1.204 Lecture 16

Branch and bound:
Method, knapsack problem

Branch and bound

• Technique for solving mixed (or pure) integer programming problems, based on tree search
  – Yes/no or 0/1 decision variables, designated $x_i$
  – Problem may have continuous, usually linear, variables
  – $O(2^n)$ complexity
    • Relies on upper and lower bounds to limit the number of combinations examined while looking for a solution
    • Dominance at a distance
      – Solutions in one part of tree can dominate other parts of tree
      – DP only has local dominance: states in same stage dominate
    • Handles master/subproblem framework better than DP
    • Same problem size as dynamic programming, perhaps a little larger: data specific, a few hundred 0/1 variables
  – Branch-and-cut is a more sophisticated, related method
    • May solve problems with a few thousand 0/1 variables
    • Its code and math are complex
    • If you need branch-and-cut, use a commercial solver
• Every tree node is a problem state
  – It is generally associated with one 0-1 variable, sometimes a group
  – Other 0-1 variables are implicitly defined by the path from the root to this node
    • We sometimes store all \((x)\) at each node rather than tracing back
  – Still other 0-1 variables associated with nodes below the current node in the tree have unknown values, since the path to those nodes has not been built yet

• Tree nodes are generated dynamically as the program progresses
  – Live node is node that has been generated but not all of its children have been generated yet
  – E-node is a live node currently being explored. Its children are being generated
  – Dead node is a node either:
    • Not to be explored further or
    • All of whose children have already been explored
Managing live tree nodes

- Branch and bound keeps a list of live nodes. Four strategies are used to manage the list:
  - Depth first search: As soon as child of current E-node is generated, the child becomes the new E-node
    - Parent becomes E-node only after child’s subtree is explored
    - Horowitz and Sahni call this ‘backtracking’
  - In the other 3 strategies, the E-node remains the E-node until it is dead. Its children are managed by:
    - Breadth first search: Children are put in queue
    - D-search: Children are put on stack
    - Least cost search: Children are put on heap
  - We use bounding functions (upper and lower bounds) to kill live nodes without generating all their children
    - Somewhat analogous to pruning in dynamic programming

Knapsack problem (for the last time)

\[
\begin{align*}
\max & \quad \sum_{0 \leq i < n} p_i x_i \\
\text{s.t.} & \quad \sum_{0 \leq i < n} w_i x_i \leq M \\
& \quad x_i = 0, 1 \\
& \quad p_i \geq 0, w_i \geq 0, \ 0 \leq i < n
\end{align*}
\]

The \( x_i \) are 0-1 variables, like the DP and unlike the greedy version.
Tree for knapsack problem

Node numbers are generated but have no problem-specific meaning. We will use depth first search.

Knapsack problem tree

- Left child is always $x_i = 1$ in our formulation
  - Right child is always $x_i = 0$
- Bounding function to prune tree
  - At a live node in the tree
    - If we can estimate the upper bound (best case) profit at that node, and
    - If that upper bound is less than the profit of an actual solution found already
    - Then we don’t need to explore that node
  - We can use the greedy knapsack as our bound function:
    - It gives an upper bound, since the last item in the knapsack is usually fractional
    - Greedy algorithms are often good ways to compute upper (optimistic) bounds on problems
      - E.g., For job scheduling with varying job times, we can cut each job into equal length parts and use the greedy job scheduler to get an upper bound
    - Linear programs that treat the 0-1 variables as continuous between 0 and 1 are often another good choice
Knapsack example (same as DP)

<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, needed in branch-and-bound too

Figure by MIT OpenCourseWare. Source: Horowitz/Sahni previous edition
Knapsack solution tree

- Numbers inside a node are profit and weight at that node, based on decisions from root to that node
- Nodes without numbers inside have same values as their parent
- Numbers outside the node are upper bound calculated by greedy algorithm
  - Upper bound for every feasible left child (x=1) is same as its parent's bound
  - Chain of left children in tree is same as greedy solution at that point in the tree
  - We only recomputed the upper bound when we can't move to a feasible left child
- Final profit and final weight (lower bound) are updated at each leaf node reached by algorithm
  - Nodes A, B, C and D in previous slide
  - Solution improves at each leaf node reached
  - No further leaf nodes reached after D because lower bound (optimal value) is sufficient to prune all other tree branches before leaf is reached
- By using floor of upper bound at nodes E and F, we avoid generating the tree below either node
  - Since optimal solution must be integer, we can truncate upper bounds
  - By truncating bounds at E and F to 159, we avoid exploring E and F

KnapsackBB constructor

```java
public class KnapsackBB {
    private DItem[] items;     // Input list of items
    private int capacity;      // Max weight allowed in knapsack
    private int[] x;           // Best solution array: item i in if xi=1
    private int[] y;           // Working solution array at current tree node
    private double solutionProfit = -1;  // Profit of best solution so far
    private double currWgt;     // Weight of solution at this tree node
    private double currProfit;  // Profit of solution at this tree node
    private double newWgt;      // Weight of solution from bound() method
    private double newProfit;   // Profit of solution from bound() method
    private int k;              // Level of tree in knapsack() method
    private int partItem;       // Level of tree in bound() method

    public KnapsackBB(DItem[] i, int c) {
        items = i;
        capacity = c;
        x = new int[items.length];
        y = new int[items.length];
    }
}
```
KnapsackBB knapsack()

```java
public void knapsack() {
    int n = items.length; // Number of items in problem
    do {
        while (bound() <= solutionProfit) {
            while (k != 0) { // Back up while item k not in sack
                k--;
                if (k == 0) { // If at root, we're done. Return.
                    return;
                }
            }
            y[k] = 0; // Else take k out of soln (R branch)
            currWgt -= items[k].weight; // Reduce soln wgt by k's wgt
            currProfit -= items[k].profit; // Reduce soln profit by k's prof
        }
        k = partItem; // Set tree level k to last, possibly
        if (k == n) { // If we've reached leaf node, have
            solutionProfit = currProfit; // actual soln, not just bound
            System.arraycopy(y, 0, x, 0, y.length); // Copy soln into array x
            k = n-1; // Back up to prev tree level, which may leave solution
        } else { // Else not at leaf, just have bound
            y[k] = 0; // Take last item k out of soln
        }
    } while (true); // infinite loop till backtrack to k=0
}
```

KnapsackBB bound()

```java
private double bound() {
    boolean found = false; // Was bound found? i.e., is last item partial
    double boundVal = -1; // Value of upper bound
    int n = items.length; // Number of items in problem
    newProfit = currProfit; // Set new prof as current prof at this node
    newWgt = currWgt;
    partItem = k+1; // Go to next lower level, try to put in soln
    while (partItem < n && !found) { // More items & haven't found partial
        if (newWgt + items[partItem].weight <= capacity) { // If fits
            newWgt += items[partItem].weight; // Update new wgt, prof
            newProfit += items[partItem].profit; // by adding item wgt, prof
            y[partItem] = 1; // Update curr soln to show item k is in it
        } else { // Current item only fits partially
            boundVal = newProfit + (capacity -
                newWgt)*items[partItem].profit/items[partItem].weight;
            found = true; // Compute upper bound based on partial fit
        }
    }
    if (found) { // If we have fractional soln for last item in sack
        partItem--; // Back up to prev item, which is fully in sack
        return boundVal; // Return the upper bound
    } else { // Return newProfit; // Return profit including last item
        return newProfit;
    }
}
```
public static void main(String[] args) {
  // Sentinel - must be in 0 position even after sort
  DPItem[] list = {new DPItem(0, 0),
                   new DPItem(11, 1),
                   new DPItem(21, 11),
                   new DPItem(31, 21),
                   new DPItem(33, 23),
                   new DPItem(43, 33),
                   new DPItem(53, 43),
                   new DPItem(55, 45),
                   new DPItem(65, 55)};

  Arrays.sort(list, 1, list.length); // Leave sentinel in 0
  int capacity = 110;
  // Assume all item weights <= capacity. Not checked. Discard
  // Assume all item profits > 0. Not checked. Discard.
  KnapsackBB knap = new KnapsackBB(list, capacity);
  knap.knapsack();
  knap.outputSolution();
}

// main() almost identical to DPKnapsack.
// DPItem identical, outputSolution() almost identical to DP code

### Depth first search in branch and bound

- **Depth first search used in combination with breadth first search in many problems**
  - Common strategy is to use depth first search on nodes that have not been pruned
    - This gets to a leaf node, and a feasible solution, which is a lower bound that can be used to prune the tree in conjunction with the greedy upper bounds
      - If greedy upper bound < lower bound, prune the tree!
    - Once a node has been pruned, breadth first search is used to move to a different part of the tree
  - **Depth first search bounds tend to be very quick to compute if you move down the tree sequentially**
    - E.g. our greedy bound doesn’t need to be recomputed
    - Linear program as bounds are often quick too: few simplex pivots
Next time

- Breadth first search in branch and bound trees
- Fixed facility location problem
  - Mixed integer problem
  - Uses linear program (LP) as subproblem
  - We solve the LP with a shortest path algorithm!
- The depth first search for the knapsack problem is mostly pedagogical
  - Sometimes depth first search works well enough for your particular problem and data
  - Usually you need to be a bit more sophisticated