1.204 Lecture 17

Branch and bound: Method
Warehouse location problem

Breadth first search

- Breadth first search manages E-nodes in the branch and bound tree
  - An E node is the node currently being explored
  - In breadth first search, E-node stays live until all its children have been generated
  - The children are placed on a queue, stack or heap
- Typical strategies to select E-nodes
  - Choose node with largest upper bound (in a maximization problem), using a heap
  - Choose node likely to be optimal, even if we can't prove it's optimal immediately
    • Use problem specific, heuristic rule
    • It can be a previous optimal result to similar problem
  - Choose 'quick improvement' node based on a gradient estimate from the upper and lower bounds on a node
Branching on nodes

- Several strategies are used to decide which branch (0 or 1) to take:
  - User specified rules. Again, heuristics are used
  - Set a group of 0-1 variables to given values, not just one
    - This seems to perform better in many problems
    - Our code in this lecture does not do this
- Other strategies to improve performance:
  - If dual can be computed, it provides a lower bound
  - Bound tightening, such as truncation (like we used in the last lecture on the knapsack problem)
  - Adding linear constraints \((0 \leq x \leq 1)\) in subproblems in hopes that integer answers are obtained
  - Greedy heuristics, including dual descent and others...

Facility location problem

- e.g., Amazon
- Intel
- Tropicana
Facility location example

<table>
<thead>
<tr>
<th>Warehouse</th>
<th>Fixed cost</th>
<th>Cost to ship to customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>3 10 8 18 14</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>9 4 6 5 5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>12 6 10 4 8</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>8 6 5 12 9</td>
</tr>
</tbody>
</table>

- Set of 4 possible warehouses (0-3) to serve 5 possible customers (0-4)
- Table gives annual capital (fixed) cost of warehouse if it is built, and the annual cost of shipping to each customer via that warehouse
- Decision is whether to build ($x_i=1$) or not build ($x_i=0$) each warehouse
- Objective is to minimize fixed plus shipping costs

Computing bounds

- Lower (optimistic) bound at each node is sum of:
  - Minimum transport cost over all built or unknown warehouses
  - Fixed cost of built warehouses
- Upper (pessimistic) bound at each node is sum of
  - Minimum transport cost over all built warehouses
  - Fixed cost of built warehouses
- Pruning rules
  - If minimum (pessimistic) savings from building a warehouse are greater than its fixed cost, we build it
  - If maximum (optimistic) savings from building a warehouse are less than its fixed cost, we don’t build it
- All combinations are feasible in this problem, so there is no reduction in the size of the tree from feasibility constraints
  - We can introduce capital budget constraints in some cases

Pruning rules from Akinc, Khumwala
Computational strategy

• Start at root node
  – Apply upper and lower bound at root
  – Try to lock in or lock out some warehouses
• Then create tree node with arbitrary warehouse locked in or out
  – Apply upper and lower bound at this node
  – Try to lock in or lock out additional warehouses
  – Generate children if bounds don't prune them
    • Use stack, queue or heap to hold children
• Continue until all E-nodes have been explored
  – Output optimal solution
  – Difference between lower and upper bound decreases as algorithm continues
    • We can stop when the difference is small enough, even without an exact optimal solution

Computational example: root node

• Root node
  – All warehouses $x_i$ are unknown
  – Upper bound at root is “infinity”, by convention
  – Lower bound at root is sum of:
    • Cost of built warehouses (none) plus
    • Minimum transport cost over all built or unknown warehouses, which is all of them. Lower bound= 21
  – Use convention:
    • $x= 1$ is built warehouse
    • $x= 0$ is unknown warehouse
    • $x= -1$ is warehouse not built
  – Thus, root node solution is {0, 0, 0, 0}
Pruning rules at root

- **Minimum savings at all warehouses (if warehouse is cheapest, compare it with next cheapest):**

<table>
<thead>
<tr>
<th>Warehouse</th>
<th>Fixed cost</th>
<th>Cost to ship to customer $j$</th>
<th>Minimum savings</th>
<th>Pruning decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
<td>$f[k]$</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>3</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>9</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>12</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

- **Maximum savings at all warehouses (compare it with most expensive):**

<table>
<thead>
<tr>
<th>Warehouse</th>
<th>Fixed cost</th>
<th>Cost to ship to customer $j$</th>
<th>Maximum savings</th>
<th>Pruning decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
<td>$f[k]$</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>3</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>9</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>12</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

- **Thus we are able to prune the $x_0$ branch of the tree**

Orange is cheapest warehouse to serve customer; gray is next cheapest

---

Tree from root

$x = \{0,0,0,0\}$
- Upper bound $= \infty$
- Lower bound $= 21 + 0 = 21$

$x = \{1,0,0,0\}$
- Upper bound $= 53 + 4 = 57$
- Lower bound $= 21 + 4 = 25$

$x_0 = 1$
$x_1 = -1$
$x_2 = -1$
$x_3 = -1$

Pruned
Generate E-node

- Generate E-node to left of root:
  - Warehouse 0 is built ($x_0 = 1$ in root solution)
- Compute upper and lower bounds at E-node
  - All customers served from warehouse 0
  - Upper bound= 4 (fixed cost) + 53 (transport cost) = 57
    - Assume all customers served from warehouse 0
  - Lower bound= 4 (fixed cost) + 21 (transport cost) = 25
    - Assume customers served from built and unknown warehouses
- No further pruning is possible at this node
- Arbitrarily branch on warehouse 1. Set $x_1 = 1$

\[
\begin{align*}
\text{Generate E-node bounds} \\
\text{x= \{0,0,0\}} \\
\text{Upper bound= } \infty \\
\text{Lower bound= 21+0= 21} \\
\text{x= \{1,0,0\}} \\
\text{Upper bound= 53+4=57} \\
\text{Lower bound= 21+4=25} \\
\text{x_1=1} \\
\text{x_1=-1} \\
\text{Pruned} \\
\text{x= \{1,1,0\}} \\
\text{Upper bound= 23+10=33} \\
\text{Lower bound= 21+10=31} \\
\text{x= \{1,1,0\}} \\
\text{Upper bound= 23+10=33} \\
\text{Lower bound= 21+10=31} \\
\text{x_1=1} \\
\text{x_1=-1} \\
\text{x=-1} \\
\end{align*}
\]
Pruning rules at E-node

- Minimum savings at all warehouses:

<table>
<thead>
<tr>
<th>Warehouse</th>
<th>Fixed cost</th>
<th>Cost to ship to customer</th>
<th>Minimum savings</th>
<th>Pruning decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>3 10 18</td>
<td>14 NA</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>9 4 5</td>
<td>5 NA</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>12 6 10</td>
<td>8 1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>8 6 5</td>
<td>12 9</td>
<td></td>
</tr>
</tbody>
</table>

- Maximum savings at all warehouses:

<table>
<thead>
<tr>
<th>Warehouse</th>
<th>Fixed cost</th>
<th>Cost to ship to customer</th>
<th>Maximum savings</th>
<th>Pruning decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>3 10 18</td>
<td>14 NA</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>9 4 5</td>
<td>5 NA</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>12 6 10</td>
<td>8 1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>8 6 5</td>
<td>12 9</td>
<td></td>
</tr>
</tbody>
</table>

- Thus we are able to prune the $x_2$ and $x_3$ branches of the tree

Yellow is cheapest warehouse to serve customer; gray is next cheapest

---

**E node bounds**

$x = (0,0,0,0)$
Upper bound = $\infty$
Lower bound = $21 + 0 = 21$

$x = (1,0,0,0)$
Upper bound = $53 + 4 = 57$
Lower bound = $21 + 4 = 25$

$x = (1,1,0,0)$
Upper bound = $23 + 10 = 33$
Lower bound = $21 + 10 = 31$

$x = (1,1,-1,-1)$
Upper bound = $23 + 10 = 33$
Current best solution = $33$

We now have just one E-node left to explore
Pruning rules at E-node

- Minimum savings at all warehouses:
  
<table>
<thead>
<tr>
<th>Warehouse</th>
<th>f[k]</th>
<th>Cost to ship to customer j</th>
<th>Minimum savings decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

- This locks in warehouse 2
  - We don’t do the maximum savings calculation
  - We next compute the bounds at the new node (x2=1)

E node bounds

Pruned

x = {0, 0, 0, 0}
Upper bound = ∞
Lower bound = 21 + 0 = 21

Pruned

x = {1, 0, 0, 0}
Upper bound = 53 + 4 = 57
Lower bound = 21 + 4 = 25

Pruned

x = {1, 1, 0, 0}
Upper bound = 23 + 10 = 33
Lower bound = 21 + 10 = 31

Pruned

x = {1, 1, 1, -1}
Upper bound = 23 + 10 = 33
Current best soln = 33

Pruned

x = {1, -1, 1, 0}
Upper bound = 29 + 10 = 39
Lower bound = 26 + 10 = 36

Bounded:
lower bound > best solution
Termination

- We are done:
  - There are no more live nodes
  - All have either been pruned
    - Maximum savings < fixed cost or
    - Minimum savings > fixed cost
  - Or bounded
    - Lower bound > best solution so far
- Optimal solution is the best solution found:
  - \{1, 1, -1, -1\}
  - Cost= 33
- We examined 7 nodes in tree out of 31
  - In larger problems, we can only examine a small fraction of total nodes, since there are \(2^n\) nodes for \(n\ 0\-1\) variables

Algorithm pseudocode

```java
public boolean branchAndBound() {
    upperBound = infinity;
    eNode = root;
    initialize queue empty; // Holds children of eNode
    bound(root);
    if (root is leaf) { upperBound = cost(root); answer = {root};};
    do {
        generate left and right child at first xi=0 of eNode;
        bound(left child); // May also generate/prune nodes
        bound(right child); // May also generate/prune nodes
        for each child w of eNode {
            if (lowerBound(w) < upperBound) {
                if (w is a leaf) { upperBound = lowerBound(w); answer = {w} }
                else {
                    add w to queue;
                    if (upperBound(w) + TOLERANCE < upperBound)
                        upperBound = upperBound(w) + TOLERANCE;
                }
            }
        }
        do {
            if (queue empty) return;
            delete eNode from front of queue;
        } while (lowerBound(eNode) >= upperBound);
    } while (number of children < maximum number of children);
```
Algorithm operation

- We can change algorithm from classical breadth first search (BFS) to D-search or LC-search by substituting a stack or heap for the queue
  - LC search can use lower bound, upper bound or other criteria as priority to explore child
- We don’t store parent of node
  - We store answer array, from which we generate children
  - BFS, D-search and LC-search never backtrack, so parent is not needed
  - If you want to backtrack, then store parent
  - Our bound() methods can generate several children during their calculations
    - This makes it more convenient for us to store the answer array
- One special case is not handled in our code:
  - If all costs from warehouses to customers are equal, maximum savings from any warehouse will be zero and all warehouses will be closed at root node
    - If this occurs, we know only one warehouse needs to be open
    - Pick the cheapest.

Branch and bound example
Branch and bound example

- Shipping sugar harvest from Brazil for export
- Warehouse customers: nodes 0 through 49
  - Ship product to warehouse
  - Each customer has a quantity produced and shipped
  - Each arc in highway network has a cost
- Warehouses: nodes 50 through 57
  - Warehouses are on rail lines, ship to port by rail
  - Each warehouse has fixed cost, if built
  - No capacity constraint
- Which warehouses do we build to minimize cost?
  - What customers ship to each warehouse?
  - What are flows, costs for each customer and warehouse?

LC-search branch and bound

- LCBB.java (least-cost search) imports
  import src.dataStructuresHeap; // LC-search
  import src.greedy.Graph; // Used by all BB codes
  - Use nested class BBNode to allow LC-search to use lower bound, upper bound or answer array to select next node to search
- DBB.java (D-search) and BFSBB (breadth-first search) import
  import src.dataStructures.Stack; // For D-search only
  import src.dataStructures.Queue; // For BFS search only
  import src.greedy.Graph; // Used by all BB codes
  - Stack and queue implementations use BBNode as well, to demonstrate interchangeability of code
- BBAArray.java uses BFS and 'raw' arrays rather than a BBNode (branch and bound) nested class like the first three
  - There are extensive comments in the BBAArray.java file
  - All classes use java.io.* and java.util.*
Code outline

Graph class: constructor, shortHK()
LCBB class:
  LCBB data members: input, calculation, output
  BBNode inner class: data members, constructor, compareTo()
LCBB() constructor
  bbNetwork(): read warehouse.txt input data
branchAndBound():
  setC(): call g.shortHK() on all warehouses, create costs
  initializeBB(): cost initialization, create root of BB tree
  bb(): branch-and-bound algorithm
  bound(): compute min, max savings, lower/upper bounds
  warehouseBound(): compute min, max savings at 1 warehouse
  bbAssign(): postprocess output, assign customer to warehouse
  bbOutput(): prints out solution, costs, flows
main():
  create Graph object g
  create LCBB object w
  call w.branchAndBound()

LCBB data members

public class LCBB {
  // input data
  private int nw;  // Number of potential warehouses
  private int nc;  // Number of customers
  private int[] f;  // Fixed cost of each potential warehouse
  private int[][] c;  // Cost from customer to warehouse
  private int[] railcost;  // Cost by rail, warehouse to port
  private int[] prod;  // Production volume from each customer
  private final static int EPS= 1;  // Epsilon, tolerance
  private final static int MAXBBNODES= 10000;

  // Data used by branch and bound calculations
  private int[] savMax;  // Calculated by warehouseBound()
  private int[] savMin;  // Calculated by warehouseBound()
  private BBNode[] nodes;  // Branch and bound nodes
  private Heap h;  // Keeps nodes to be visited still
      // Stack or Queue in other versions

  // Solution
  private int[] ans;  // Solution: 1 if in, -1 if not, 0 unknown
  private int upperBound;  // Global upper bound
  private boolean optimumFound;
  int[] wAssign;  // Warehouse assgd to customer
  int[] flow;  // Flow through each warehouse
BBNode nested class

```java
private class BBNode implements Comparable {
    private int[] x; // Solution
    private int upBound; // Upper bound (cost) estimate
    private int lowBound; // Lower bound (cost) estimate

    public BBNode() {
        x = new int[nw];
    }

    // Place node with lowest lower bound at top of heap
    public int compareTo(Object other) {
        BBNode o = (BBNode) other;
        if (lowBound < o.lowBound)
            return 1;
        else if (lowBound > o.lowBound)
            return -1;
        else
            return 0;
    }
}

// Can create general rule for which node is at top
```

LCBB constructor

```java
public LCBB(String filename) {
    // Input data
    bbNetwork[filename];
    c = new int[nw+1][nc]; // Cost matrix, cust-whse
    // Last row holds max cost
    // Data used by branch and bound calculations
    savMax = new int[nw];
    savMin = new int[nw];
    nodes = new BBNode[MAXBBNODES];
    // Allocate all BBNode memory first
    for (int i = 0; i < MAXBBNODES; i++)
        nodes[i] = new BBNode();
    h = new Heap(); // Or Stack or Queue

    // Solution
    ans = new int[nw];
    upperBound = Integer.MAX_VALUE;
    whAssign = new int[nc];
    flow = new int[nw];
}
```
bbNetwork: read warehouse input file

```java
public void bbNetwork(String filename) {
    try {
        FileReader fin = new FileReader(filename);
        BufferedReader in = new BufferedReader(fin);
        nc = Integer.parseInt(in.readLine());
        nw = Integer.parseInt(in.readLine());
        f = new int[nw];
        railcost = new int[nw];
        prod = new int[nc+nw];
        for (int i=0; i < nw; i++) {
            String str = in.readLine();
            StringTokenizer t = new StringTokenizer(str, ",");
           *wNumber = (Integer.parseInt(t.nextToken()));
            railcost[i] = (Integer.parseInt(t.nextToken()));
            f[i] = (Integer.parseInt(t.nextToken()));
        }
        for (int i=0; i < nc; i++) {
            String str = in.readLine();
            StringTokenizer t = new StringTokenizer(str, ",");
            cNumber = (Integer.parseInt(t.nextToken()));
            prod[i] = (Integer.parseInt(t.nextToken()));
        }
    in.close(); ...  // Catch exception, and end method
```

setC()

```java
public void setC(Graph g) {
    int[][] DW = new int[nw+1][nc+nw];
    int[][] PW = new int[nw+1][nc+nw];
    int nodes = g.getNodes();

    for (int root = nc; root < (nc + nw); root++) {
        g.shortHK(root);
        int[] D = g.getD();
        int[] P = g.getP();
        for (int i = 0; i < nodes; i++) {
            DW[root-nc][i] = D[i];
            PW[root-nc][i] = P[i];
        }
    }
    for (int k = 0; k < nw; k++)
        for (int j = 0; j < nc; j++)
            c[k][j] = (DW[k][j] + railcost[k]) * prod[j];
```
public void initializeBB() {
    for (int m = 0; m < nc; m++) { // Write highest cost
        int temp = 0;
        for (int j = 0; j < nw; j++)
            if (c[j][m] > temp)
                temp = c[j][m];
        c[nw][m] = temp;
    }
    // bound returns true if leaf
    if (bound(0)) { // Find upper, lower bounds
        upperBound = nodes[0].lowBound;
        for (int k = 0; k < nw; k++)
            ans[k] = nodes[0].x[k];
    }
    // if all warehouses closed at root, select cheapest
    // one. This special case not handled.
}

public boolean bb() {
    BBNode eNode = nodes[0]; // Root, node 0, is the first e-node
    int i = 0; // Root is 0th node in tree
    int inOut = 1; // Toggles between -1 and +1
    do {
        // Infinite loop until queue empty
        int w = -1;
        do { w++;
        } while (!((eNode.x[w] == 0 || w >= nw));
        if (w < nw) { // if unknown warehouse found, gen children
            for (int z = 0; z <= 1; z++) {
                i++;
                // Generate child
                for (int j = 0; j < nw; j++)
                    nodes[i].x[j] = eNode.x[j]; // Copy parent's solution
                nodes[i].x[w] = -inOut; // Set unknown whse state
                boolean leaf = bound(i); // Bound this child (t if leaf)
                if (nodes[i].lowBound < upperBound) { // if worth going
                    if (leaf) { // if child is leaf, we have new optimum
                        upperBound = nodes[i].lowBound; // Update upper bound
                        for (int k = 0; k < nw; k++)
                            ans[k] = nodes[i].x[k]; // Update solution
                    } else {
                        h.insert(nodes[i]); // Add to heap
                        if (nodes[i].upBound + EPS < upperBound)
                            upperBound = nodes[i].upBound + EPS; // Update upper
                    }
                } else {
                    // Child is not leaf
                    h europé
                }
            }
        }
    }

bb(), p.2

do {
    // Find new e-node
    if (h.isEmpty()) // If heap empty, we're done
        return true; // Found optimum
    eNode = (BBNode) h.delete(); // Get e-node from heap
} while (eNode.lowBound >= upperBound);
} while (i < MAXBBNODES-2);
return false; // Generated maximum nodes w/o finding optimum

bound()

private boolean bound(int i) { // Returns true if leaf node
    boolean change = false;
    do {
        // Lock in/out warehouses based on max/min savings
        change = false;
        for (int k = 0; k < nw; k++)
            if (nodes[i].x[k] == 0)
                warehouseBound[i, k]; // Find min, max savings for k
        for (int k = 0; k < nw; k++)
            if (nodes[i].x[k] == 0) {
                if (savMin[k] <= f[k] >= 0) {
                    change = true;
                    nodes[i].x[k] = 1; // Lock in warehouse
                    for (int j = 0; j < nc; j++)
                        if (c[k][j] <= c[nw][j])
                            c[nw][j] = c[k][j];
                }
            }
    } while (change);
    return true;
}
} // Lock out warehouse
bound(), p.2

// Compute lower and upper bound. Start by adding up
// transportation costs over all customers to non-closed
// warehouses (lower bound) and to open warehouses (upper)
int lowc = 0, minc = 0, uppc = 0, maxc = 0;
for (int j = 0; j < nc; j++) {
    minc = Integer.MAX_VALUE;
    maxc = c[nw][j];
    for (int k = 0; k < nw; k++)
    if (nodes[i].x[k] != -1 && (c[k][j] < minc))
        minc = c[k][j]; // Find min transportation cost
    if (minc == Integer.MAX_VALUE)
        minc = 0;
    lowc += minc;
    uppc += maxc;
}

bound(), p.3

// Add fixed costs of open warehouses to lower and upper
// bounds, fixed cost of unknown warehouses to upper bound
boolean leaf = true;
for (int k = 0; k < nw; k++) {
    if (nodes[i].x[k] == 1) {
        lowc += f[k];
        uppc += f[k];
    }
    if (nodes[i].x[k] == 0) {
        leaf = false;
        uppc += f[k];
    }
}
nodes[i].lowBound = lowc;
nodes[i].upBound = uppc;
return leaf;
warehouseBound()

private void warehouseBound(int i, int wh) {
    // Find minimum and maximum savings for a warehouse
    // i = current node, wh = warehouse being examined
    int minSav = 0;
    int maxSav = 0;
    for (int h = 0; h < nc; h++) {  // Loop thru each customer
        minSav = Integer.MAX_VALUE;
        if (c[wh][h] < c[nw][h])
            maxSav = c[nw][h] - c[wh][h];
        else
            maxSav = 0;
        for (int g = 0; g < nw; g++)  // Loop thru each warehouse
            if (((g != wh) && (nodes[i].x[g] != -1) &&
                ((c[g][h] - c[wh][h]) < minSav))
                minSav = c[g][h] - c[wh][h];
        if (minSav == Integer.MAX_VALUE || minSav < 0)
            minSav = 0;
        savMin[wh] += minSav;
        savMax[wh] += maxSav;
    }
}

bbAssign()

// Output method, after solution is computed
public void bbAssign() {
    for (int k = 0; k < nc; k++) {
        int temp = Integer.MAX_VALUE;
        for (int j = 0; j < nw; j++) {
            if (c[j][k] < temp && ans[j] == 1) {
                temp = c[j][k];
                whAssign[k] = j;
            }
        }
    }
    for (int k = 0; k < nc; k++)
        flow[whAssign[k]] += prod[k];
bbOutput()

public void bbOutput() {    // More output
    System.out.println("Optimum found? + optimumFound");
    if (!optimumFound) {       // This code only lightly tested
        System.out.println("Upper bound: + upperBound");
        // Go through all E nodes in heap to find lowest lower bound
        int lowerBound=Integer.MAX_VALUE;
        while (!h.isEmpty()) {
            BBNode n=(BBNode) h.delete();
            if (n.lowBound < lowerBound)
                lowerBound=n.lowBound;
        }
        System.out.println("Lower bound: + lowerBound");
    }
    // If no leaf node visited yet, answer array will be all zeros.
    // Can insert code here to set ans array= x array of node with
    // continues on next slide

bbOutput(), p.2

int constr=0;
System.out.println("\nCenter \tConstruct? \tFixed Cost");
for (int j=0; j<nw; j++) {
    System.out.println(j+"\t\t+ ans[j]+"\t\t+ f[j]);
    if (ans[j] == 1)
        constr += f[j];
}
int trans=upperBound - constr;
System.out.println("\nTransport cost: + trans +
* fixed cost: + constr");
System.out.println("\nFlow through consolidation centers");
System.out.println("\nCenter\tTons");
for (int j=0; j<nw; j++)
    System.out.println(j+"\t+ flow[j]");
for (int j=0; j<nw; j++)
    if (ans[j] == 1) {
        System.out.println("\nAreas that ship to center +j");
        for (int k=0; k<nc; k++)
            if (whAssign[k] == j)
                System.out.print(" + k");
    }
}
```java
public void branchAndBound(Graph g) {
    setC(g);
    initializeBB();
    optimumFound = bb();
    bbAssign();
    bbOutput();
}

public static void main(String[] args) {
    Graph g = new Graph("src/bb/graph.txt");
    LCBB w = new LCBB("src/bb/warehouse.txt");
    w.branchAndBound(g);
}
```