JOURNAL OF OBJECT TECHNOLOGY

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

Vol. 2, No. 4, July-August 2003

Branch and Bound Implementations for the Traveling Salesperson Problem - Part 3: Multi-threaded solution with many inexpensive nodes

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

The multi-threaded implementation presented in this column sets the stage for the distributed processing implementation to be presented in the next column.

In the previous column a best-first branch and bound algorithm was introduced and implemented. This algorithm forms the basis for the current work (please review the details of that algorithm before continuing with this paper).

Class *Node* is unchanged from the single-threaded implementation presented in the previous column. A new thread class *ProcessNodes* is introduced and is presented in Listing 1.

Listing 1 – Thread class ProcessNodes

```
import java.util.*;

public class ProcessNodes extends Thread {

    // Fields
    private TreeSet queue;
    public int numRows;
    private int numCols;
    private Node bestNode;
    public Cost c;
    private long totalNodeCount = 0L;
    private boolean stop = false;
    private TSP tsp;
    private int threadNumber;

    // Constructor
    public ProcessNodes (int threadNumber, TreeSet queue, int numRows,
                         Cost c, long totalNodeCount, TSP tsp) {
        this.threadNumber = threadNumber;
        this.queue = queue;
        this.numRows = numRows;
        this.c = c;
        this.tsp = tsp;
    }

    // Commands
    public void remove (Node node) {
        queue.remove(node);
    }

    public void setQueue (TreeSet queue) {
        this.queue = queue;
    }

    public void setStop () {
        stop = true;
    }

    public void run () {
        while (!stop  && queue.size() > 0) {
            Node next = (Node) queue.first();
            if (next.size() == TSP.numRows - 1 && next.lowerBound() <
                    TSP.bestTour) {
                tsp.output2(next, threadNumber, queue, totalNodeCount);
            }
            synchronized (queue) {
                queue.remove(next);
            }
            if (next.lowerBound() < TSP.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) {
                            tsp.output1(threadNumber, queue,
                                        totalNodeCount);
                        }
                        newNode.computeLowerBound();
                        int lowerBound = newNode.lowerBound();
                        if (lowerBound < TSP.bestTour) {
                            synchronized (queue) {
                                queue.add(newNode);
                            }
                        }
                        else {
                            newNode = null;
                        }
                    }
                }
            }
            else {
                next = null;
            }
        }
        if (!stop) {
            tsp.stop(false, threadNumber);
        }
    }

    public TreeSet queue () {
        return queue;
    }

    public long totalNodeCount () {
        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;
    }
}
```

Thread instances are spawned by the revised class *TSP* presented in Listing 2. When a *ProcessNode* thread is spawned, the constructor takes as input a thread number (assigned by a *TSP* object), a priority queue (a single node at level 2 in the tree structure – there are n – 1 such nodes where n is the number of cities in the TSP problem), the number of cities, the cost matrix, the total node count to-date for the thread and an instance of class *TSP* (so the thread can communicate with the *tsp* object).

Listing 2 – Class TSP

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

public class TSP implements Serializable {

    // Fields
    public static int numRows;
    public static int bestTour = Integer.MAX_VALUE / 4;
    public static Node bestNode;
    public static Cost c;
    public static TimeInterval t = new TimeInterval();

    private TSPUI gui;
    private TreeSet queue = new TreeSet();
    private long totalNodeCount = 0L;
    private boolean stop = false;
    private double elapsedTime = 0.0;
    private static int numberThreads = 6;
    private ProcessNodes [] threads = new ProcessNodes[numberThreads];
    private int numberStopped = 0;
    private double accumulatedTime = 0.0;

    public TSP (int [][] costMatrix, int size, int bestTour,
                TSPUI gui) {
        this.gui = gui;
        this.bestTour = bestTour;
        numRows = size;
        c = new Cost(numRows, numRows);
        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 {
          // Save queue
          stream.writeInt(queue.size());
          Object [] nodes = queue.toArray();
          for (int i = 0; i < nodes.length; i++) {
              stream.writeObject(nodes[i]);
          }
          stream.writeDouble(t.getElapsedTime() + accumulatedTime);
          stream.writeInt(numberThreads);
          stream.writeInt(numRows);
          stream.writeLong(totalNodeCount);
          stream.writeInt(bestTour);
          stream.writeObject(bestNode);
          for (int threadNumber = 0; threadNumber < numberThreads;
                   threadNumber++) {
              stream.writeInt(threads[threadNumber].queue().size());
              nodes = threads[threadNumber].queue().toArray();
              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 {
          int queueSize = stream.readInt();
          Node [] nodes = new Node[queueSize];
          TreeSet queue = new TreeSet();
          for (int i = 0; i < queueSize; i++) {
              nodes[i] = (Node) stream.readObject();
              queue.add(nodes[i]);
          }
          accumulatedTime = stream.readDouble();
          numberThreads = stream.readInt();
          threads = new ProcessNodes[numberThreads];
          numRows = stream.readInt();
          totalNodeCount = stream.readLong();
          bestTour = stream.readInt();
          bestNode = (Node) stream.readObject();
          TreeSet [] queues = new TreeSet[numberThreads];
          for (int threadNumber = 0; threadNumber < numberThreads;
                   threadNumber++) {
              queueSize = stream.readInt();
              nodes = new Node[queueSize];
              queue = new TreeSet();
              for (int i = 0; i < queueSize; i++) {
                  nodes[i] = (Node) stream.readObject();
                  queue.add(nodes[i]);
              }
              threads[threadNumber] =
                  new ProcessNodes(threadNumber + 1,
                      queues[threadNumber] = new TreeSet(queue),
                          numRows, c, 0, this);
          }
      } catch (Exception ex) {
          System.out.println(ex);
      }
  }

  public synchronized void output1 (int threadNumber,
                                    TreeSet queue,
                                    long totalNodeCount) {
```

```
        System.out.println();
        System.out.println("Thread number: " + threadNumber);
        TSP.t.endTiming();
        double time = TSP.t.getElapsedTime();
        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: " +
        TSP.t.getElapsedTime() + " seconds.      <" +
        hours + " hours " + minutes + " minutes " +
        seconds + " seconds>");

        System.out.println("Nodes generated: " +
        totalNodeCount / 1000000 +
        " million nodes.");
        System.out.println("queue.size(): " +
        queue.size());
        if (TSP.bestTour != Integer.MAX_VALUE / 4) {
            System.out.println("Best tour cost: " +
            TSP.bestTour);
            System.out.println("Best tour: " + TSP.bestNode);
            System.out.println();
        }
    }

    public synchronized void output2 (Node next,
                                      int threadNumber,
                                      TreeSet queue,
                                      long totalNodeCount) {
        int bestTour = next.lowerBound();
        bestNode = next;
        if (bestTour < this.bestTour) {
            setBestTour(bestTour);
            setBestNode(bestNode);
            System.out.println();
            System.out.println("\nThread number: " + threadNumber +
                               " improves best score.");
            t.endTiming();
            double time = t.getElapsedTime();
            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: " + TSP.t.getElapsedTime() +
                               " seconds.      <" + hours + " hours " +
                                minutes +
                               " minutes " + seconds + " seconds>");
            System.out.println("Nodes generated: " + totalNodeCount);
            System.out.println("Best tour cost: " + bestTour);
            System.out.println("Best tour: " + bestNode);
            System.out.println("queue.size(): " + queue.size());
            System.out.println();
        }
    }




    public synchronized void stop (boolean forced, int threadNumber) {
        if (forced && stop) {
            return;
        }
        if (queue.size() == 0) {
            numberStopped++;
        } else if (!forced) {
            TreeSet t = new TreeSet();
            Node n = (Node) queue.first();
            t.add(n);
            long totalNodeCount =
                threads[threadNumber - 1].totalNodeCount();
            threads[threadNumber - 1] =
                new ProcessNodes(threadNumber, t, numRows, c,
                                 totalNodeCount, this);
            threads[threadNumber - 1].start();
            synchronized (queue) {
                queue.remove(n);
            }
        }
        if (numberStopped == numberThreads || forced) {
            stop = true;
            for (int i = 0; i < numberThreads; i++) {
                threads[i].setStop();
            }
            t.endTiming();
            // Count the total number of nodes generated from the
            // threads
            long nodesGenerated = 0;
            for (int i = 0; i < numberThreads; i++) {
                nodesGenerated += threads[i].totalNodeCount();
            }
            totalNodeCount += nodesGenerated;
            if (!forced) {
                System.out.println("Optimum solution obtained.");
            } else {
                System.out.println(
        "Solution forced to stop prematurely and may not be optimum.");
            }
            System.out.println(
                "The total number of nodes generated: " +
                 totalNodeCount);
            System.out.println("Tour cost: " + bestTour);
            double time = TSP.t.getElapsedTime() + accumulatedTime;
            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: " + (TSP.t.getElapsedTime() +
                accumulatedTime) +
                " seconds.      <" + hours + " hours " + minutes +
                " minutes " + seconds + " seconds>");
            gui.displayOutput();
        }
    }

    public void setBestTour (int bestTour) {
        if (bestTour < this.bestTour) {
            this.bestTour = bestTour;
        }
    }

    public void setBestNode (Node bestNode) {
        this.bestNode = bestNode;
    }

    public void generateSolution (boolean ongoing) {
        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);
            Node next = (Node) queue.first();
            synchronized (queue) {
                queue.remove(next);
            }
            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++;
                    newNode.computeLowerBound();
                    int lowerBound = newNode.lowerBound();
                    queue.add(newNode);
                }
            }
            // Spawn the threads and start the process going
            for (int i = 0; i < numberThreads; i++) {
                TreeSet t = new TreeSet();
                Node n = (Node) queue.first();
                t.add(n);
                threads[i] = new ProcessNodes(i + 1, t, numRows, c,
                                              0L, this);

                synchronized (queue) {
                    queue.remove(n);
                }
            }
        }
        for (int i = 0; i < numberThreads; i++) {
            threads[i].start();
        }
    }

    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;
    }
}
```

In method *generateSolution*, the root node and all its children (at level 2) are generated before any threads are spawned. Then the following segment of code is used to create the threads:

```
// Spawn the threads and start the process going
for (int i = 0; i < numberThreads; i++) {
     TreeSet t = new TreeSet();
     Node n = (Node) queue.first();
     t.add(n);
     threads[i] = new ProcessNodes(i + 1, t, numRows, c,
                                  0L, this);

     synchronized (queue) {
         queue.remove(n);
     }
 }
```

Each thread is handed a child node (partial tour consisting of node 1 as well as another city). The child nodes at level 2 are prioritized based on their lower bound (the smallest lower bound having the highest priority). The threads are held in an array of *ProcessNode* (called *threads*). The *thread[0]* starts with a child node of smallest lower bound. The *thread[1]* has the second lowest lower bound. There is of course no guarantee that the solution exists under any of the initial level 2 nodes handed off to the various threads (6 are shown by default). When a thread completes the processing of the node that it was handed (this will probably require the generation of millions of nodes), it sends the tsp object a *stop* command. If the pool of available *tsp* nodes is depleted (queue size of size 0), the value of the field *numberStopped* is incremented by one (see method *stop* in class *TSP*). Otherwise a new node from among the original level 2 nodes is handed off to a fresh thread that is spawned (recall that once a thread is terminated it cannot be re-started) and removed from the *tsp* queue. The application terminates when all the level 2 nodes in the *tsp* object have been removed (and handed off to threads) and all the threads have terminated (the value *numberStopped* is equal to the number of threads).

Empirical Results

We consider some results run on two computers: a single-processor Pentium 4 with a 1.7Ghz processor and a dual G4 processor Powermac running under Mac OS 10.2.3. It is interesting to see how the Powermac with its dual processors is able to take advantage of the multiple threads. JDK 1.4.1 is used on each of the machines (a beta release on the Powermac).

The results are summarized in the table below.

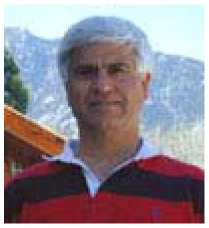
Comparison of Single-Threaded vs. Mult-Threaded TSP Implementations on a Single vs. Dual Processor Machines

Machine	**Number of Cities**	**Execution Time** (Single Threaded Previous Implementation)	**Execution Time** (Using 6 threads with current implementation)
Pentium 4, 1.7 GHz	**20**	**21.801** seconds	**19.358** seconds
Powermac (Dual g4 processors)	**20**	**30.597** seconds	**16.486** seconds
Pentium 4, 1.7GHz	**24**	**299.25** seconds	**343.944** seconds
Powermac (Dual G4 processors)	**24**	**410.984** seconds	**276.729** seconds

Several conclusions may be gleaned from the results in the above table.

1. The Mac OS is able to take good advantage of multiple threads by efficiently dispatching threads to each of its two processors. For both TSP problems the Mac's multi-threaded solution was significantly faster than the single-threaded solution. In fact for both problems the Powermac's multi-threaded solution was the fastest among the two computers even allowing for the fact that the Pentium 4 machine has a clock speed that is 1.36 times faster than the Powermac's clock speed.
2. The execution time for the single-threaded implementation on the two computers is close to the ratio of their respective clock speeds. This suggests that the Powermac is not able to take advantage of its dual processors in the single-threaded implementation.
3. The multi-threaded solution on the Pentium 4 machine was slightly faster for the 20 city problem and significantly slower for the 24 city problem. This clearly suggests (as should be fairly obvious) that the problem instance itself affects the ability of the multi-threaded solution to take advantage of the input data. This is further reflected in the relative improvement that the multi-threaded solution on the Powermac displayed for the 20 and 24 city problems. The degree of improvement was smaller for the 24 city problem than the 20 city problem where the execution time was almost cut in half.

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.