JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2003**

Vol. 2, No. 2, February-March 2003

Branch and Bound Implementations for the Traveling Salesperson Problem - Part 1: A solution with nodes containing partial tours with constraints

Richard Wiener, Editor-in-Chief, JOT, Associate Professor, Department of Computer Science, University of Colorado at Colorado Springs

1 INTRODUCTION

This is a multi-part column that will appear in the next several issues of JOT.

This first installment outlines a well known branch and bound algorithm for solving the Traveling Salesperson Problem (TSP). A single-threaded Java implementation of this algorithm is presented and discussed along with some results on several moderate sized potentially intractable problems. The implementation provides an opportunity to discuss several important Java implementation issues.

The second column presents another well-known branch and bound algorithm and its single-threaded Java implementation. This implementation also provides an opportunity to discuss some interesting implementation issues.

In the third column, a multi-threaded implementation will be presented based on the second branch and bound algorithm introduced in the second column. Its performance will be compared to the single-threaded implementations presented earlier. This multi-threaded implementation sets the stage for the multi-process distributed implementation to be presented in the fourth column.

This fourth column shall present a distributed implementation for solving TSP using remote method invocation (RMI) in Java. Results using five networked computers running in parallel and featuring three different operating systems will be presented and compared with the single and multi-threaded solutions.

Some of the results presented in these columns might be of some value to educators teaching algorithm design or advanced Java or distributed computing.

2 THE TSP PROBLEM

As researchers in the area of algorithm design know, the Traveling Salesperson Problem (TSP) is one of many combinatorial optimization problems in the set NP-complete. The only known solutions are of exponential complexity and are therefore intractable.

Recall that the TSP assumes that the distances between n cities are specified in a cost matrix that specifies the non-negative cost between any pair of cities. A salesperson is given the task of starting at city 1, traveling to each of the other cities and then returning to city 1. Each city must be visited exactly once and no cities can be skipped. The goal is to find the sequence of cities that starts and ends with city 1 such that the overall cost of the tour is minimized.

When one first encounters this classic problem, it seems relatively easy to solve. Perhaps that is because the problem itself is easy to understand. But consider the fact that there are (n – 1)! tours in an n-city problem since each tour must start and end with city 1. So only for toy-size problems, say n <= 14, would a brute force evaluation of all possible tours and subsequent computation of the minimum tour be feasible. For a problem with 4 cities, the 3! = 6 possible tours are:

1, 2, 3, 4, 1
1, 2, 4, 3, 1
1, 3, 2, 4, 1
1, 3, 4, 2, 1
1, 4, 2, 3, 1
1, 4, 3, 2, 1

Two different fairly well known branch and bound approaches to solving the TSP will be explored and implemented in Java. Several moderate sized problems of sizes 20, 24 and 33 will be used to test the implementations.

3 BRANCH AND BOUND ALGORITHM 1 FOR TSP WITH SYMMETRIC COST MATRIX

A tree of nodes is generated where each node has specified constraints regarding edges connecting two cities in a tour that must be present and edges connecting two cities in a tour that cannot be present. Based on the constraints in a given node, a lower bound is formulated for the given node. This lower bound represents the smallest solution that would be possible if a sub-tree of nodes leading eventually to leaf nodes containing legal tours were generated below the given node. If this lower bound is higher than the best known solution to-date, the node may be pruned. This pruning has the effect of sparing the computational process the need to generate nodes below the given node. This could result in a significant saving if the pruned node were relatively near the top of the tree.

Let us explore the mechanism for computing lower bounds for a node.

The exact cost of a tour always equals the sum of the edges going into and out of each city in the tour divided by 2. For example, consider the 3 city problem shown below with cost matrix:

0	3	3
4	0	5
2	0	4

Consider the tour 1, 2, 3, 1. The sum of the edges going into and out of each node is:

Node 1: 2 + 3 = 5
Node 2: 3 + 5 = 8
Node 3: 5 + 2 = 7
Therefore the tour cost = (5 + 8 + 7) / 2 = 10

For the tour 1, 3, 2, 1 the sum of the edges going into and out of each node is:

Node 1: 3 + 4 = 7
Node 2: 4 + 4 = 8
Node 3: 3 + 4 = 7
Therefore the tour cost = (7 + 8 + 7) / 2 = 11

In general, *the sum of the edges going into and out of a tour node is no less than the sum of the two edges of least cost going into and out of the tour node. Therefore, no tour can cost less than one-half the sum over all the nodes of the two lowest cost edges going into and out of each node.*

Construction of Solution Tree

A solution tree is constructed by adding edges in lexicographic order. Each time we add a new node we employ decision tree logic regarding which nodes must be included or excluded from tours represented by the nodes. The rules that are used are:

1. If excluding an edge (x, y) would make it impossible for x or y to have as many as two adjacent edges in the tour, then (x, y) must be included.
2. If including (x, y) would cause x or y to have more than two edges adjacent in the tour, or would complete a non-tour cycle with edges already included, then (x, y) must be excluded.

When the algorithm branches, and after imposing the decision logic to include or exclude edges, a lower bound is computed for the node. If the lower bound for a given node is as high or higher than the lowest cost tour found so far, we prune the node. If neither child can be pruned, the algorithm descends to the node with smaller lower bound using a depth-first search in the tree. After considering one child, we must again consider whether the sibling can be pruned since a new best solution may have been found.

Example

Consider the six-city problem with cost matrix given as follows:

0	8	5	3	1	2
8	0	4	9	2	8
5	4	0	9	6	7
3	9	9	0	1	1
1	2	6	1	0	9
2	8	7	1	9	0

Although for this toy-sized problem it would be easy to enumerate the 120 possible tours and compute the tour with lowest cost, we shall illustrate the branch and bound process by constructing a solution tree.

To compute the lower bound for a solution to the problem for the root node that contains no constraints (all tours are possible):

The five edges incident on node 1 are: 8, 5, 3, 1, 2. The two smallest edges are 1 and 2.
The five edges incident on node 2 are: 8, 4, 9, 2, 8. The two smallest edges are 2 and 4.
The five edges incident on node 3 are: 5, 4, 9, 6, 7. The two smallest edges are 4 and 5.
The five edges incident on node 4 are: 3, 9, 9, 1, 1. The two smallest edges are 1 and 1.
The five edges incident on node 5 are: 1, 2, 6, 1, 9. The two smallest edges are 1 and 1.
The five edges incident on node 6 are: 2, 8, 7, 1, 9. The two smallest edges are 1 and 2.
Therefore, twice the lower bound for this unconstrained root node is therefore: 1 + 2 + 2 + 4 + 4 + 5 + 1 + 1 + 1 + 1 + 1 + 2 = 25.

From the root node we generate two child nodes with the constraints 12 and *12. The constraint *12* means that the edge 12 must be present in the tour. The constraint **12* means that the edge 12 cannot be present in the tour.

Let us walk through the computation of the lower bound 35 when the constraint is 12.

The five edges incident on node 1 are: 8, 5, 3, 1, 2. Since edge 12 must be present, the sum of the two smallest edges in the presence of this constraint is 1 + 8 = 9.

The five edges incident on node 2 are: 8, 4, 9, 2, 8. Since edge 12 must be present, the sum of the two smallest edges in the presence of this constraint is 2 + 8 = 10.

The five edges incident on node 3 are: 5, 4, 9, 6, 7. The two smallest edges are 4 and 5.
The five edges incident on node 4 are: 3, 9, 9, 1, 1. The two smallest edges are 1 and 1.
The five edges incident on node 5 are: 1, 2, 6, 1, 9. The two smallest edges are 1 and 1.
The five edges incident on node 6 are: 2, 8, 7, 1, 9. The two smallest edges are 1 and 2.

Twice the lower bound with the constraint 12 is therefore 9 + 10 + 4 + 5 + 1 + 1 + 1 + 1 + 1 + 2 = **35**.

Using the same approach, it is easy to show that the lower bound on the node with constraint *12 is **25**.

Since the right child node has a smaller lower bound than the left child node, the best first algorithm constructs nodes below the right child node.

The left child of the node with constraint *12 has the constraints *12 and 13 and the right child the constraints *12 and *13 (the 13 and *13 are introduced in lexicographical order). The reader is left to verify that twice the lower bounds under these constraints are 28 and 26 respectively.

The depth-first generation of the tree continues using the node with lower bound 26.

Let us consider the right node with twice the lower bound of 26. The three constraints are 15, 16 and *56. These are constraints derived from the decision rules stated earlier. The 15 and 16 are present because of the presence of the *12, *13 and *14 (each tour must have two edges incident from node 1). The *56 is present since a non-tour cycle given by 1, 5, 6, 1 would be possible if the constraint *56 were not present.

We shall perform one last lower bound computation on the node with constraints: *12, *13, 15, 16, *56.

The five edges incident on node 1 are: 8, 5, 3, 1, 2. Edges 1 and 2 are smallest.
The five edges incident on node 2 are: 8, 4, 9, 2, 8. Edges 2 and 4 are smallest.
The five edges incident on node 3 are: 5, 4, 9, 6, 7. Edges 4 and 6 are smallest subject to the constraint *13.
The five edges incident on node 4 are: 3, 9, 9, 1, 1. Edges 1 and 1 are smallest.
The five edges incident on node 5 are: 1, 2, 6, 1, 9. Edges 1 and 1 are smallest.
The five edges incident on node 6 are: 2, 8, 7, 1, 9. Edges 1 and 2 are smallest.
Therefore, twice the lower bound is the sum which equals 26.

For the problem stated above and using a descriptive symbolic notation, we show the complete sequence of nodes generated and pruned using the best-first, branch and bound algorithm outlined above. We define each node by its constraints, its computed lower bound (actually twice the lower bound) and its left and right children, each with their lower bounds. The actual sequence of actions may be followed by examining the sequence of bold-faced lines.

It would be instructive to sketch the entire tree based on the sequence of events shown below.

branchAndBound() – Root node with no constraints
Twice the lower bound = 25
leftChild = 12   with lowerBound() = 35
rightChild = *12   with lowerBound() = 25

branchAndBound(*12) – The constraints on this node are *12
Twice the lower bound = 25
leftChild = *12 13   with lowerBound() = 28
rightChild = *12 *13   with lowerBound() = 26

branchAndBound(*12 *13) – The constraints on this node are *12 and *13
Twice the lower bound = 26
leftChild = *12 *13 14   with lowerBound() = 29
rightChild = *12 *13 *14 15 16 *56   with lowerBound() = 26

branchAndBound(*12 *13 *14 15 16 *56)
Twice the lower bound = 26
leftChild = *12 *13 *14 15 16 23 *56   with lowerBound() = 26
rightChild = *12 *13 *14 15 16 *23 *56   with lowerBound() = 33

branchAndBound(*12 *13 *14 15 16 23 *56)
Twice the lower bound = 26
leftChild = *12 *13 *14 15 16 23 24 *25 *26 *34 *56   with lowerBound() = 41
rightChild = *12 *13 *14 15 16 23 *24 *56   with lowerBound() = 26

branchAndBound(*12 *13 *14 15 16 23 *24 *56)
Twice the lower bound = 26
leftChild = *12 *13 *14 15 16 23 *24 25 *26 34 *35 *36 *45 46 *56   with lowerBound() = 38
rightChild = *12 *13 *14 15 16 23 *24 *25 26 34 *35 *36 45 *46 *56   with lowerBound() = 50

branchAndBound(*12 *13 *14 15 16 23 *24 25 *26 34 *35 *36 *45 46 *56)
*12 *13 *14 15 16 23 *24 25 *26 34 *35 *36 *45 46 *56   is a tour of cost 19.

Prune node *12 *13 *14 15 16 23 *24 *25 26 34 *35 *36 45 *46 *56
Prune node *12 *13 *14 15 16 23 24 *25 *26 *34 *56

branchAndBound(*12 *13 *14 15 16 *23 *56)
Twice the lower bound = 33
leftChild = *12 *13 *14 15 16 *23 24 *56   with lowerBound() = 42 must be pruned.

rightChild = *12 *13 *14 15 16 *23 *24 25 26 34 *35 *36 *45 *46 *56 with lowerBound() = 44 must be pruned.

branchAndBound(*12 *13 14)
Twice the lower bound = 29
leftChild = *12 *13 14 15 *16 *45 with lowerBound() = 35
rightChild = *12 *13 14 *15 16 *46 with lowerBound() = 37

branchAndBound(*12 *13 14 15 *16 *45)
Twice the lower bound = 35
leftChild = *12 *13 14 15 *16 23 *45 with lowerBound() = 35
rightChild = *12 *13 14 15 *16 *23 *45 with lowerBound() = 42 must be pruned.

branchAndBound(*12 *13 14 15 *16 23 *45)
Twice the lower bound = 35
leftChild = *12 *13 14 15 *16 23 24 *25 *26 *34 *35 36 *45 *46 56 with lowerBound() = 66 must be pruned.
rightChild = *12 *13 14 15 *16 23 *24 *45 with lowerBound() = 35

branchAndBound(*12 *13 14 15 *16 23 *24 *45)
Twice the lower bound = 35
leftChild = *12 *13 14 15 *16 23 *24 25 *26 *34 *35 36 *45 46 *56 with lowerBound() = 36
rightChild = *12 *13 14 15 *16 23 *24 *25 26 *36 *45 with lowerBound() = 46 must be pruned.

branchAndBound(*12 *13 14 15 *16 23 *24 25 *26 *34 *35 36 *45 46 *56)
*12 *13 14 15 *16 23 *24 25 *26 *34 *35 36 *45 46 *56 is a tour of cost 18.

Prune node *12 *13 14 *15 16 *46

branchAndBound(*12 13)
Twice the lower bound = 28
leftChild = *12 13 14 *15 *16 *34 with lowerBound() = 38 must be pruned.
rightChild = *12 13 *14 with lowerBound() = 28

branchAndBound(*12 13 *14)
Twice the lower bound = 28
leftChild = *12 13 *14 15 *16 *35 with lowerBound() = 33
rightChild = *12 13 *14 *15 16 *36 with lowerBound() = 30

branchAndBound(*12 13 *14 *15 16 *36)
Twice the lower bound = 30

leftChild = *12 13 *14 *15 16 23 *26 *34 *35 *36 with lowerBound() = 30
rightChild = *12 13 *14 *15 16 *23 *36 with lowerBound() = 36

branchAndBound(*12 13 *14 *15 16 23 *26 *34 *35 *36)
Twice the lower bound = 30
leftChild = *12 13 *14 *15 16 23 24 *25 *26 *34 *35 *36 45 *46 56 with lowerBound() = 60 must be pruned.
rightChild = *12 13 *14 *15 16 23 *24 25 *26 *34 *35 *36 45 46 *56 with lowerBound() = 30

branchAndBound(*12 13 *14 *15 16 23 *24 25 *26 *34 *35 *36 45 46 *56)
*12 13 *14 *15 16 23 *24 25 *26 *34 *35 *36 45 46 *56 is a tour of cost 15.

Prune node *12 13 *14 *15 16 *23 *36
Prune node *12 13 *14 15 *16 *35
Prune node 12

Summary Results

Cost of optimum tour: 15
Optimum tour: 1 3 2 5 4 6 1
Total of nodes generated: 31
Total number of nodes pruned: 13

Even for this toy-sized problem, the number of nodes actually generated is a relatively small fraction of the total number of nodes that would have been required if a bounding algorithm were not in play.

For larger problems, there is of course no guarantee that the number of nodes that are pruned would make the solution computationally tractable. There is no way (at least no way known to this author) to predict the success of this algorithm other than simply deploying it on a given problem and deciding when “enough-is-enough.”

4 JAVA IMPLEMENTATION

There are many challenges involved in implementing the algorithm described above. These include:

1. Determining how to generate and store the correct set of constraints for each node.
2. Determining how to compute the lower bound for a node given its set of constraints.
3. Determining when a node represents a complete tour.

4. Determining a mechanism for generating nodes in the correct order and pruning nodes based on their computed lower bound.

The answers to the first three questions are encapsulated in a class *Node* presented below.

Among the fields defined for class *Node* is constraints, a two-dimensional array of type byte. After much experimentation it was decided to use this structure since it is computationally fast (as all array types are in Java) and relatively inexpensive (the byte is Java's smallest unit of storage).

The methods *addRequiredEdges* and *addDisallowedEdges* are used to formulate the constraints for a particular node.

The method *isCycle* uses the standard class *Bitset*. This method is of fundamental importance.

It is recommended that the interested reader "walk-through" the various methods using a specific node taken from the example presented above. Verify that the methods *addRequiredEdges* and *addDisallowedEdges* provide the answer to the first question given above. The method *computeLowerBound* provides the answer to the second question. The method *isTour* provides the answer to the third question. Each of these methods closely follows the algorithm discussion presented earlier.

Listing 1 – Class Node

```
/**
  * The information contained in a tree used for solving TSP using
  * branch and bound.
*/

import java.util.*;
import java.awt.*;

public class Node {
    // Fields
    private int lowerBound;
    private int numRows, numCols;
    private byte [][] constraint;
            // -1 indicates no edge from row to column allowed,
            //  1 indicates that edge from row to column required,
            //  0 indicates that edge from row to column allowed

    private  short  []  nodeCosts;  //  Used  to  compute  smallest  and
                            // nexSmallest
    private int edges;          // Used by isTour query
    private int tourCost;       // Used by isTour query
    private byte [] trip;
    private String nodeAsString; // Used by isTour query
    private boolean isLoop = false;
    static BitSet b; // Used by isCycle and initialized in TSPUI

    // Constructor
    public Node (int numRows, int numCols) {
        this.numRows = numRows;
        this.numCols = numCols;
        nodeCosts = new short[numCols + 1]; // Natural indexing
        constraint = new byte[numRows + 1][numCols + 1]; // Natural
                                                         // indexing
        trip = new byte[numRows + 1];
    }

    // Commands
    public void assignConstraint (byte value, int row,
                                  int col) {
        constraint[row][col] = value;
        constraint[col][row] = value;
    }

    public int assignPoint (Point p, int edgeIndex) {
        // Advance edgeIndex until edge that is unconstrained is found
        Point pt = p;
        while (edgeIndex < TSP.newEdge.size() &&
                constraint[(int)           Math.abs(pt.getX())][(int)
                            Math.abs(pt.getY())]
                != 0) {
            edgeIndex++;
            if (edgeIndex < TSP.newEdge.size()) {
                pt = (Point) TSP.newEdge.get(edgeIndex);
            }
        }
        if (edgeIndex < TSP.newEdge.size()) {
            if (pt.getX() < 0) {
                assignConstraint((byte) -1, (int) Math.abs(pt.getX()),
                                 (int) Math.abs(pt.getY()));
            } else {
                assignConstraint((byte) 1, (int) pt.getX(),
                                  (int) pt.getY());
            }
        }
        return edgeIndex;
    }

    public void setConstraint (byte [][] constraint) {
        this.constraint = constraint;
    }

    public void addDisallowedEdges () {
        for (int row = 1; row <= numRows; row++) {
            // Count the number of paths from row.

            // If the count exceeds one then disallow all other paths
            // from row
            int count = 0;
            for (int col = 1; col <= numCols; col++) {
                if (row != col && constraint[row][col] == 1) {
                    count++;
                }
            }
            if (count >= 2) {
                for (int col = 1; col <= numCols; col++) {
                    if (row != col && constraint[row][col] == 0) {
                        constraint[row][col] = -1;
                        constraint[col][row] = -1;
                    }
                }
            }
        }

        // Check to see whether the presence of a col causes a premature
        // circuit
        for (int row = 1; row <= numRows; row++) {
            for (int col = 1; col <= numCols; col++) {
                if (row != col && isCycle(row, col) &&
                        numCities(b) < numRows) {
                    if (constraint[row][col] == 0) {
                        constraint[row][col] = -1;
                        constraint[col][row] = -1;
                    }
                }
            }
        }
    }


    public void addRequiredEdges () {
        for (int row = 1; row <= numRows; row++) {
            // Count the number of paths excluded from row

            // If the count equals numCols - 3, include all remaining
            // paths
            int count = 0;
            for (int col = 1; col <= numCols; col++) {
                if (row != col && constraint[row][col] == -1) {
                    count++;
                }
            }
            if (count >= numRows - 3) {
                for (int col = 1; col <= numCols; col++) {
                    if (row != col && constraint[row][col] == 0) {
                        constraint[row][col] = 1;
                        constraint[col][row] = 1;
                    }
                }
            }
        }
    }

    public void computeLowerBound () {
        int lowB = 0;
        for (int row = 1; row <= numRows; row++) {
            for (int col = 1; col <= numCols; col++) {
                nodeCosts[col] = TSP.c.cost(row, col);
            }
            nodeCosts[row] = Short.MAX_VALUE;
            // Eliminate edges that are not allowed
            for (int col = 1; col <= numCols; col++) {
                if (constraint[row][col] == -1) {
                    nodeCosts[col] = Short.MAX_VALUE; // Taken out of
                                                      // contention
                }
            }
            int [] required = new int[numCols - 1]; // Natural indexing
            int numRequired = 0;

            // Determine whether an edge is required
            for (int col = 1; col <= numCols; col++) {
                if (constraint[row][col] == 1) {
                    numRequired++;
                    required[numRequired] = nodeCosts[col];
                    nodeCosts[col] = Short.MAX_VALUE; // Taken out of
                                                      // contention
                }
            }
            int smallest = 0, nextSmallest = 0;
            if (numRequired == 0) {
                smallest = smallest();
                nextSmallest = nextSmallest();
            } else if (numRequired == 1) {
                smallest = required[1];
                nextSmallest = smallest();
            } else if (numRequired == 2) {
                smallest = required[1];
                nextSmallest = required[2];
            }
            if (smallest == Short.MAX_VALUE) {
                smallest = 0;
            }
            if (nextSmallest == Short.MAX_VALUE) {
                nextSmallest = 0;
            }
            lowB += smallest + nextSmallest;
        }
        lowerBound = lowB; // This is twice the actual lower bound
    }

    public void setTour () {
        byte path = 0;
        for (int col = 2; col <= numCols; col++) {
            if (constraint[1][col] == 1) {
                path = (byte) col;
                break;
            }
        }
        tourCost = TSP.c.cost(1, path);
        trip[1] = path;
        int row = 1;
        int col = path;
        int from = row;
        byte pos = path;
        nodeAsString = "" + row + " " + col;
        while (pos != row) {
            for (byte column = 1; column <= numCols; column++) {
                if (column != from && constraint[pos][column] == 1) {
                    from = pos;
                    pos = column;
                    nodeAsString += " " + pos;
                    tourCost += TSP.c.cost(from, pos);
                    trip[from] = pos;
                    break;
                }
            }
        }
    }

    // Queries
    public int tourCost () {
        return tourCost;
    }

    public byte [] trip () {
        return trip;
    }

    public byte constraint (int row, int col) {
        return constraint[row][col];
    }

    public byte [][] constraint () {
        return constraint;
    }

    public int lowerBound () {
        return lowerBound;
    }

    public boolean isTour () {
        // Determine path from 1
        int path = 0;
        for (int col = 2; col <= numCols; col++) {
            if (constraint[1][col] == 1) {
                path = col;
                break;
            }
        }
        if (path > 0) {
            boolean cycle = isCycle(1, path);
            return cycle && numCities(b) == numRows;
        } else {
            return false;
        }
    }

    public boolean isCycle (int row, int col) {
        // b = new BitSet(numRows + 1);
        for (int i = 0; i < numRows + 1; i++) {
            b.clear(i);
        }

        b.set(row);
        b.set(col);
        int from = row;
        int pos = col;
        int edges = 1;
        boolean quit = false;
        while (pos != row && edges <= numCols &&
                !quit) {
            quit = true;
            for (int column = 1; column <= numCols; column++) {
                if (column != from && constraint[pos][column] == 1) {
                    edges++;
                    from = pos;
                    pos = column;
                    b.set(pos);
                    quit = false;
                    break;
                }
            }
        }
        return pos == row || edges >= numCols;
    }

    public String tour () {
        return nodeAsString;
    }

    public String toString () {
        // String representation of constraint matrix
        String returnString = "";
        for (int row = 1; row <= numRows; row++) {
            for (int col = row + 1; col <= numCols; col++) {
                if (constraint[row][col] == 1) {
                    returnString += "" + row + col + "  ";
                } else if (constraint[row][col] == -1) {
                    returnString += "*" + row + col + "  ";
                }
            }
        }
        return returnString;
    }

    // Internal methods
    private int smallest () {
        int s = nodeCosts[1];
        int index = 1;
        for (int i = 2; i <= numCols; i++) {
            if (nodeCosts[i] < s) {
                s = nodeCosts[i];
                index = i;
            }
        }
        short temp = nodeCosts[1];
        nodeCosts[1] = nodeCosts[index];
        nodeCosts[index] = temp;
        return nodeCosts[1];
    }

    private int nextSmallest () {
        int ns = nodeCosts[2];
        int index = 2;
        for (int i = 2; i <= numCols; i++) {
            if (nodeCosts[i] < ns) {
                ns = nodeCosts[i];
                index = i;
            }
        }
        short temp = nodeCosts[2];
        nodeCosts[2] = nodeCosts[index];
        nodeCosts[index] = temp;
        return nodeCosts[2];
    }

    private int numCities (BitSet b) {
        int num = 0;
        for (int i = 1; i <= numRows; i++) {
            if (b.get(i)) {
                num++;
            }
        }
        return num;
    }
}
```

The fourth question is embedded in the next class TSP presented below.

The methods *generateSolution* and its supporting private method *branchAndBound* answer this fourth question. Although these methods are voluminous, they are straight-forward. Numerous comments throughout the code indicate the purpose of each segment. The generation of nodes follows the algorithm description given earlier.

Listing 2 – Class TSP

```
/**
  * TSP Branch and Bound
*/

import java.awt.*;
import java.util.*;

public class TSP {
    // Fields

    private final int numRows;
    private final int numCols;
    private TimeInterval t = new TimeInterval();

    private int bestTour = Integer.MAX_VALUE / 4;

    private Node bestNode;
    public static Cost c;
    public static ArrayList newEdge = new ArrayList();
         // Contains objects of type Point

    private int newNodeCount = 0;
    private int numberPrunedNodes = 0;
    private Random rnd = new Random();

    public TSP (short [][] costMatrix, int size, int bestTour) {
        this.bestTour = bestTour;
        numRows = numCols = size;
        c = new Cost(numRows, numCols);
        for (int row = 1; row <= size; row++) {
            for (int col = 1; col <= size; col++) {
                c.assignCost(costMatrix[row][col], row,
                             col);
            }
        }
    }

    public TSP (short [][] costMatrix, int size) {
        numRows = numCols = size;
        c = new Cost(numRows, numCols);
        for (int row = 1; row <= size; row++) {
            for (int col = 1; col <= size; col++) {
                c.assignCost(costMatrix[row][col], row,
                             col);
            }
        }
    }

    public void generateSolution () {
        Point pt;

        // Load newEdge Vector of edge points
        for (int row = 1; row <= numRows; row++) {
            for (int col = row + 1; col <= numCols; col++) {
                pt = new Point(row, col);
                newEdge.add(pt);
                pt = new Point(-row, -col);
                newEdge.add(pt);
            }
        }

        // Create root node
        Node root = new Node(numRows, numCols);
        newNodeCount++;
        root.computeLowerBound();
```

```
        System.out.println(
           "Twice the lower bound for root node (no constraints): " +
            root.lowerBound());

        // Apply the branch and bound algorithm
        t.startTiming();
        branchAndBound(root, -1);
        t.endTiming();
        if (bestNode != null) {
            System.out.println("\n\nCost of optimum tour: " +
                    bestTour + "\nOptimum tour: " + bestNode.tour() +
                    "\nTotal of nodes generated: " + newNodeCount +
                    "\nTotal number of nodes pruned: " +
                    numberPrunedNodes);
        } else {
            System.out.println(
                    "Tour obtained heuristically is the best tour.");
        }
        System.out.println("Elapsed time for entire algorithm: " + t
                .getElapsedTime() + " seconds.");
        System.out.println();
    }

    // Queries
    public int nodesCreated () {
        return newNodeCount;
    }

    public int nodesPruned () {
        return numberPrunedNodes;
    }

    public String tour () {
        if (bestNode != null) {
            return bestNode.tour();
        } else {
            return "";
        }
    }

    public int tourCost () {
        return bestTour;
    }

    public byte [] trip () {
        if (bestNode != null) {
            return bestNode.trip();
        } else {
            return null;
        }
    }


    private void branchAndBound (Node node, int edgeIndex) {
        if (node != null && edgeIndex < newEdge.size()) {
```

```
Node leftChild, rightChild;
int leftEdgeIndex = 0, rightEdgeIndex = 0;
if (node.isTour()) {
    node.setTour();
    if (node.tourCost() < bestTour) {
        bestTour = node.tourCost();
        bestNode = node;
        System.out.println("\n\nBest tour cost so far: " +
          bestTour + "\nBest tour so far: " +
          bestNode.tour() +
          "\nNumber of nodes generated so far: " +
          newNodeCount +
          "\nTotal number of nodes pruned so far: " +
          numberPrunedNodes +
          "\nElapsed time to date for branch and bound: " +
                t.getElapsedTime() + " seconds.\n");
    }
} else {
    if (node.lowerBound() < 2 * bestTour) {
        // Create left child node
        leftChild = new Node(numRows, numCols);
        newNodeCount++;
        if (newNodeCount % 1000 == 0) {
          Point p = (Point) newEdge.get(edgeIndex);
          t.endTiming();
          System.out.println(
           "\nTotal number of nodes created so far: " +
           newNodeCount +
           "\nTotal number of nodes pruned so far: " +
           numberPrunedNodes +
           "\nElapsed time to date for branch and bound: " +
                    t.getElapsedTime() + " seconds.");
        } else if (newNodeCount % 25 == 0) {
            System.out.print(".");
        }
        if (newNodeCount % 10000 == 0 &&
                bestNode != null) {
            System.out.println(
                    "\n\nBest tour cost so far: " +
                    bestTour + "\nBest tour so far: " +
                    bestNode.tour());
        }
        leftChild.setConstraint(copy(node.constraint()));
        if (edgeIndex != -1 &&
            ((Point) newEdge.get(edgeIndex)).getX() > 0) {
            edgeIndex += 2;
        } else {
            edgeIndex++;
        }
        if (edgeIndex >= newEdge.size()) {
            return;
        }
        Point p = (Point) newEdge.get(edgeIndex);
        leftEdgeIndex =
            leftChild.assignPoint(p, edgeIndex);
        leftChild.addDisallowedEdges();
        leftChild.addRequiredEdges();
        leftChild.addDisallowedEdges();
        leftChild.addRequiredEdges();
        leftChild.computeLowerBound();
        if (leftChild.lowerBound() >= 2 * bestTour) {
            leftChild = null;
            numberPrunedNodes++;
        }

        // Create right child node
        rightChild = new Node(numRows, numCols);
        newNodeCount++;
        if (newNodeCount % 1000 == 0) {
          System.out.println(
          "\nTotal number of nodes created so far: " +
          newNodeCount +
          "\nTotal number of nodes pruned so far: " +
                    numberPrunedNodes);
        } else if (newNodeCount % 25 == 0) {
            System.out.print(".");
        }
        rightChild.setConstraint(copy(node.constraint()));
        if (leftEdgeIndex >= newEdge.size()) {
            return;
        }
        p = (Point) newEdge.get(leftEdgeIndex + 1);
        rightEdgeIndex =
            rightChild.assignPoint(p, leftEdgeIndex +
                                               1);
        rightChild.addDisallowedEdges();
        rightChild.addRequiredEdges();
        rightChild.addDisallowedEdges();
        rightChild.addRequiredEdges();
        rightChild.computeLowerBound();
        if (rightChild.lowerBound() > 2 * bestTour) {
            rightChild = null;
            numberPrunedNodes++;
        }

        if (leftChild != null && rightChild == null) {
            branchAndBound(leftChild, leftEdgeIndex);
        } else if (leftChild == null &&
                rightChild != null) {
            branchAndBound(rightChild, rightEdgeIndex);
        } else if (leftChild != null &&
                rightChild != null &&
                leftChild.lowerBound() <= rightChild.
                lowerBound()) {
            if (leftChild.lowerBound() < 2 * bestTour) {
                branchAndBound(leftChild,
                               leftEdgeIndex);
            } else {
                leftChild = null;
                numberPrunedNodes++;
            }
```

```
                        if (rightChild.lowerBound() < 2 * bestTour) {
                            branchAndBound(rightChild,
                                           rightEdgeIndex);
                        } else {
                            rightChild = null;
                            numberPrunedNodes++;
                        }
                    } else if (rightChild != null) {
                        if (rightChild.lowerBound() < 2 * bestTour) {
                            branchAndBound(rightChild,
                                           rightEdgeIndex);
                        } else {
                            rightChild = null;
                            numberPrunedNodes++;
                        }
                        if (leftChild.lowerBound() < 2 * bestTour) {
                            branchAndBound(leftChild,
                                           leftEdgeIndex);
                        } else {
                            leftChild = null;
                            numberPrunedNodes++;
                        }
                    }
                }
            }
        }
    }

    private byte [][] copy (byte [][] constraint) {
        byte [][] toReturn = new byte[numRows + 1][numCols + 1];
        for (int row = 1; row <= numRows; row++) {
            for (int col = 1; col <= numCols; col++) {
                toReturn[row][col] = constraint[row][col];
            }
        }
        return toReturn;
    }
}
```

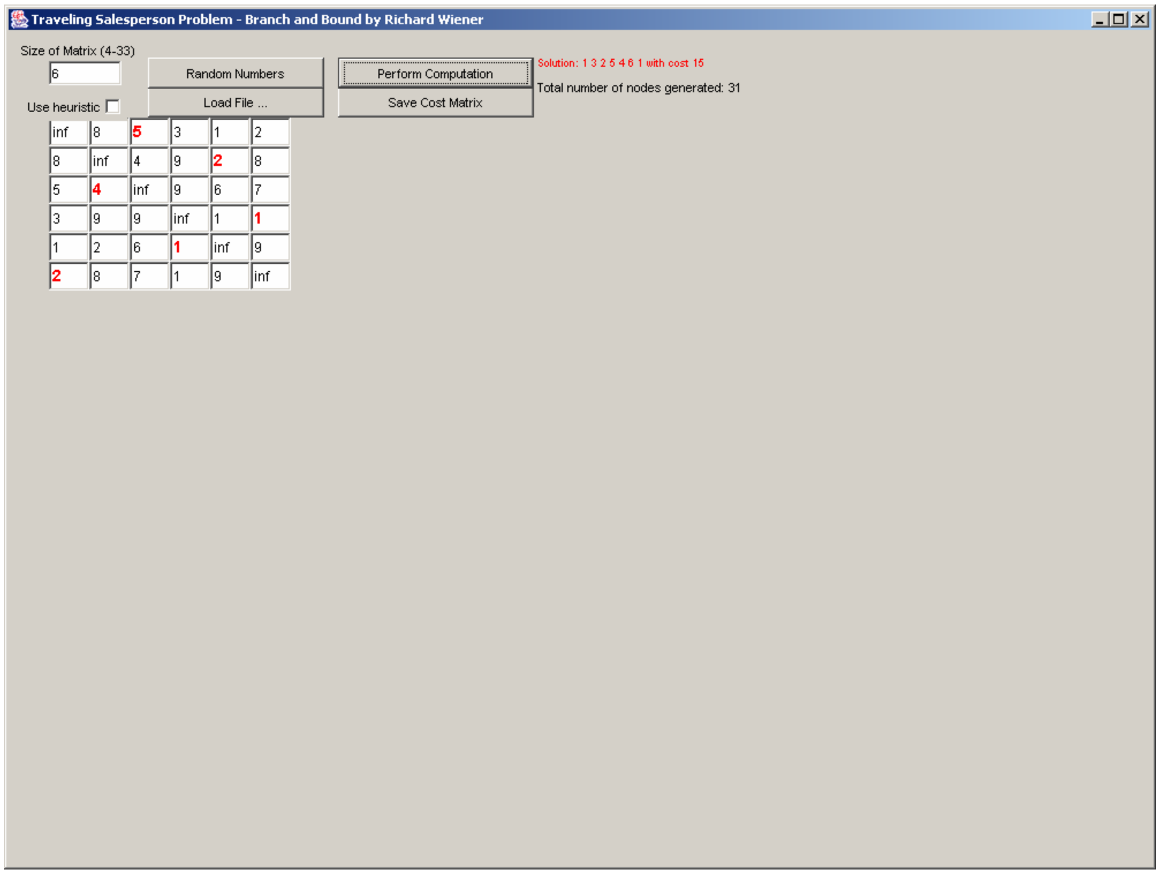
5 EMPIRICAL RESULTS

The process of generating nodes is relatively expensive both in computational as well as memory resources. A tree containing millions, if not billions, of nodes may need to be generated in solving even a moderate sized TSP problem. As nodes are pruned (and hopefully there will be many of these), the facilities of automatic garbage collection is critically important. Storage for these nodes must be reclaimed by the garbage collector. Earlier attempts using languages without automatic garbage collection were not as successful because of heap overflow problems as well as other memory-related problems.

A GUI application was constructed that enables the user to specify the size of a TSP problem and either enter the cost matrix by hand or load an external text file that contains one number per line (c_{11}, c_{12}, …, c_{1n}, c_{21}…, c_{2n}, …) or have the cost matrix determined using random numbers.

A screen shot of the GUI using the 6 city problem discussed earlier is shown below. The entries in each row that comprise a solution are shown in red.

The Console window provides a running account of the progress of the algorithm as well as a useful history that may be reviewed later.


Traveling Salesperson Problem - Branch and Bound by Richard Wiener
Random Numbers
Load File ...
Perform Computation
Save Cost Matrix

Size of Matrix (4-33) 6

Use heuristic ☐

Solution: 1 3 2 5 4 6 1 with cost 15

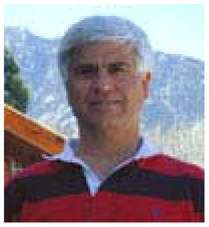
Total number of nodes generated: 31

inf	8	**5**	3	1	2
8	inf	4	9	**2**	8
5	**4**	inf	9	6	7
3	9	9	inf	1	**1**
1	2	6	**1**	inf	9
2	8	7	1	9	inf

The most challenging problem solved to-date using this implementation is a 33 city problem that utilizes real data taken from the Rand McNally Road Atlas for 33 US cities. Running on a 1.7 GHz Pentium 4 machine under Windows 2000 with 256Mbytes of RAM, it took 5637 seconds (1.57 hours) to find the optimum solution of 10861 after generating 872,000 nodes of which 435,729 were pruned. It should be clear from the number of nodes generated on this relatively fast processor that the generation of nodes is computationally expensive. The average number of nodes generated per second was about 154 nodes per second. This is very slow.

The reader must again be reminded that although a 33 city problem may sound like a small problem, there are 32! tours possible so such a problem qualifies as a computationally hard (perhaps intractable) combinatorial optimization problem. It is rather remarkable that any solution at all was found in reasonable time. It should be mentioned that the optimum solution of 10,861 was determined rather early in the computation process (reported in the system Console) even though it took almost two hours to finalize the solution. One might therefore be tempted to use this algorithm as a heuristic approximation in cases where a solution is not finalized in a reasonable period of time. This author has little experience in doing this for larger problem with known solutions. This suggests an area of potentially fruitful research.

About the author



Richard Wiener is Associate Professor of Computer Science at the University of Colorado at Colorado Springs. He is also the Editor-in-Chief of JOT and former Editor-in-Chief of the Journal of Object Oriented Programming. In addition to University work, Dr. Wiener has authored or co-authored 21 books and works actively as a consultant and software contractor whenever the possibility arises.