1.204 Lecture 15

Dynamic programming:
Knapsack

When multistage graphs don’t work

• If the resource has many levels:
  – Large range of ints
  – Floating point number
• Then the multistage graph can’t be constructed
  – And label correction is not a sufficient implementation for pruning
• We need a set representation instead
  – Different than our Set data structure, alas
• We keep all the elements in the solution at any stage in a set
  – We purge dominated elements
  – In a knapsack problem, for example, we purge any element whose weight is same or higher and its profit is same or lower than another element
  – This is how we implement pruning
• We still need to structure the problem so that feasibility constraints keep the size of the sets low
Knapsack problem

- Problem is modeled as a series of decisions on whether to include item 1, item 2, item 3, ...
  - Each item has a profit (benefit) and a weight (cost)
  - The knapsack has a maximum weight (cost)
  - Each project is either in or out of the knapsack
    • No fractional values allowed, as were in the greedy version
- Algorithm
  - Forward pass: builds sets instead of graph
    • Sets contain cumulative (profit, weight) pairs
  - Backward pass: traces sets back from sink to source to recover solution
  - Algorithm can produce solution for all weights less than or equal to maximum weight in a single run

First example

<table>
<thead>
<tr>
<th>Item</th>
<th>Profit</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

- Maximum weight= 9
- Item 0 is a sentinel with 0 weight, 0 profit always
Forward pass: build sets

- $S(0) = (0,0)$  
  S holds cumulative profit, weight  
- $S' = (1,2)$  
  S' is set of items to merge  
- $S(1) = (0,0) (1,2)$  
  S(n) is merged $S(n-1)$ and S'  
- $S' = (2,3) (3,5)$  
- $S(2) = (0,0) (1,2) (2,3) (3,5)$  
- $S' = (5,4) (6,6) (7,7) (8,9)$  
  - Note that (3,5) is purged when $S(3)$ is constructed  
  - It is dominated by (5,4): higher profit, lower weight  
- $S(3) = (0,0) (1,2) (2,3) (5,4) (6,6) (7,7) (8,9)$  
  - If maximum weight were 7, (8,9) pair would not be built  
    - Infeasibility

Backward pass: get solution

- $S(0) = (0,0)$  
- $S(1) = (0,0) (1,2)$  
- $S(2) = (0,0) (1,2) (2,3) (3,5)$  
- $S(3) = (0,0) (1,2) (2,3) (5,4) (6,6) (7,7) (8,9)$  
  - Maximum weight: 4 5 6 7 8 9  
    - Last pair is optimal (profit, weight) for entire problem  
- If pair exists in previous set, item not in solution  
- If pair not in previous set, item is in solution  
  - Subtract item profit, weight and find that pair in previous set  
  - Continue to trace back to source node
Second example

<table>
<thead>
<tr>
<th>Item</th>
<th>Profit</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>31</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>33</td>
<td>23</td>
</tr>
<tr>
<td>5</td>
<td>43</td>
<td>33</td>
</tr>
<tr>
<td>6</td>
<td>53</td>
<td>43</td>
</tr>
<tr>
<td>7</td>
<td>55</td>
<td>45</td>
</tr>
<tr>
<td>8</td>
<td>65</td>
<td>55</td>
</tr>
</tbody>
</table>

- Maximum weight 110
- Item 0 is sentinel

Forward pass: build sets

Traceback uses same logic as before
Algorithm implementation

• Follows examples, but there are complications:
  – We must keep each set S(i) to trace back the answer
    • In example 1, if we kept only the final set S(3), the pair (3,5) would have been purged and we would not be able to trace back the solution
    • Pairs dominated by pairs considered later can still be part of an optimal subsequence in the optimal solution
    • Storage requirements for all the sets are significant
      – We discard S’ at each step
  – The sets are of varying and difficult-to-predict length
    • We use Java ArrayLists
      – O(1) add() method, which is all we use
      – Allow flexible number of pairs to be stored
    – The dominance operation is difficult to code
    – A sentinel, item 0, with 0 profit and 0 weight is needed
      • Must be at start of input regardless of input sort order

Algorithm implementation 2

• We sort the items in descending profit/weight order, as in the greedy algorithm
  – Putting ‘good’ items into the solution early usually allows more pruning to occur
  – Our dominance operation must handle any item order
• An alternative is to sort the items in descending weight order, if many items’ weights are large relative to the knapsack maximum weight
  – This may make the sets smaller because feasibility constraints eliminate many combinations early
• It’s always good to run the greedy version first
  – If it finds an integer solution, it’s optimal
  – Even if it doesn’t, its solution will give you insights on the nature of your problem data, and an approximate solution in case your DP doesn’t terminate
Generalizing the set-based dynamic programming code

• We use ints in this implementation
  – Can handle doubles but must use TOLERANCE when computing dominance to manage numerical error

• This implementation can be modified to handle other dynamic programming problems that can’t be done with a multistage graph
  – E.g., the job scheduling dynamic program would keep a triplet (profit, time, deadline) instead of (profit, weight)
  – The dominance calculation would need to be modified to match the problem statement
    • The changes aren’t as tough as writing it the first time

```java
public class DPItem implements Comparable {
    int profit;
    int weight;

    public DPItem(int p, int w) {
        profit = p;
        weight = w;
    }

    public boolean equals(Object other) {
        DPItem o = (DPItem) other;
        if (profit == o.profit && weight == o.weight)
            return true;
        else
            return false;
    }

    public int compareTo(Object o) {
        DPItem other = (DPItem) o;
        double ratio = (double) profit/weight;
        double otherRatio = (double) other.profit/other.weight;
        if (ratio > otherRatio) // Descending sort
            return -1;
        else if (ratio < otherRatio)
            return 1;
        else
            return 0;
    }
} // toString() method not shown
```
**DPSet constructor, extend()**

```java
public class DPSet {
    ArrayList<DPItem> data; // Flexible capacity, fast add
    private static int capacity; // Maximum weight

    public DPSet() {
        data = new ArrayList<DPItem>();
    }

    public static void setCapacity(int c) {
        capacity = c;
    }

    public DPSet extend(DPItem other) { // Add item to set
        DPSet result = new DPSet();
        for (DPItem i : data) {
            int cumWgt = i.weight + other.weight;
            if (cumWgt <= capacity) {
                int cumProf = i.profit + other.profit;
                result.data.add(new DPItem(cumProf, cumWgt));
            }
        }
        return result;
    }
}
```

**DPSet merge(), p. 1**

```java
public DPSet merge(DPSet other) { // Merges DPSet other with this DPSet, with dominance pruning
    // Items in any input sort order wind up in weight order
    DPSet result = new DPSet();
    // Define limits for while loop on DPSet other
    int indexOther = 0;
    int maxIndexOther = other.data.size()-1;
    // Last item profit used for dominance check at end of set
    int lastItemProfitOther = other.data.get(maxIndexOther).profit;

    // Define limits for while loop on this DPSet
    int index = 0;
    int maxIndex = data.size()-1;
    int lastItemProfit = data.get(maxIndex).profit;

    // Continues on next slide, which compares items and other items
```
Dominance

- If item weight < other weight
  - Write item to results; it cannot be dominated
  - If other profit <= item profit, other is dominated; skip it
    - Keep looping over next other items 'til not dominated
- If item weight= other weight
  - If item profit >= other profit
    - Skip other item; it's dominated
  - Else skip item; it's dominated
  - Don't write either of them into solution yet
    - Either may be dominated by a previous pair.
    - Wait for next comparison
- If other weight < item weight
  - Same logic as first case holds

DPSet merge(), p. 2

```java
while (index <= maxIndex || indexOther <= maxIndexOther) {
  if (index <= maxIndex && indexOther <= maxIndexOther) { // Both ok
    DItem item= data.get(index);
    DItem otherItem= other.data.get(indexOther);
    if (item.weight < otherItem.weight) {
      result.data.add(item); // Add item; not dominated by other item
      index++;
      while (otherItem.profit < item.profit && indexOther < maxIndexOther) {
        otherItem= other.data.get(++indexOther); // Other dominated, skip
      }
    } else if (item.weight == otherItem.weight) {
      if (item.profit >= otherItem.profit) // Other item dominated
        indexOther++;
      else
        ++index; // Item dominated
    } else { // otherItem.weight < item.weight
      result.data.add(otherItem); // Add other item, not dominated
      indexOther++;
      while (item.profit < otherItem.profit && index < maxIndex) {
        item= data.get(+index); // Item dominated; skip it
      }
    }
  } else {
    // otherItem.weight < item.weight
    result.data.add(item); // Add item, not dominated
    index++;
  }
} // Continues on next slide, within while loop; end condition
```
DPSet merge(), p. 3

// One loop index is already at end. Handle remaining in other set

else if (index > maxLength) { // Only other items left to consider
    while (indexOther <= maxLengthOther) {
        DItem otherItem = other.data.get(indexOther);
        if (otherItem.profit > lastItemProfit)
            result.data.add(otherItem);
        indexOther++;
    }
} else { // indexOther > maxLengthOther. Only items left
    while (index <= maxLength) {
        DItem item = data.get(index);
        if (item.profit > lastItemProfitOther)
            result.data.add(item);
        index++;
    }
}
return result;

DPKnap constructor, knapsack()

public class DPKnap {
    private DItem[] items; // Input items
    private int m; // Capacity of knapsack
    private DSet[] sets; // Subsequences, sets
    private DItem[] solution; // Solution with optimal items only

    public DPKnap(DItem[] i, int maxCap) {
        items = i;
        m = maxCap;
        sets = new DSet[items.length];
        solution = new DItem[items.length];
    }

    public void knapsack() {
        buildSets();
        backPath();
        outputSolution();
    }
}
DPKnap buildSets()

```java
private void buildSets() {
    DPsSet.setCapacity(m);
    // Build set 0 with node 0
    DPSet s = new DPSet();
    // Add item 0 to set 0. Sentinel w/o profit, weight.
    s.data.add(items[0]);
    sets[0] = s;

    // For sets 1 and above
    for (int i = 1; i < sets.length; i++) {
        // Add item and find cumulative profit, weight pairs
        DPSet sNext = s.extend(items[i]);
        // Merge, with dominance, with prior set
        s = s.merge(sNext);
        // Store new set: needed to trace back solution
        sets[i] = s;
    }
}
```

DPKnap backPath()

```java
private void backPath() {
    int lastSetIndex = sets.length - 1; // Start at last set
    int lastSetItem = sets[lastSetIndex].data.size() - 1;
    DpItem lastItem = sets[lastSetIndex].data.get(lastSetItem);

    int cumProfit = lastItem.profit;
    int cumWeight = lastItem.weight;
    DpItem prevItem = lastItem;

    for (int i = lastSetIndex - 1; i >= 0; i--) {
        boolean itemFound = false; // Is item in previous set
        int prevSetIndex = i + 1;
        DPSet currSet = sets[i];
        int currItemIndex = currSet.data.size() - 1;
        for (int j = currItemIndex; j >= 0; j--) {
            DpItem currItem = currSet.data.get(j);
            if (currItem.equals(prevItem)) {
                itemFound = true;
                break;
            }
            if (currItem.weight < prevItem.weight)
                break; // No need to search further
        }
    }
}
```
DPKnap backPath() 2

// Pair not found in preceding set; item is in solution
if (itemFound) {
    solution[prevSetIndex] = items[prevSetIndex];
    cumProfit -= items[prevSetIndex].profit;
    cumWeight -= items[prevSetIndex].weight;
    prevItem = new DItem(cumProfit, cumWeight);
} else keep searching for prev item in the next set
}
}

DPKnap outputSolution()

private void outputSolution() {
    int totalProfit = 0;
    int totalWeight = 0;
    System.out.println("Items in solution:");
    // Position 0 in solution is sentinel; don't output
    for (int i = 1; i < solution.length; i++)
        if (solution[i] != null) {
            System.out.println(items[i]);
            totalProfit += items[i].profit;
            totalWeight += items[i].weight;
        }
    System.out.println("Profit: " + totalProfit);
    System.out.println("Weight: " + totalWeight);
}
DPKnap main()

public static void main(String[] args) {
    // Sentinel must be in 0 position even after sort
    DItem[] list = {new DItem(0, 0),
                    new DItem(11, 1),
                    new DItem(21, 11),
                    new DItem(31, 21),
                    new DItem(33, 23),
                    new DItem(43, 33),
                    new DItem(53, 43),
                    new DItem(55, 45),
                    new DItem(65, 55)};
    Arrays.sort(list, 1, list.length); // Leave sentinel in position 0
    // Int capacity = 110;
    // Assume all item weights <= capacity. Not checked. Discard such items.
    // Assume all item profits > 0. Not checked. Discard such items.
    DPKnap knap = new DPKnap(list, capacity);
    knap.knapsack();
}

DPKnap example 2 output

Items in solution:
Profit: 11 weight: 1
Profit: 21 weight: 11
Profit: 31 weight: 21
Profit: 43 weight: 33
Profit: 53 weight: 43
Profit: 159
Weight: 109
Problem size

- How large a problem will the set-based dynamic programming approach solve?
  - It's highly data-dependent
  - If you're lucky, you may solve a problem with hundreds or even thousands of items
    - If maximum capacity is low, so feasibility check cuts out many combinations
    - If profit/weight sort or other heuristic is effective in pruning many combinations from the sets
  - If you're unlucky, the program will get to about 40 or 50 items and stall (2^{40} is a large number)
    - You may run out of storage for the sets before your computation time also becomes excessive

Dynamic programming

- Generally used on smaller 0-1 decision problems, often of size 20 to 40, or perhaps 100 items
  - Dynamic programming occasionally works on large problems
- Generally used on 'integrated problems' that don't decompose into a master problem and subproblems
  - We will study branch-and-bound methods next, which are better suited for problems that decompose
- With multistage graphs, dynamic programming is a label correcting shortest path algorithm on a graph (that we don't actually need to build)
  - One source (origin), one sink (destination)
  - Running time depends on the size of the virtual graph
- With sets, dynamic programming uses a dominance criterion
  - Not as efficient as label correction, but a graph can't be built
  - More effective pruning by comparing all states in a stage
- Keys are to use pruning/dominance and feasibility constraints to keep the graph or set sizes small
  - Efficient implementations that don't store unnecessary data or do unnecessary calculations can help significantly
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.