

Approximate Algorithms for TSP

1. **Multifragment-heuristic algorithm** (inspiration from Kruskal's algorithm)

Step 1. Sort the edges in increasing order of their weights (can use a min-heap)

Step 2. while (!heap.isEmpty())

 Edge nextEdge = heap.deletemin()

 if (next edge does not create a vertex of degree 3 || nextEdge does not create cycle of length < n)

 add it to the tour else skip it

Step 3. return the set of tour edges

If inter-city distances satisfy the following properties:

a) triangle inequality

$d[i,j] \leq d[i,k] + d[k,j]$ for any triple of cities i, j, k

b) symmetry

$d[i,j] = d[j,i]$ for any pair of cities i, j

then these instances of TSP are called Euclidean, and if $f(s_n)$ = the length of a tour calculated by the multifragment-heuristic algorithm and $f(s_n^*)$ is the optimal n-city tour, then we have the following bound:

$$\frac{f(s_n)}{f(s_n^*)} \leq \frac{1}{2} (\lceil \log_2 n \rceil + 1)$$

This gives us an upper-bound on how far the result from the approximate algorithm Differs from the optimal value -- if we could only calculate the optimal in time $O(P(n))$.

2. **Minimum-spanning tree based algorithms** (for a complete Euclidean Graph)

 “Here we go (twice) ‘round the mulberry bush”

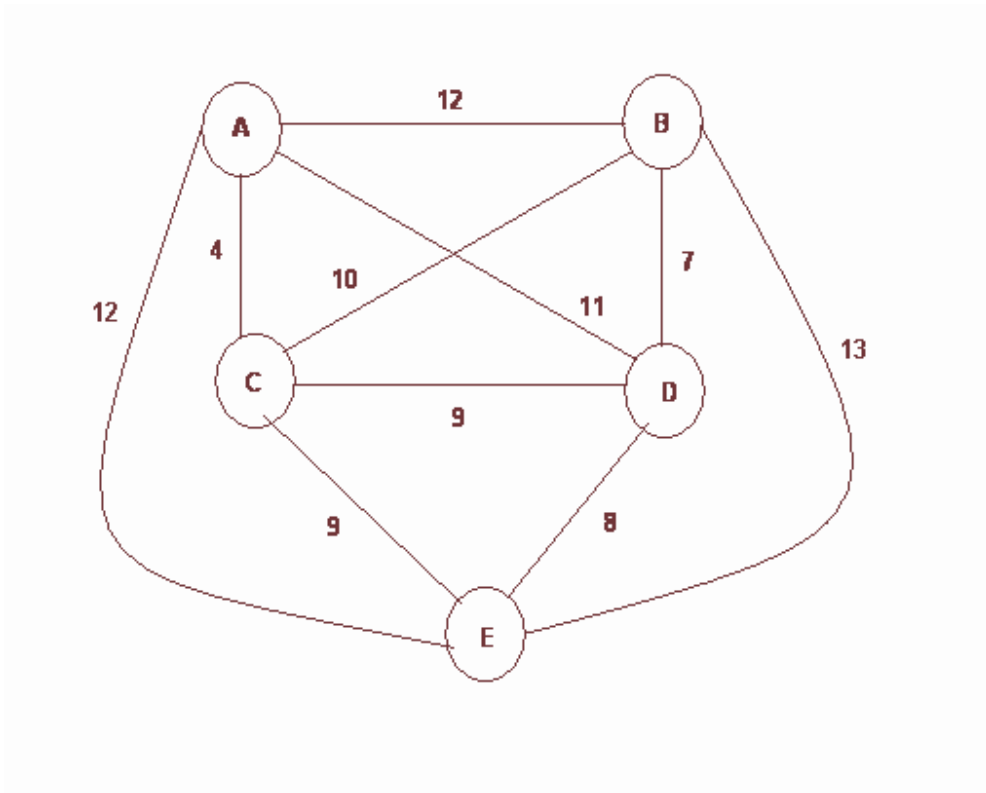
Twice-around the tree algorithm

Step 1. Use Prim or Kruskal to construct an MST of the graph corresponding to an instance of TSP.

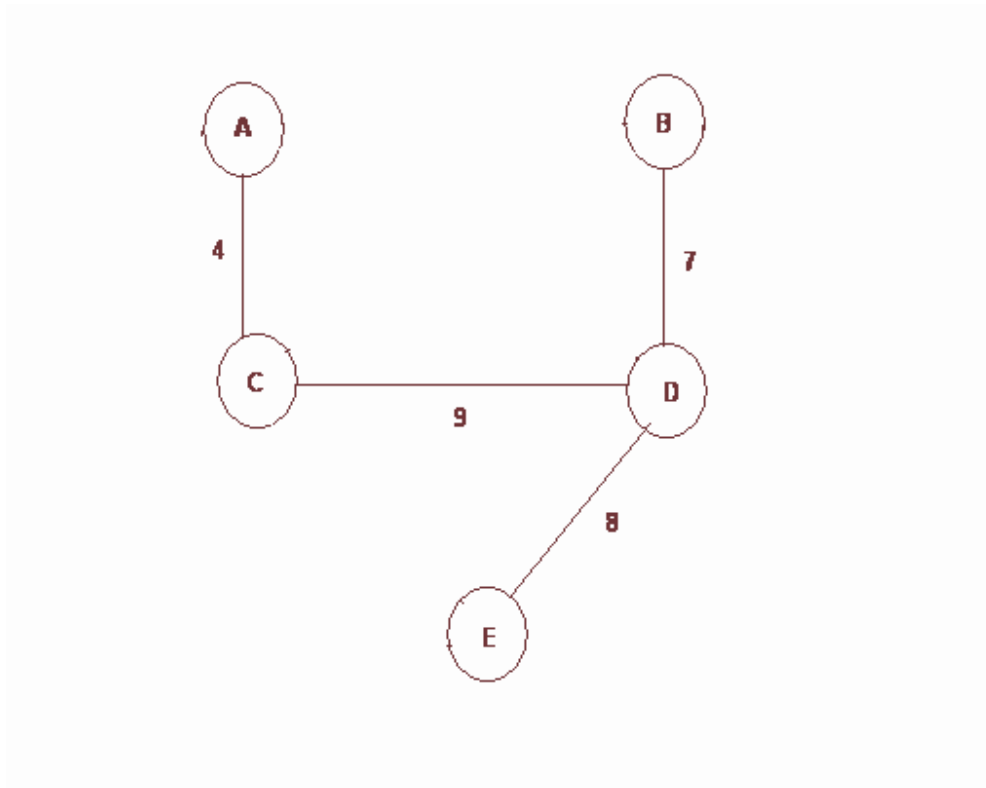
Step 2. Starting at an arbitrary vertex, perform a depth-first traversal of the MST and add each vertex as visited to a list.

Step 3. Iterate through the list of vertices, marking each vertex encountered as visited. When a previously visited vertex is encountered, remove this vertex from the list unless it is the starting vertex.

Consider the following example



The MST in this graph is determined to be the following: $wt(MST) = 28$



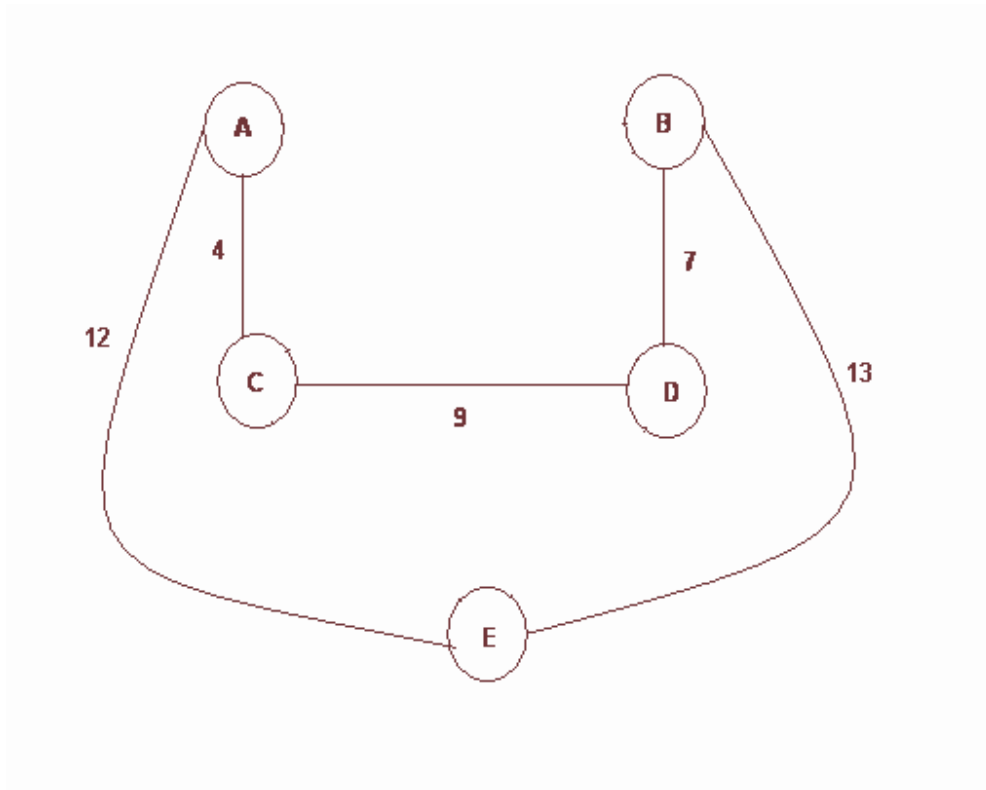
Starting at vertex A and performing a DFS where we list each vertex every time they are visited will produce the following list:

(A, C, D, B, D, E, D, C, A)

Now traverse this list and remove every vertex that has been previously visited

(A, C, D, B, E, A)

This produces the following TSP tour of length 45:



In a Euclidean graph, we have $f(s_n) \leq 2f(s_n^*)$, the tour obtained from the twice around the tree algorithm is no worse than twice the optimal tour length.

Proof

Removing any edge from the optimal tour, results in a spanning tree T where $wt(T) \geq wt(T^*)$ where T^* is the MST of the graph. Thus

$$f(s_n^*) > wt(T) \geq wt(T^*) \quad \text{This inequality implies that}$$

$$2f(s_n^*) > 2wt(T^*) = \text{length of walk obtained in Step 2 above.}$$

In Step 3, the act of removing a previously visited vertex, w , from the list, resulted in replacing a path from vertex u to vertex v through an intermediate vertex w with a direct path between u and v . For Euclidean graphs, the triangle inequality states that

$$d[u,v] \leq d[u,w] + d[w,v]$$

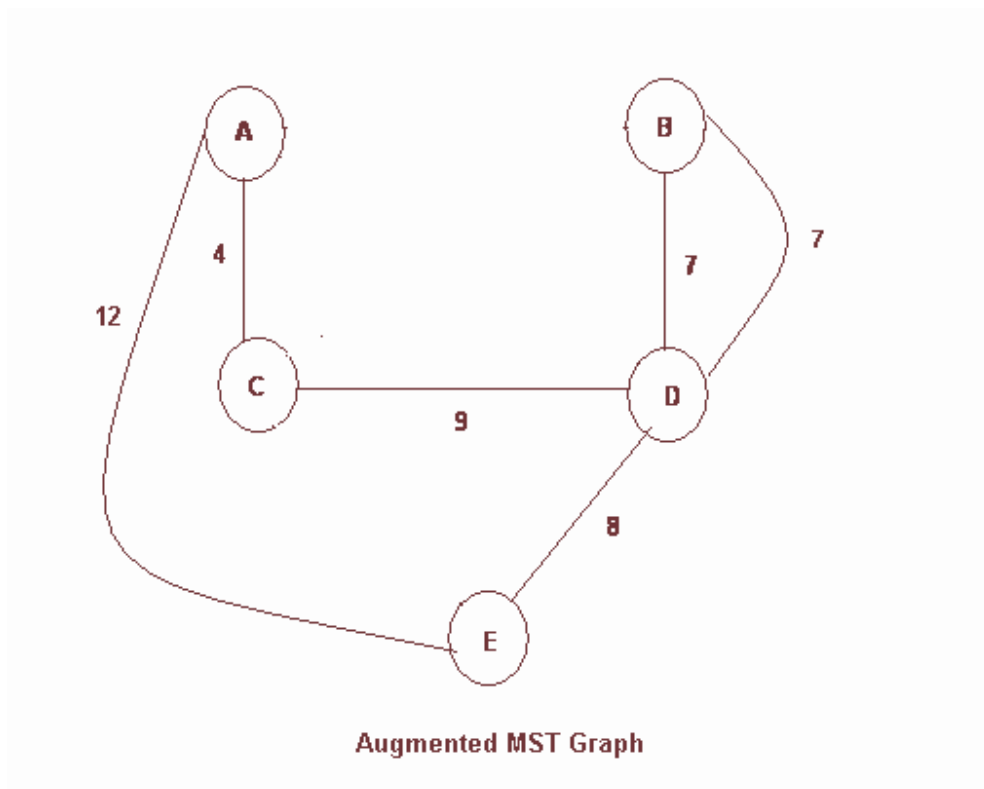
$$2wt(T^*) \geq f(s_n) \quad \text{hence} \quad 2f(s_n^*) > f(s_n)$$

3. Christofides Algorithm

This algorithm finds an MST in the graph for the given instance of TSP. In the MST it locates all of the vertices of odd-degree, and starting at one of these vertices, it adds the minimum weight edges to the MST graph so that all vertices have even degree. Starting from a single vertex, find the set of edges connecting vertices of odd degree that has minimum total weight.

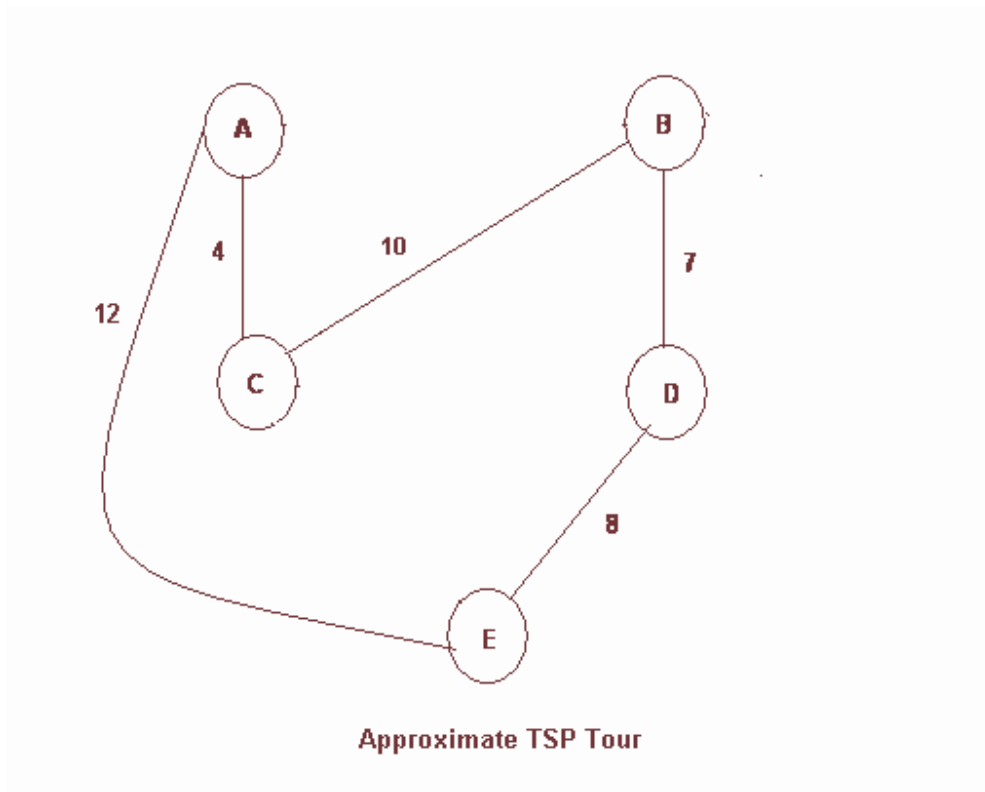
For the Graph shown in the previous algorithm, the vertices A, B, D, and E are all of odd-degree. If we start at vertex A we can add edges $\{(A,E), (D,B)\}$ with $wt = 19$, $\{(A,B), (E,D)\}$ with $wt = 20$, or $\{(A,D), (B,E)\}$ with $wt = 24$. The choice producing the minimum additional weight would be $\{(A,E), (D,B)\}$. (Note! Since this is an undirected graph, it is permissible, in fact necessary, to add edges already in the MST. These edges will be traversed in each direction in the Eulerian tour.) Now since all of the vertices have even degree, find an Eulerian tour through this augmented graph. An Eulerian tour through this graph, starting at vertex A, would be

(A, E, D, B, D, C, A)



Now traverse through this Eulerian tour, removing previously visited vertices and thus creating “short cuts”, to get an approximate TSP tour. In this example we obtain

(A, E, D, B, C, A) with a total length of 41.



4. Branch – and –Bound

This is an adaptation of a backtracking approach that will produce an optimal tour, but will not be $O(P(n))$ for all instances of TSP.

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:

Vertex	Adjacent (outgoing) Edges			
1	(1,2) 14	(1,3) 4	(1,4) 10	(1,5) 20
2	(2,1) 14	(2,3) 7	(2,4) 8	(2,5) 7
3	(3,1) 4	(3,2) 5	(3,4) 7	(3,5) 16
4	(4,1) 11	(4,2) 7	(4,3) 9	(4,5) 2
5	(5,1) 18	(5,2) 7	(5,3) 17	(5,4) 4

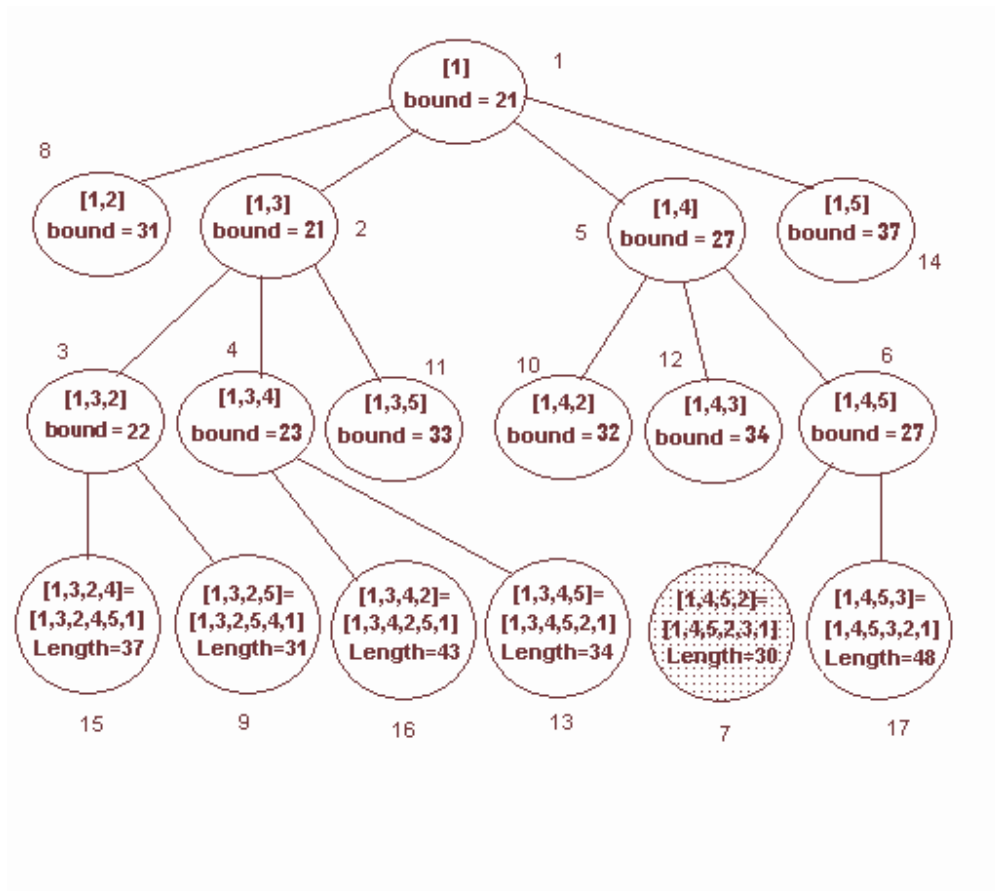
We assume in the implementation of this algorithm that vertices are labeled by an integer number and edges contain the source and sink vertices and a cost or length label. The tour will start at vertex 1, and the initial bound for the minimum tour is the sum of the minimum outgoing edges from each vertex.

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:

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

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 each vertex – used by method bound( )
    private double [ ] minEdge; //second of two arrays
    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);
}
}

```