Branch and Bound

In backtracking, we used depth-first search with pruning to traverse the (virtual) state space. We can achieve better performance for many problems using a breadth-first search with pruning. This approach is known as branch-and-bound.
In breadth-first search, a queue is used as an auxiliary data structure.

The advantage of the breadth-first approach is that a node that was judged "promising" when it was placed in the queue may no longer be "promising" when it is removed. If it is no longer "promising", it is discarded and the testing of its children is avoided. We illustrate the branch and bound approach with two examples. In the first example we reexamine the instance of the 0-1 knapsack problem and modify the backtracking algorithm used previously to perform a breath-first search. In the second example we use a ***best-first*** traversal of the state space to find a solution to an instance of the traveling salesperson problem (TSP). In the best-first approach we use a priority queue to hold the nodes of state space. (We can convert the 0-1 knapsack algorithm from a breadth-first to a best-first traversal simply be replacing the queue with a binary heap (priority queue).)

The previous 0-1 knapsack problem is restated below.

The 0-1 Knapsack

Consider of size **K** and we want to select from a set of ***n*** objects , where the i^{th} object has size s_i and value v_i, a subset of these objects to maximize the value contained in the knapsack with the contents of the knapsack less than or equal to K.

Suppose that K = 16 and n = 4, and we have the following set of objects ordered by their value density.

i	v_i	s_i	v_i/s_i
1	$45	3	$15
2	$30	5	$ 6
3	$45	9	$ 5
4	$10	5	$ 2

We will construct the state space where each node contains the total current value in the knapsack, the total current size of the contents of the knapsack, and maximum potential value that the knapsack can hold. In the algorithm, we will also keep a record of the maximum value of any node (partially or completely filled knapsack) found so far. When we perform the depth-first traversal of the state-space tree, a node is "promising" if its maximum potential value is greater than this current best value.

We begin the state space tree with the root consisting of the empty knapsack. The current weight and value are obviously 0. To find the maximum potential value we treat the problem as if it were the fractional knapsack problem and we were using the greedy algorithmic solution to that problem. We have shown that the greedy approach to the fractional knapsack problem yields an optimal solution. We place each of the remaining objects, in turn, into the knapsack until the next selected object is too big to fit into the knapsack. We then use the fractional amount of that object that could be placed in the

knapsack to determine the maximum potential value.

totalSize = currentSize + size of remaining objects that can be fully placed

bound (maximum potential value) = currentValue + value of remaining objects fully placed +
(K - totalSize) * (value density of item that is partially placed)

In general, for a node at level i in the state space tree (the first i items have been considered for selection) and for the k^{th} object as the one that will not completely fit into the remaining space in the knapsack, these formulae can be written:

$$\text{totalSize} = \text{currentSize} + \sum_{j=i+1}^{k-1} s_j$$

$$\text{bound} = \text{currentValue} + \sum_{j=i+1}^{k-1} v_j + (K - \text{totalSize}) * (v_k/s_k)$$

For the root node, currentSize = 0, currentValue = 0

totalSize = $0 + s_1 + s_2 = 0 + 3 + 5 = 8$

bound = $0 + v_1 + v_2 + (K - \text{totalSize}) * (v_3/s_3) = 0 + \$45 + \$30 + (16 - 8) * (\$5) = \$75 + \$40 = \$115$

The computation of the bound and the selection criteria for promising nodes is the same as before. We must replace the depth-first traversal of the state space tree with a breadth first traversal. In the depth-first traversal the auxiliary data structure used to store Nodes was (implicitly) the stack. In breath-first traversal, the auxiliary data structure is explicitly the queue.

```
Algorithm breadthFirstSearch {
    Queue q = new Queue( );
    //prepare the first Node and place it in the queue
    Node n = new Node( parameters );
    q.enqueue(n);
    while (!q.isEmpty( ) ) {
         Node temp = (Node) q.dequeue( );
        visit (temp);      //do whatever you are doing a bfs for
         for ( all children of temp )
             Node w = new Node ( parameters );
             q.enqueue(w);
         }
    }
}
```

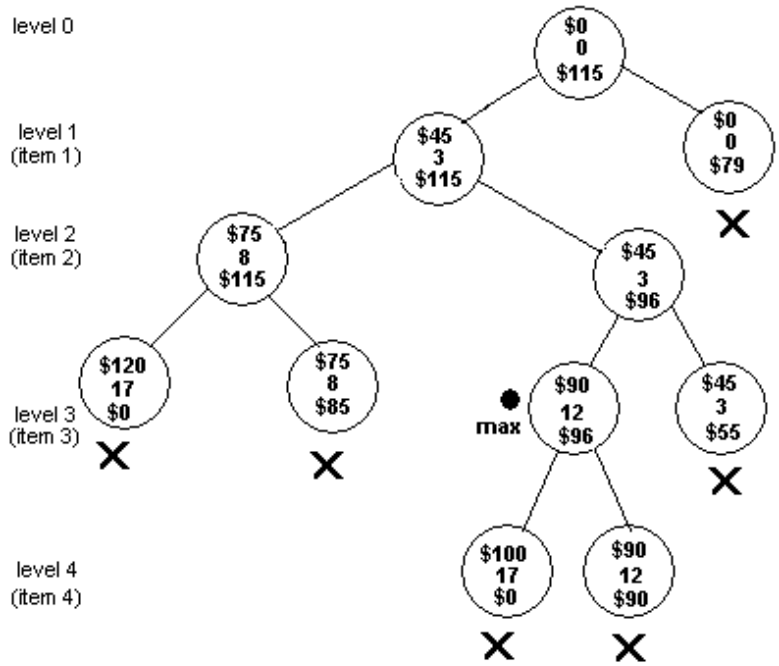
Notice, recursion (stacking) is replaced by an iterative traversal over a queue. We adapt this breath-first search algorithm to the traversal of the state space tree in the 0-1 knapsack problem.

```
import java.util.*;  //LinkedList
public class KnapsackBandB {
    private double maxValue;
    private double K;           //knapsack capacity
    private double [ ] s;        //array of sizes
    private double [ ] v;        //array of values (both ordered by value density)
    private Vector bestList;     //members of solution set for current best value
    private int numItems;         //number of items in set to select from
                                 //(first item is dummy 0)
    private Queue q;

    class Node {
        int level;
        double size, value, bound;
        List contains;
        protected Node( ) {
            level = 0;
            size = 0.0;
            value = 0.0;
            bound = 0.0;
            contains = null;
        }
        protected void copyList (Vector v) {
            if ( v == null || v.isEmpty( ) )
                contains = new Vector( );
            else
                contains = new Vector(v);
        }
        protected void add (int index) {
            contains.add(new Integer(index) );
        }
    }

    public KnapsackBandB (double capacity, double [ ] size, double [ ] value,
        int num) {
        maxValue = 0.0;
        K = capacity;
        s = size;
        v = value;
        numItems = num;
        bestList = null;
        q = new Queue( );
    }
    private void knapsack( ) {
```

```
        while (!q.isEmpty( ) ) {
            Node temp = (Node) q.dequeue( );
            if (temp.bound > maxValue) {
                Node u = new Node( );
                u.level = temp.level + 1;
                u.size = temp.size + s[temp.level+1];
                u.value = temp.value + v[temp.level+1];
                u.copyList(temp.contains);
                u.add(temp.level+1);
                if (u.size < K && u.value > maxValue)  {
                    maxValue = u.value;
                    bestList = new Vector (u.contains);
                }
                u.bound = bound(u.level, u.size, u.value);
                if (u.bound > maxValue)
                    q.enqueue(u);
                Node w = new Node( );
                w.level = temp.level + 1;
                w.size = temp.size;
                w.value = temp.value;
                w.copyList(temp.contains);
                w.add(temp.level+1);
                w.bound = bound(w.level, w.size, w.value);
                if (w.bound > maxValue)
                    q.enqueue(w);
            }
        }
    }
    private double bound (int item, double size, double value) {
        double bound = value;
        double totalSize = size;
        int k = item + 1;
        if (size > K)  return 0.0;
        while (k < numItems && totalSize + s[k] <= K) {
            bound += v[k];
            totalSize += s[k];
            k++;
        }
        if (k < numItems)
            bound += (K - totalSize) * (v[k]/s[k]);
        return bound > maxValue;
    }
    public void findSolution( ) {
        Node root = new Node( );
        root.level = 0;
        root.size = 0.0;
        root.value = 0.0;
```

```
            root.bounds = bounds(0, 0.0, 0.0);
            root.copyList(null);
            q.enqueue(root);
            knapsack ( );
            System.out.print("The solution set is:  ");
            for (int i = 0; i < bestList.size( ); i++) {
                System.out.print("  " + bestList.get(i) );
            System.out.println( );
            System.out.println("The value contained in the knapsack is:  $" +
                                maxValue);
        }
    }
```

For the example problem this algorithm traverses the following state space:



Traveling Salesperson Problem

Instead of using a Queue to perform a breadth-first traversal of the state space, we will use a PriorityQueue and perform a "best-first" traversal. For the TSP we first compute the minimum possible tour by finding the minimum edge exiting each vertex. The sum of these edges may not (most likely don't) form a possible tour, but since every vertex must be visited once and only once, every vertex must be exited once. Therefore, no tour can be shorter than the sum of these minimum edges.

At each subsequent node, the lower bound for a "tour in progress" is the length of the tour to that point plus the sum of the minimum edge exiting the end vertex of the partial tour and each of the minimum

edges leaving all of the remaining unvisited vertices. If this bound is less than the current minimum tour, the node is "promising" and the node is added to the queue. Initially the minTour is set to infinity. When a node whose path includes all of the vertices except one is reviewed (at level N - 2), there is only one possible way for the tour to complete. The remaining vertex and the first are added to the path and the length of the tour is the current length plus the length of the edge to the remaining vertex and the length of the edge from there back to the starting vertex. If this tour length is better than the current minimum, it becomes the minimum tour length. Once a first complete tour is discovered, nodes whose bound is greater than or equal to this minTour are deemed "non-promising" and are pruned.

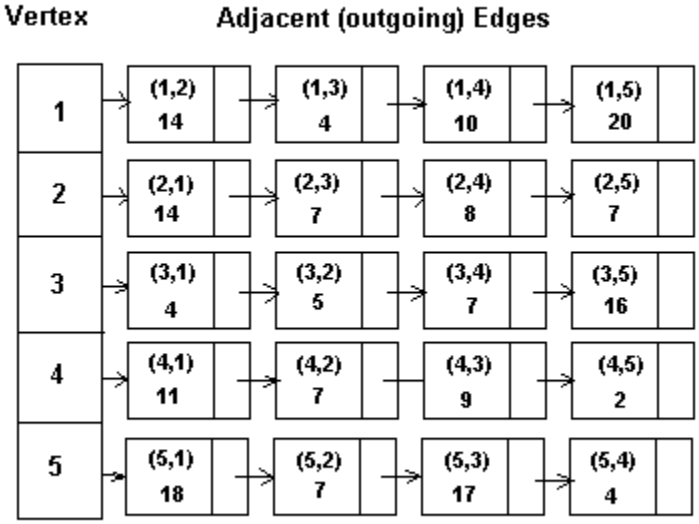
The nodes in state space must carry the following information:

- their level in the state space tree
- the length of the partial tour
- the path of the partial tour
- the bound
- (for efficiency) the last vertex in the partial tour

In a branch and bound algorithm, a node is judged to be promising before it is placed in the queue and tested again after it is removed from the queue. If a lower minTour is discovered during the time a node is in the queue, it may no longer be promising after it is removed, and it is discarded. Using a Priority Queue, the search traverses the state space tree in neither a breadth-first nor depth-first fashion, but alternates between the two approaches in a greedy, opportunistic fashion. In the example problem below, a diagram of the best-first traversal of the state space indicates by number when each of the nodes is removed from the priority queue.

Example

Let G be a fully connected directed graph containing five vertices that is represented by the following adjacency list:



We assume in the implementation of this algorithm that vertices are labeled by an integer number and edges contain the source and sink vertices and a cost or length label. The tour will start at vertex 1, and

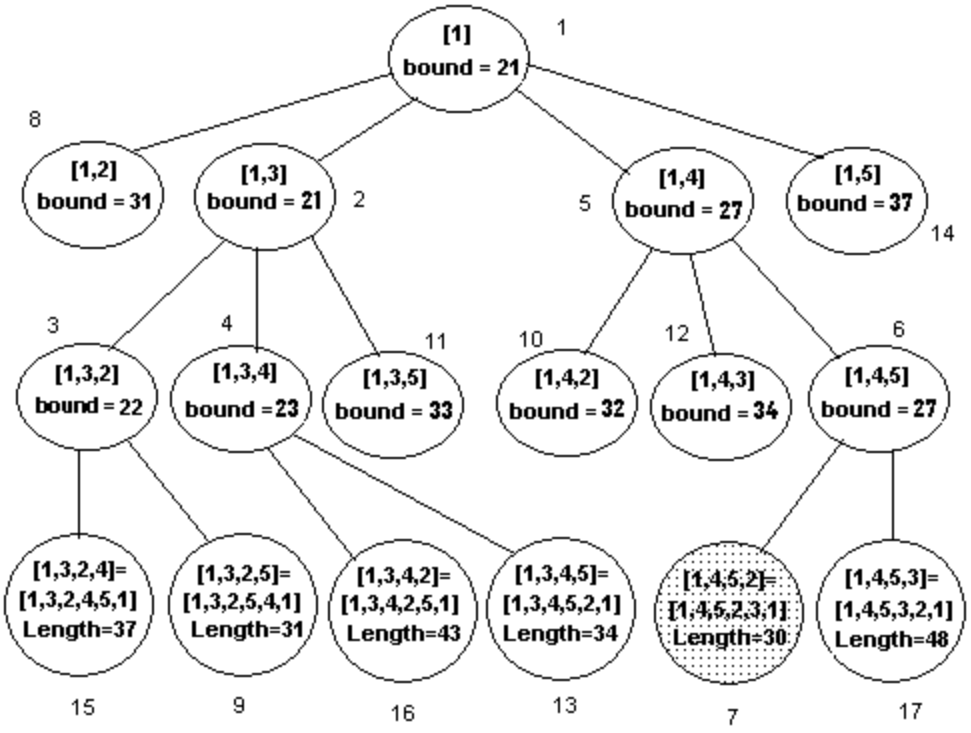
the initial bound for the minimum tour is the sum of the minimum outgoing edges from each vertex.

Vertex 1	min (14, 4, 10, 20)	= 4
Vertex 2	min (14, 7, 8, 7)	= 7
Vertex 3	min (4, 5, 7, 16)	= 4
Vertex 4	min (11, 7, 9, 2)	= 2
Vertex 5	min (18, 7, 17, 4)	= 4
bound		= 21

Since the bound for this node (21) is less than the initial minTour (×), nodes for all of the adjacent vertices are added to the state space tree at level 1. The bound for the node for the partial tour from 1 to 2 is determined to be:

bound = length from 1 to 2 + sum of min outgoing edges for vertices 2 to 5 = 14 + (7 + 4 + 2 + 4) = 31

After each new node is added to the PriorityQueue, the node with the best bound is removed and similarly processed. The algorithm terminates when the queue is empty.



Note that the node for the state with a partial tour from [1,2] is the second node placed in the priority

queue, but the 8th node to be removed. By the time it is removed and examined, a tour of length 30 (which turns out to be the optimal tour) has already been discovered, and, since its bound exceeds this length, it is discarded without having to check any of the possible tours that extend it.

Here is a Branch and Bound algorithm for an adjacency list representation of a graph. (Note! If the first vertex is numbered 1 instead of 0, the array bounds for ***mark*** and ***minEdge*** would have to be (length) N + 1 and the loops traversing these arrays would have to be from 0 (unused) to N (inclusive)).

```
class TSPBranchAndBound {
    private final double INFINITY = 1E10;
    private double minTour;
    private Graph g;       //an adjacency list representation of the graph
    private int N;           //number of vertices
    private Vector bestList;
    private boolean [ ] mark;   //two arrays to keep track of min outgoing edge from
    private double [ ] minEdge;          //each vertex -- used by method bound( )
    private BinaryHeap q;

    class TSPNode implements Comparable{
        int level;
        double length, bound;
        Vector path;
        Object lastVertex;
        protected TSPNode(Object vert) {
            level = 0;
            length = 0.0;
            bound = 0.0;
            lastVertex = vert;
            path = new Vector( );
        }
        protected void copyList(Vector v) {
            if (v == null || v.isEmpty( ) )
                path = new Vector( );
            else
                path = new Vector(v);
        }
        protected void add(Object vtx) {
            //post-condition:     vtx is added to the end of the partial tour
            path.add(vtx);
        }
        public int compareTo(Object obj) {
            //the bound of the two nodes is compared
            TSPNode n = (TSPNode)obj;
            return (int)(n.bound - this.bound);
        }
    }
```

```
public TSPBranchAndBound(Graph G) {
    g = G;
    N = g.size( );
    minTour = INFINITY;
    q = new BinaryHeap( );
    mark = new boolean[N];
    minEdge = new double[N];
}

private void tsp( ) throws HeapUnderflowException {
    while (!q.isEmpty( ) ) {
      //remove node with smallest bound from the queue
      TSPNode temp = (TSPNode)q.deleteMin( );
      if (temp.bound < minTour) {
          Iterator itr = g.neighbors(temp.lastVertex);
          while (itr.hasNext( )) {
              Object nextVert = itr.next( );
              if (!temp.path.contains(nextVert) ) {
                  //if vertex nextVert is not already in the partial tour,
                  // form a new partial tour that extends the tour in node
                  // temp by appending nextVert
                   TSPNode u = new TSPNode(nextVert);
                   u.level = temp.level+1;
                   u.length = temp.length + length(temp.lastVertex, nextVert);
                   u.copyList(temp.path);
                   u.add(nextVert);
                   if (u.level == N - 2) {
                       //if the new partial tour is of length N -1, there is only
                       //one possible
                       //complete tour that can be formed -- form it now
                       Iterator ntr = g.neighbors(nextVert);
                       while (ntr.hasNext( ) ) {
                           Object x = ntr.next( );
                           if (!u.path.contains(x) ) {
                                u.add(x);
                                u.length += length(nextVert, x);
                                u.add(u.path.get(0) );
                                u.length += length(x, u.path.get(0));
                                if (u.length < minTour) {
                                    //if this new complete tour is the best so far,
                                        //save it
                                    minTour = u.length;
                                    bestList = new Vector(u.path);
                                }
                                break;
                           }
                       }
```

```
                    }
                    else {
                        //if the partial tour is "promising" add  node to the
                                //priority queue
                        u.bound = bound(u);
                        if (u.bound < minTour)
                            q.add(u);
                    }
                }
            }
        }
    }

    private double length(Object v1, Object v2) {
        Edge e = g.getEdge(v1, v2);
        Object hold = e.label( );
        return ((Double)hold).doubleValue( );
    }
    private double bound (TSPNode n) {
        //keep an array of vertices -- with minimum outgoing distance for each
        //vertices are labeled by number
        for (int i = 0; i < N; i++)
            mark[i] = false;
        Iterator itr = n.path.iterator( );
        //mark all of the vertices in the partial tour
        while (itr.hasNext( ) ) {
            Integer hold = (Integer)itr.next( );
            int num = hold.intValue( );
            mark[num] = true;
        }
        //unmark the last vertex in the path
        Integer lastv = (Integer)n.lastVertex;
        mark[lastv.intValue( )] = false;
        double bnd = n.length;
        for (int i = 0; i < N; i++) {
            if (!mark[i])
                bnd += minEdge[i];
        }
        return bnd;
    }
    private void init( ) {
       //find and record the minimum outgoing edge from each vertex
       for (int i = 0; i < N; i++) {
           mark[i] = false;
       }
       Iterator itr = g.iterator( );
```

```
            while (itr.hasNext( ) ) {
                Object obj = itr.next( );
                GraphListVertex v = (GraphListVertex)obj;
                Iterator jtr = g.neighbors(v);
                double cost = INFINITY;
                while (jtr.hasNext( ) ) {
                    Object w = jtr.next( );
                    double len = length(v.label( ), w);
                    if (len < cost)
                        cost = len;
                }
                minEdge[((Integer)v.label( )).intValue( )] = cost;
            }
        }

        public void tspPath( ) throws HeapUnderflowException{
            init( );
            Integer v1 = new Integer(1);
            TSPNode root = new TSPNode(v1);
            root.add(v1);
            root.bound = bound(root);
            q.add( root );
            tsp( );
            System.out.print("The TSP path is: ");
            Iterator itr = bestList.iterator( );
            while (itr.hasNext( ) ) {
                System.out.print("  " + itr.next( ) );
            }
            System.out.println( );
            System.out.println("The minimum path length is " + minTour);
        }
    }
```