JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 2, No. 3, May-June 2003

Branch and Bound Implementations for the Traveling Salesperson Problem - Part 2: Single threaded solution with many inexpensive nodes

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

1 BRANCH AND BOUND ALGORITHM 2 FOR TSP WITH ANY COST MATRIX

In the first column published in the March/April, 2003 issue, a branch and bound algorithm that involves generating nodes that represent partial tours with constraints was presented and implemented. The computational cost as well as the memory overhead required to generate each node was shown to be significant. Furthermore, the algorithm assumes that the cost matrix that specifies the cost of traveling from one city to another is symmetric. All of these problems are overcome using the algorithm to be presented and implemented in this column.

An alternative branch-and-bound that works for a cost matrix of any type (no requirement that it be symmetric as is the case for the first algorithm) is based on the following:

At the first level in the tree a node representing the partial tour 1 is constructed. At the next level, nodes representing the partial tours [1, 2], [1, 3], [1, 4] …, [1, n] are generated. At the third level, nodes representing the partial tours [1, 2, 3], [1, 2, 4], [1, 2, 5], …, [1, 2, n], [1, 3, 2], [1, 3, 4], …, [1, 3, n], …, [1, n, 2], [1, n, 3], …, [1, n, n – 1]. This pattern continues until at the lowest level, nodes representing full tours are obtained.

The algorithm proceeds by starting with a root node (level 1) and a partial tour of [1]. Then all the nodes at level 2 are generated. For each of these nodes a lower bound is computed (details to be shown below). The node with the smallest lower bound is used to generate nodes at level 3. After the lower bounds for each of these nodes are computed, level four nodes are generated from the level 3 node with smallest lower bound. This process of rapid descent down the tree continues until nodes that represent full tours at level n are produced. The lowest cost node at level n is used to prune nodes as the depth-

Cite this column as follows: Richard Wiener: "Branch and Bound Implementations for the Traveling Salesperson Problem – Part 2", in *Journal of Object Technology*, vol. 2, no. 3, May-June 2003, pp. 65-76. http://www.jot.fm/issues/issue_2003_05/column7

first algorithm backtracks up the tree and generates more nodes at the lowest level possible.

This algorithm tends to generate many more nodes than the first algorithm but at much lower cost per node. There are no constraints to compute for each node and no constraints to store in each node.

Let us examine the mechanism for computing a lower bound for each node in the tree described above. In any tour the length of an edge taken when leaving a city must be at least as large as the length of the shortest edge emanating from that city. This leads to a branch and bound algorithm as shown with the example that follows.

Consider a 5 city problem with cost matrix as follows (this example is taken from the book *Foundations of Algorithms* by Neapolitan and Naimipour, Heath, 1996):

0	**14**	**4**	**10**	**20**
14	**0**	**7**	**8**	**7**
4	**5**	**0**	**7**	**16**
11	**7**	**9**	**0**	**2**
18	**7**	**17**	**4**	**0**

To compute the lower bound on the root node with partial tour [1], we reason as follows:

The lower bounds on the costs of leaving the five verticies (cities) are:

City 1: minimum(14, 4, 10, 20) = 4
City 2: minimum(14, 7, 8, 7) = 7
City 3: minimum(4, 5, 7, 16) = 4
City 4: minimum(11, 7, 9, 2) = 2
City 5: minimum(18, 7, 17, 4) = 4

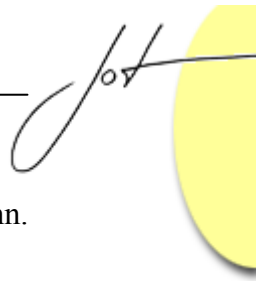
The lower bound is the sum of these minimums which equals 21.

Suppose we consider the process for another more complex node, say the node that represents the partial tour [1, 2, 3].

City 1: 14 (the tour contains 1 -> 2)
City 2: 7 (the tour contains 2 -> 3)
City 3: minimum(7, 16) 7 (cannot touch nodes already on the tour)
City 4: minimum(11, 2) 2 (cannot touch nodes already on the tour)
City 5: minimum(18, 4) 4 (cannot touch nodes already on the tour)

The lower bound is therefore 34.

It should be clear from this example that the cost of computing each lower bound is quite low. Furthermore the storage required in each node is minimal compared to the



storage required using the constrained tour approach presented in the previous column. This offers the hope of generating significantly more nodes per unit time.

A key issue in implementing the branch and bound algorithm outlined above is how to represent the tree structure. Since at a given level nodes may be ranked by their computed lower bound, a priority queue may be used to hold the tree structure. Using such a priority queue, the algorithm may be formulated as follows:

1. Set an initial value for the best tour cost
2. Initialize the priority queue (PQ)
3. Generate the first node with partial tour [1] and compute its lower bound.
4. Insert this node into the PQ
5. while not empty (PQ)
6. remove the first element in the PQ and assign it to parent node
7. if the lower bound < best tour cost
8. set the level of the node to level of parent node + 1
9. if this level equals n – 1 (n being the number of cities)
10. add 1 to the end of the path and compute the cost of the full tour
11. if this cost < best tour cost
12. set the best tour cost and the best tour accordingly
13. else (the level is not equal to n – 1)
14. for all i such that 2 <= I <= n and i is not in the path of parent
15. copy the path from parent to the new node
16. add i to the end of this path
17. compute the lower bound for this new node
18. if this lower bound is less than the best tour cost
19. insert this new node into the priority queue
20. end

This algorithm follows a best-first branch and bound descent of a tree (like the first algorithm). Before a node is inserted into the priority queue it is screened to determine that its lower bound is less than the currently known best tour. This helps to assure that the number of nodes being held in this priority queue is manageable.

What priority rules must the queue enforce? First, nodes at a deeper level (higher level number) have priority over nodes at a shallower level. This assures that the tree grows downwards and that leaf nodes representing complete tours are generated as quickly as possible. In comparing two nodes at the same level, priority is given to the node with the smallest lower bound. In the event of a tie (two nodes with equal lower bound), the sum of the cities in the partial tour is computed. The node with the smaller sum is given a higher priority than the node with the larger sum (a tie cannot occur). It should be evident that the rules just stated disallow two distinct nodes from having an equal priority.

2 IMPLEMENTATION

In searching among the huge number of classes provided in the Java API, there is no class priority queue class. But there is a standard Java collection class that is close in behavior – class *TreeSet*. This collection class orders its elements (which must implement the interface *Comparable*) based on the definition of the *compareTo* method. It is in this method that we encapsulate the rules given above for the priority queue. Listing 1 presents the details of this method *compareTo* (in class *Node*).

Listing 1 – Method compareTo

```
public int compareTo (Object obj) {
       if (this == obj) {
       return 0;
      }
      Node other = (Node) obj;
      if (size < other.size) {
          // size represents # cities in partial tour
          return 1;
      } else if (size > other.size) {
            return -1;
      } else if (size == other.size) {
        if (lowerBound < other.lowerBound) {
            return -1;
         } else if (lowerBound > other.lowerBound) {
            return 1;
         } else if (lowerBound == other.lowerBound) {
            // Add up the sum of the cities
            int sumThis = 0;
            for (int i = 1; i <= size; i++) {
              sumThis += cities[i];
            }
            int sumOther = 0;
            for (int i = 1; i <= size; i++) {
              sumOther += other.cities[i];
            }
            if (sumThis <= sumOther) {
               return -1;
            } else if (sumThis > sumOther) {
               return 1;
            }
        }
    }
    return 100;  // This line can never be reached
}
```

Each of the rules stated above are included in this *compareTo* method. When nodes are added to a *TreeSet*, they will be ordered according to the logic given above.

The entire class *Node* is presented in Listing 2.

Listing 2 – Class Node

```
import java.util.*;
import java.awt.*;
import java.util.*;
import java.io.*;

public class Node implements Comparable, Serializable {
    // Fields
    private int lowerBound;
    private int size; // Number of cities in partial tour
    private byte [] cities; // Stores partial tour
    private boolean [] blocked = new boolean[TSP.numRows + 1];
    private int level; // The level in the tree

    // Constructor
    public Node (byte [] cities, int size) {
        this.size = size;
        this.cities = cities;
    }

    // Commands
    public void computeLowerBound () {
        lowerBound = 0;
        if (size == 1) {
            for (int i = 1; i <= TSP.numRows; i++) {
                lowerBound += minimum(i);
            }
        } else {
            // Obtain fixed portion of bound
            for (int i = 2; i <= size; i++) {
                blocked[cities[i]] = true;
                lowerBound += TSP.c.cost(cities[i - 1], cities[i]);
            }
            blocked[1] = true;
            lowerBound += minimum(cities[size]);
            blocked[1] = false;
            blocked[cities[size]] = true;
            for (int i = 2; i <= TSP.numRows; i++) {
                if (!blocked[i]) {
                    lowerBound += minimum(i);
                }
            }
        }
    }

    public void setLevel (int level) {
        this.level = level;
    }



    public void setCities (byte[] cities) {
        this.cities = cities;
    }

    // Queries
    public int size () {
        return size;
    }
```

```
public byte [] cities() {
    return cities;
}

public int level () {
    return level;
}

public int lowerBound() {
    return lowerBound;
}

public int compareTo (Object obj) {
    if (this == obj) {
        return 0;
    }
    Node other = (Node) obj;
    if (size < other.size) {
        return 1;
    } else if (size > other.size) {
        return -1;
    } else if (size == other.size) {
        if (lowerBound < other.lowerBound) {
            return -1;
        } else if (lowerBound > other.lowerBound) {
            return 1;
        } else if (lowerBound == other.lowerBound) {
            // Add up the sum of the cities
            int sumThis = 0;
            for (int i = 1; i <= size; i++) {
                sumThis += cities[i];
            }
            int sumOther = 0;
            for (int i = 1; i <= size; i++) {
                sumOther += other.cities[i];
            }
            if (sumThis <= sumOther) {
                return -1;
            } else if (sumThis > sumOther) {
                return 1;
            }
        }
    }
    return 100;
}


public boolean equals (Object obj) {
    return this.compareTo(obj) == 0;
}

public String toString () {
    String result = "<";
    for (int i = 1; i < cities.length; i++) {
        result += cities[i] + " ";
    }
    if (cities.length == TSP.numRows) {
        for (int i = 2; i <= TSP.numRows; i++) {
            if (!present((byte) i, cities)) {
```

```
                    result += i + " ";
                    break;
                }
            }
            result += "1>";
        } else {
            result += ">";
        }
        return result;
    }

    private int minimum (int index) {
        int smallest = Integer.MAX_VALUE;
        for (int col = 1; col <= TSP.numRows; col++) {
            if (!blocked[col] && col != index &&
                    TSP.c.cost(index, col) < smallest) {
                smallest = TSP.c.cost(index, col);
            }
        }
        return smallest;
    }

    private boolean present (byte city, byte [] cities) {
        for (int i = 1; i <= cities.length - 1; i++) {
            if (cities[i] == city) {
                return true;
            }
        }
        return false;
    }
}
```

An examination of the field structure of this class reveals how much lighter it is than the corresponding class *Node* in the previous implementation.

Class *TSP*, in Listing 3, implements the algorithm presented earlier.

Methods *read* and *write* support object persistence. This allows a computation session to be ended and the state of the system preserved on disk in a file *OnGoing.data*. When the next session begins this data file may be used to restore the state of the system to where it last was when the previous session ended. Java's object persistence through serializability is used to accomplish this. The details are shown in Listing 3.

Listing 3 – Class TSP

```
// TSP Branch and Bound Main Driver
import java.awt.*;
import java.util.*;
import java.io.*;

public class TSP implements Serializable {

    // Fields
    public static int numRows;
    private int numCols;
    private int bestTour = Integer.MAX_VALUE / 4;
    private Node bestNode;
    private TreeSet queue = new TreeSet();
```

```
public static Cost c;
private long totalNodeCount = 0L;
private double threshold;
private boolean stop = false;
private double elapsedTime = 0.0;

public TSP (int [][] costMatrix, int size, int bestTour, double
            threshold) {
    this.threshold = threshold;
    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 void write(ObjectOutputStream stream) {
    try {
      stream.writeInt(numRows);
      stream.writeInt(bestTour);
      stream.writeLong(totalNodeCount);
      stream.writeObject(bestNode);
      Object [] nodes = queue.toArray();
      stream.writeInt(queue.size());
      for (int i = 0; i < nodes.length; i++) {
          stream.writeObject(nodes[i]);
      }
    } catch (Exception ex) {
        System.out.println(ex);
    }
}

public void read(ObjectInputStream stream) {
    try {
        numRows = stream.readInt();
        bestTour = stream.readInt();
        totalNodeCount = stream.readLong();
        bestNode = (Node) stream.readObject();
        int queueSize = stream.readInt();
        Node [] nodes = new Node[queueSize];
        for (int i = 0; i < queueSize; i++) {
            nodes[i] = (Node) stream.readObject();
            queue.add(nodes[i]);
        }
    } catch (Exception ex) {
        System.out.println(ex);
    }
}

public void stop () {
    stop = true;
}

public void generateSolution (boolean ongoing) {
    TimeInterval t = new TimeInterval();
    t.startTiming();
    if (!ongoing) {
        // Create root node
        byte[] cities = new byte[2];
```

```
        cities[1] = 1;
        Node root = new Node(cities, 1);
        root.setLevel(1);
        totalNodeCount++;
        root.computeLowerBound();
        System.out.println(
            "The lower bound for root node (no constraints): " +
            root.lowerBound());
        queue.add(root);
    }
    while (!stop  && elapsedTime <= threshold &&
            queue.size() > 0) {
        Node next = (Node) queue.first();
        if (next.size() == TSP.numRows - 1 &&
                next.lowerBound() < bestTour) {
            t.endTiming();
            bestTour = next.lowerBound();
            bestNode = next;
            System.out.println("\nNodes generated: " +
                               totalNodeCount);
            System.out.println("Best tour cost: " + bestTour);
            System.out.println("Best tour: " + bestNode);
            System.out.println("queue.size(): " + queue.size());
            double time = t.getElapsedTime();
            elapsedTime = time;
            int hours = (int) (time / 3600.0);
            time -= hours * 3600;
            int minutes = (int) (time / 60.0);
            time -= minutes * 60;
            int seconds = (int) time;
            System.out.println("Elapsed time: " +
                t.getElapsedTime() + " seconds.      <" + hours + "
                hours " + minutes + " minutes " +
                seconds + " seconds>");
        }
        queue.remove(next);
        if (next.lowerBound() < bestTour) {
            int newLevel = next.level() + 1;
            byte [] nextCities = next.cities();
            int size = next.size();

            for (int city = 2; !stop && city <= TSP.numRows;
                    city++) {
                if (!present((byte) city, nextCities)) {
                    byte [] newTour = new byte[size + 2];
                    for (int index = 1; index <= size; index++) {
                        newTour[index] = nextCities[index];
                    }
                    newTour[size + 1] = (byte) city;
                    Node newNode = new Node(newTour, size + 1);
                    newNode.setLevel(newLevel);
                    totalNodeCount++;
                    if (totalNodeCount % 100000 == 0) {
                        System.out.print(".");
                    }
                    if (totalNodeCount % 1000000 == 0) {
                        t.endTiming();
                        System.out.println();
                        System.out.println("\nNodes generated: " +
                            totalNodeCount / 1000000 +
                            " million nodes.");
                        System.out.println("queue.size(): " +
```

```
                        queue.size());
                if (bestTour != Integer.MAX_VALUE / 4) {
                    System.out.println("Best tour cost: " +
                        bestTour);
                    if (totalNodeCount % 10000000 == 0) {
                        System.out.println("Best tour: " +
                            bestNode);
                        System.out.println(
                            "Session ends after " +
                            threshold + " seconds.");
                        System.out.println();
                        System.out.println(
                  "Distribution of Levels In Queue from " +
                  ((Node) queue.last()).level() + " to " +
                  ((Node) queue.first()).level());
                        int[] number =
                            new int[TSP.numRows];
                        for (Iterator iter =
                             queue.iterator();
                             iter.hasNext(); ) {
                            Node n = (Node) iter.next();
                            number[n.level()]++;
                        }
                        System.out.println();
                        for (int i = 2; i < TSP.numRows;
                              i++) {
                            System.out.print(i + ": " +
                                number[i] + " ");
                            if (i % 5 == 0) {
                                System.out.println();
                            }
                        }
                        System.out.println("\n");
                    }
                    double time = t.getElapsedTime();
                    elapsedTime = time;
                    int hours = (int) (time / 3600.0);
                    time -= hours * 3600;
                    int minutes = (int) (time / 60.0);
                    time -= minutes * 60;
                    int seconds = (int) time;
                    System.out.println("Elapsed time: " +
                        t.getElapsedTime() +
                        " seconds. <" +
                        hours + " hours " + minutes +
                        " minutes " + seconds +
                        " seconds>");
                }
            }
            newNode.computeLowerBound();
            int lowerBound = newNode.lowerBound();
            if (lowerBound < bestTour) {
                queue.add(newNode);
            } else {
                newNode = null;
            }
        }
    }
} else {
    next = null;
```

```
            }
        }
        t.endTiming();
        if (elapsedTime >= threshold) {
            System.out.println(
            "\n\nAlgorithm terminated since no new best tour found after " +
            threshold + " new nodes generated.");
            System.out.println(
            "The results should be considered a heuristic approximation.");
        } else {
            System.out.println("\n\nAn optimum tour has been found.");
        }
        double time = t.getElapsedTime();
        elapsedTime = time;
        int hours = (int) (time / 3600.0);
        time -= hours * 3600;
        int minutes = (int) (time / 60.0);
        time -= minutes * 60;
        int seconds = (int) time;
        System.out.println("\n\nCost of tour: " + bestTour +
                           "\nTour: " + bestNode +
                           "\nTotal of nodes generated: " +
                         totalNodeCount);
        System.out.println(
           "Elapsed time: " + t.getElapsedTime() + " seconds.      <" +
           hours + " hours " + minutes + " minutes " + seconds +
           " seconds>");
        System.out.println();
    }

    // Queries
    public long nodesCreated () {
        return totalNodeCount;
    }

    public Node bestNode () {
        return bestNode;
    }

    public int bestTour () {
        return bestTour;
    }

    public long nodesGenerated () {
        return totalNodeCount;
    }

    private boolean present (byte city, byte [] cities) {
        for (int i = 1; i <= cities.length - 1; i++) {
            if (cities[i] == city) {
                return true;
            }
        }
        return false;
    }
}
```

3 EMPIRICAL RESULTS

Like with the first implementation presented in the previous column, a GUI is constructed that enables the user to perform experiments. If the 33 city problem used in the previous column is loaded, the implementation presented above must generate 4.191 billion nodes (4,191,003, 606 nodes) before finalizing the same solution with cost 10,861. It takes the same computer (Pentium 4, 1.7 GHz processor with 256M RAM and running under Windows 2000) 24,170 seconds (6 hours, 42 minutes and 50 seconds) to find this solution. The priority queue size during this long computation never exceeds several hundred nodes – a testimony to the efficacy of pruning. The rate of generation of nodes is about 167,000 nodes per second (significantly faster than the several hundred nodes per second produced using the previous algorithm).

Before reaching the conclusion that this algorithm is less efficient at obtaining a solution than the previously presented solution, let us examine another problem – a 24 city problem. The cost matrix is generated randomly and saved.

Using this algorithm, it takes 4 minutes and 59 seconds and close to 52 million nodes (52,958,736) to finalize a solution. Using the previous algorithm with constrained nodes, it takes 2619 seconds (43.65 minutes) and 588,253 nodes to finalize the same solution. These two data points immediately suggest the desirability of using each of the implementations on a given problem since it is not known in advance which of the two (if either) will find a solution in a reasonable time.

On the same 20 city problem with random costs, the algorithm presented in this column took 22 seconds and 5,041,269 nodes and the node-constrained previous algorithm took 113.8 seconds and 88807 nodes. Once again the algorithm presented above outperformed the previous algorithm this time by a factor of 6 to 1.

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.