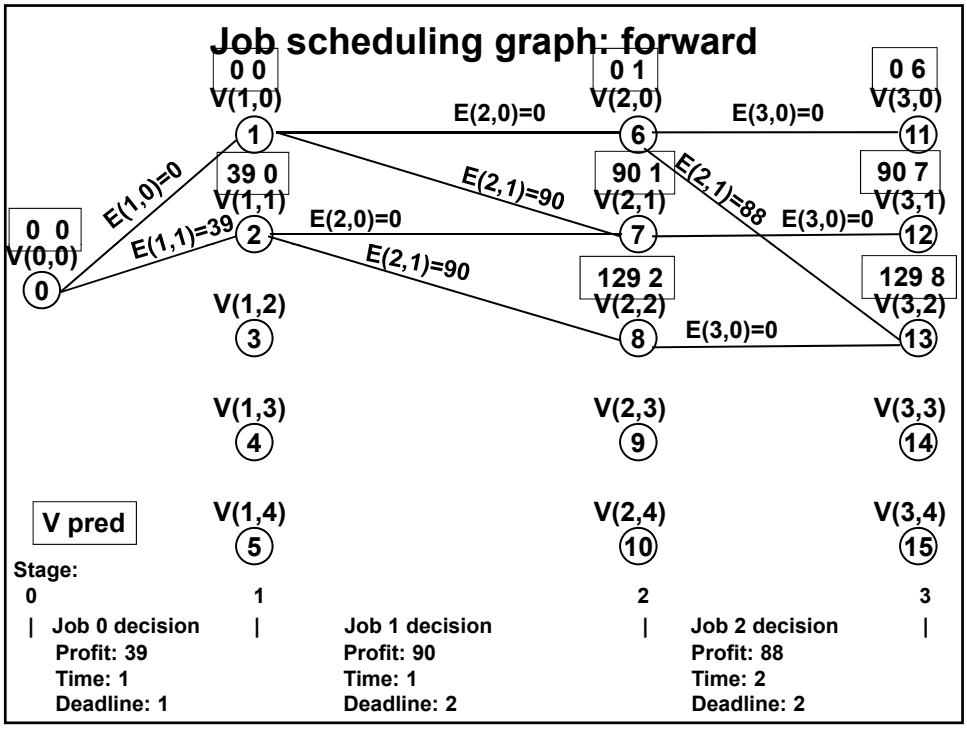
Job scheduling algorithm

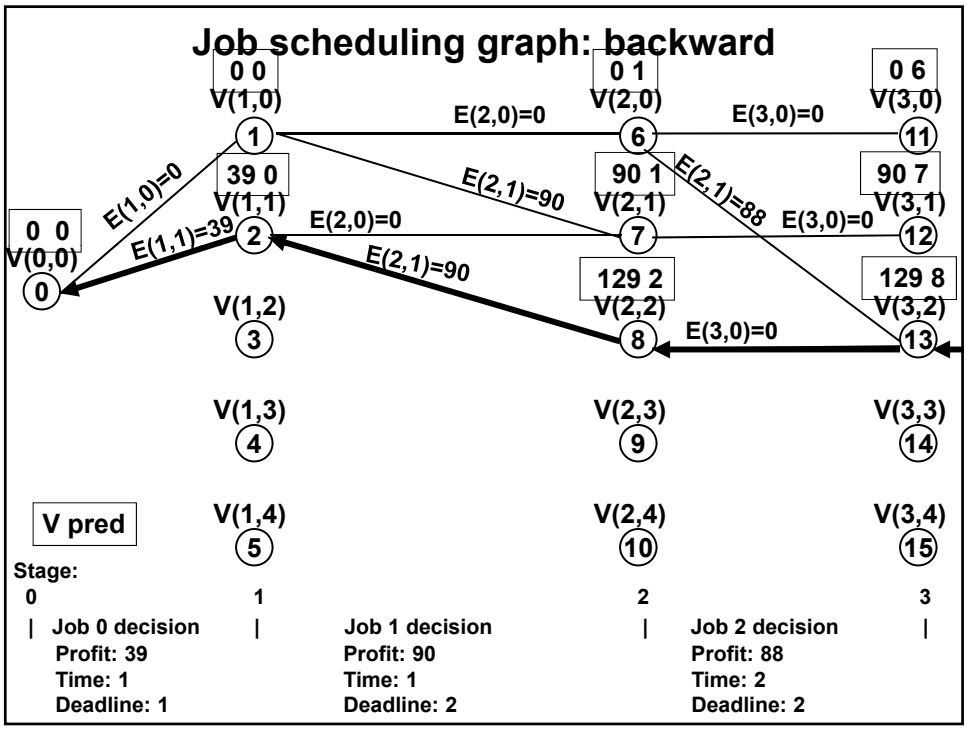
- We will label every node in the graph that we encounter with its profit and time used
  - If we find a better path to that node, we update its profit and time labels
  - This is exactly the same as the shortest path label correcting algorithm
    - We know this algorithm runs fast
  - The issue is 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

Job	Deadline	Profit	Time
0	1	39	1
1	2	90	1
2	2	88	2
3	2	20	1
4	3	37	3
5	3	25	2
6	4	70	1

4 time units of machine time available.


Job scheduling graph: forward
0 0
V(1,0)
0 1
V(2,0)
0 6
V(3,0)
E(2,0)=0
E(3,0)=0
E(1,0)=0
39 0
V(1,1)
E(2,1)=90
90 1
V(2,1)
E(2,1)=88
90 7
V(3,1)
0 0
V(0,0)
E(1,1)=39
E(2,0)=0
E(3,0)=0
E(2,1)=90
129 2
V(2,2)
129 8
V(3,2)
V(1,2)
E(3,0)=0
V(1,3)
V(2,3)
V(3,3)
V(1,4)
V(2,4)
V(3,4)
V pred
Stage:
0
1
2
3
Job 0 decision
Profit: 39
Time: 1
Deadline: 1
Job 1 decision
Profit: 90
Time: 1
Deadline: 2
Job 2 decision
Profit: 88
Time: 2
Deadline: 2


Job scheduling graph: backward
0 0
V(1,0)
0 1
V(2,0)
0 6
V(3,0)
E(2,0)=0
E(3,0)=0
E(1,0)=0
39 0
V(1,1)
E(2,1)=90
90 1
V(2,1)
E(2,1)=88
90 7
V(3,1)
0 0
V(0,0)
E(1,1)=39
E(2,0)=0
E(3,0)=0
E(2,1)=90
129 2
V(2,2)
129 8
V(3,2)
V(1,2)
E(3,0)=0
V(1,3)
V(2,3)
V(3,3)
V(1,4)
V(2,4)
V(3,4)
V pred
Stage:
0
1
2
3
Job 0 decision
Profit: 39
Time: 1
Deadline: 1
Job 1 decision
Profit: 90
Time: 1
Deadline: 2
Job 2 decision
Profit: 88
Time: 2
Deadline: 2

Job class

```
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

```
public class JobScheduler {
   private Job[] jobs;   // Input set of jobs to schedule
   private int nbrJobs;  // 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()

```
public JobScheduler(Job[] j, 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()

```
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;
     }
}
```

buildCenter()

```
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;
          }
        }
    }
} }
```

buildSink()

```
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()

```
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()

```
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_i)
  - $O(2^n)$
- 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

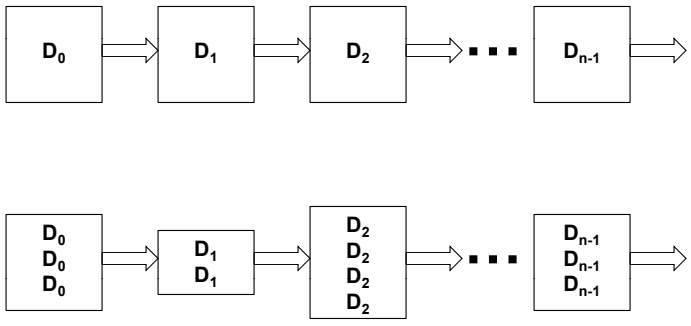
Example uses of job scheduling

- 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 = $(1-r_i)^{mi}$
  - Reliability of stage $y_i = 1 - (1-r_i)^{mi}$
- We want to maximize reliability= $\prod_n (1-(1-r_i)^{m_i})$
  - Subject to a cost constraint:

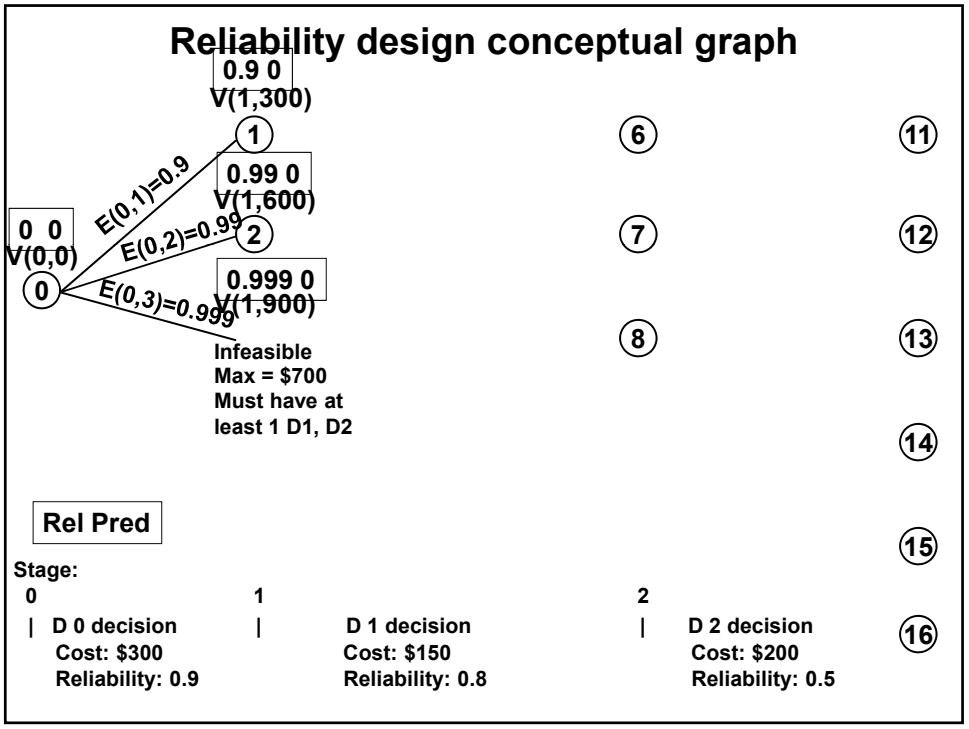
$$\sum_n c_i m_i \le C$$
$$m_i \ge 1, \text{ and integer}$$

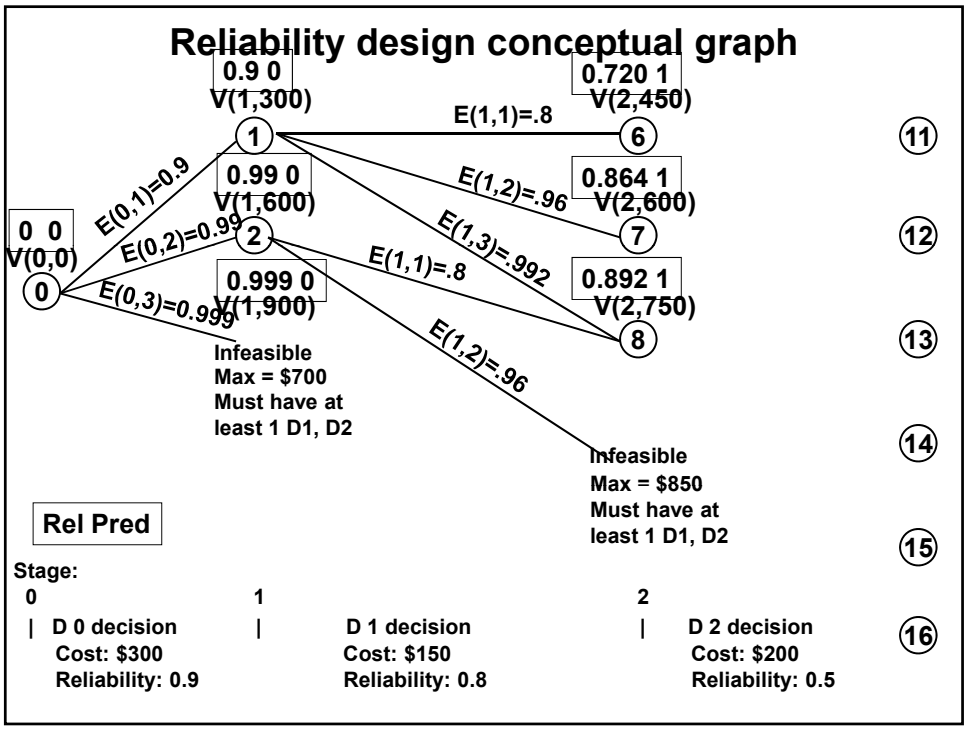
  - We need a more flexible representation: sets (but of a different sort that our Set class)

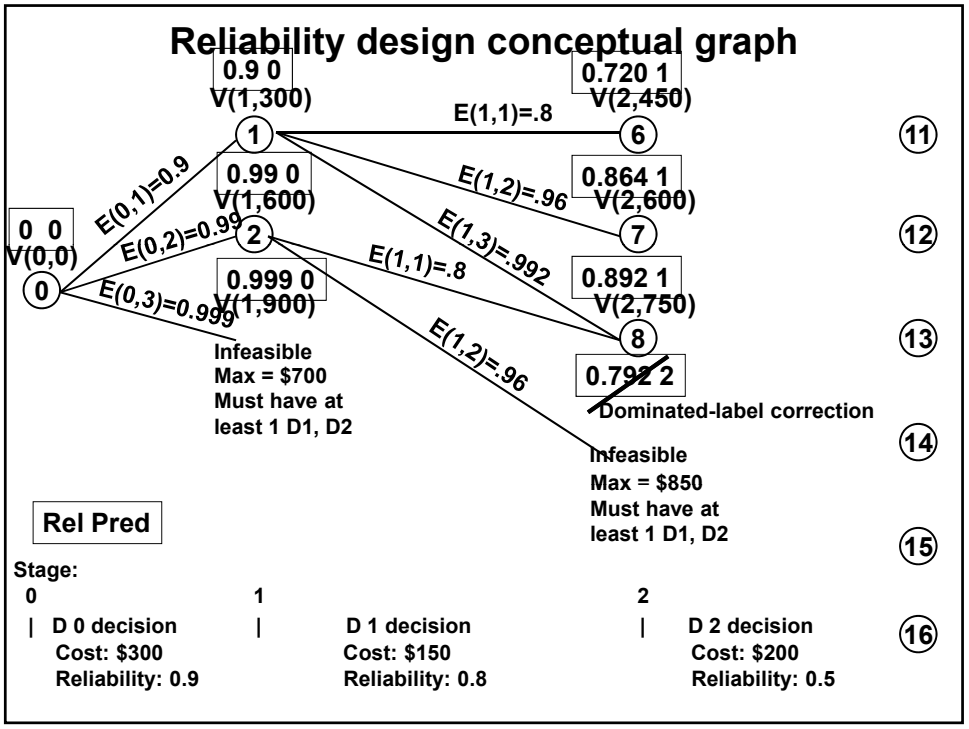
Reliability design example

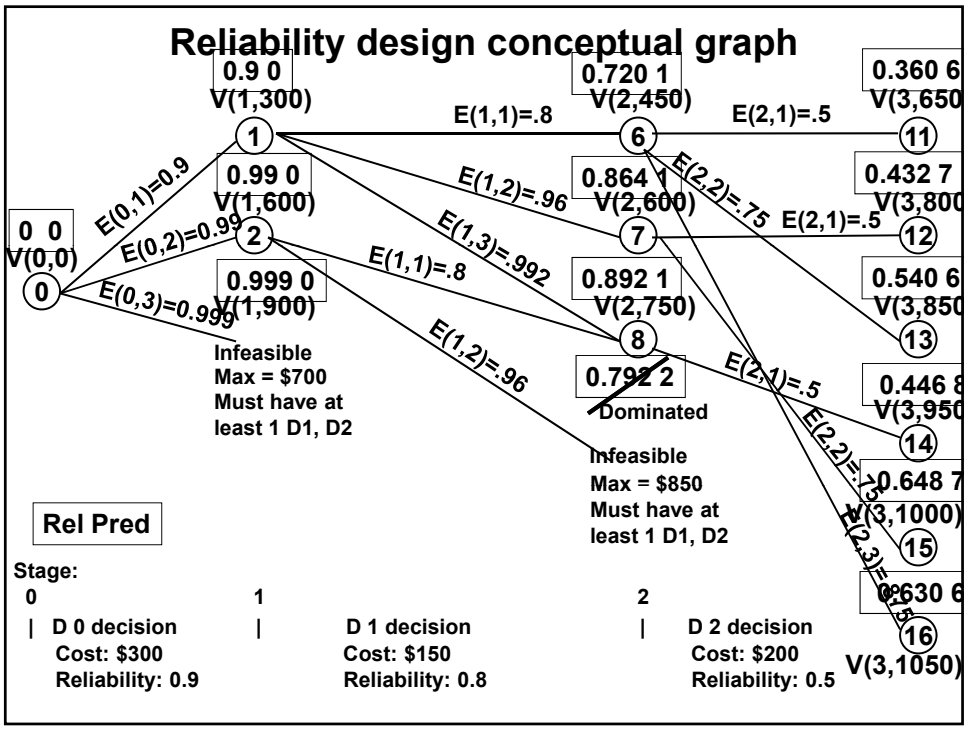
Device	D0	D1	D2
Device type	Input buffer	Processor	Output buffer
Cost	$300	$150	$200
Reliability	0.9	0.8	0.5

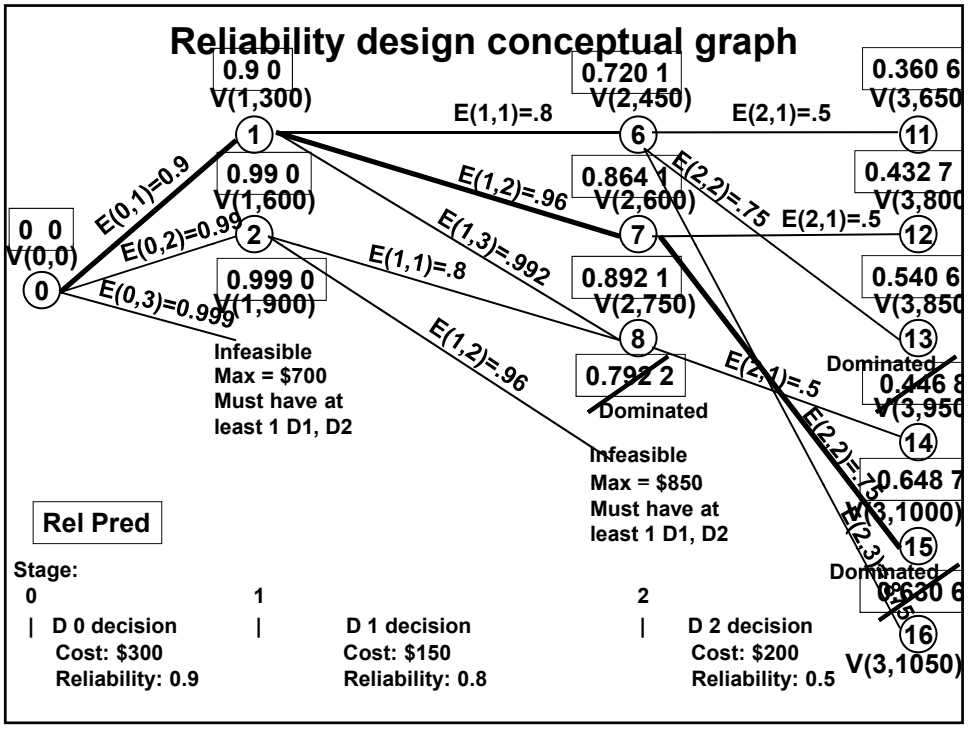
- Maximum cost= $1050


Reliability design conceptual graph
0.9 0
V(1,300)
1
6
11
0.99 0
V(1,600)
E(0,1)=0.9
0 0
V(0,0)
E(0,2)=0.99
2
7
12
0.999 0
V(1,900)
0
E(0,3)=0.999
8
13
Infeasible
Max = $700
Must have at
least 1 D1, D2
14
Rel Pred
15
Stage:
0
1
2
| D 0 decision
Cost: $300
Reliability: 0.9
| D 1 decision
Cost: $150
Reliability: 0.8
| D 2 decision
Cost: $200
Reliability: 0.5
16


Reliability design conceptual graph
0.9 0
V(1,300)
0.720 1
V(2,450)
E(1,1)=.8
1
6
11
E(0,1)=0.9
0.99 0
V(1,600)
E(1,2)=.96
0.864 1
V(2,600)
0 0
V(0,0)
E(0,2)=0.99
2
E(1,3)=.992
7
12
E(1,1)=.8
0.999 0
V(1,900)
0.892 1
V(2,750)
0
E(0,3)=0.999
8
13
E(1,2)=.96
Infeasible
Max = $700
Must have at
least 1 D1, D2
14
Infeasible
Max = $850
Must have at
least 1 D1, D2
Rel Pred
15
Stage:
0
1
2
| D 0 decision
Cost: $300
Reliability: 0.9
| D 1 decision
Cost: $150
Reliability: 0.8
| D 2 decision
Cost: $200
Reliability: 0.5
16


Reliability design conceptual graph
0.9 0
V(1,300)
0.720 1
V(2,450)
E(1,1)=.8
1
6
11
E(0,1)=0.9
0.99 0
V(1,600)
E(1,2)=.96
0.864 1
V(2,600)
0 0
V(0,0)
E(0,2)=0.99
2
7
12
E(1,3)=.992
E(1,1)=.8
0.999 0
V(1,900)
0.892 1
V(2,750)
0
E(0,3)=0.999
8
13
E(1,2)=.96
0.792 2
Dominated-label correction
14
Infeasible
Max = $700
Must have at
least 1 D1, D2
Infeasible
Max = $850
Must have at
least 1 D1, D2
Rel Pred
15
Stage:
0
1
2
16
| D 0 decision
Cost: $300
Reliability: 0.9
| D 1 decision
Cost: $150
Reliability: 0.8
| D 2 decision
Cost: $200
Reliability: 0.5


Reliability design conceptual graph
0.9 0
V(1,300)
0.720 1
V(2,450)
0.360 6
V(3,650
E(1,1)=.8
E(2,1)=.5
1
6
11
E(0,1)=0.9
0.99 0
V(1,600)
E(1,2)=.96
0.864 1
V(2,600)
E(2,2)=.75
0.432 7
V(3,800
0 0
V(0,0)
E(0,2)=0.99
2
7
E(2,1)=.5
12
E(1,3)=.992
E(1,1)=.8
0.999 0
V(1,900)
0.892 1
V(2,750)
0.540 6
V(3,850
0
E(0,3)=0.999
8
13
E(1,2)=.96
0.792 2
Dominated
E(2,1)=.5
0.446 8
V(3,950
Infeasible
Max = $700
Must have at
least 1 D1, D2
14
E(2,2)=.5
0.648 7
V(3,1000)
Infeasible
Max = $850
Must have at
least 1 D1, D2
15
E(2,3)=.5
0.630 6
Rel Pred
16
V(3,1050)
Stage:
0
1
2
| D 0 decision
Cost: $300
Reliability: 0.9
| D 1 decision
Cost: $150
Reliability: 0.8
| D 2 decision
Cost: $200
Reliability: 0.5



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(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