JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 1, January-February 2005

A Java Implementation of the Branch and Bound Algorithm: The Asymmetric Traveling Salesman Problem

Pawel Kalczynski, University of Toledo, USA

Abstract

This paper offers a description of a Java implementation of the branch-and-bound (BnB) algorithm for the Traveling Salesman Problem with asymmetric cost matrix (ATSP). A generic interface for solving minimization problems with BnB is proposed and the results of computational experiments for ATSP with random cost matrices are given for different problem sizes (50, 100, 150, 200, 250, and 300 cities).

1 INTRODUCTION

Branch and bound (BnB) is a set of enumerative methods applied to solving discrete optimization problems. The original problem, also referred to as a "root problem" is bounded from below and above. If the bounds match, the optimal solutions have been found. Otherwise the feasible region i.e., the space in which the argument of the problem function f(x) is confined by explicit constraints, is partitioned into subregions. The subregions constitute feasible regions for subproblems, which become children of the root problem in a search tree. The principle behind creating relaxed subproblems (relaxations) of the original problem, the process also known as "branching," is that unlike the original problem, the relaxations can be solved within a reasonable amount of time. If a subproblem can be optimally solved, its solution is a feasible, though not necessarily optimal, solution to the original problem. Therefore, it provides a new upper bound for the original problem. Any node of the search tree with a solution that exceeds the global upper bound can be removed from consideration, i.e., the branching procedure will not be applied to that node. The tree is searched until all nodes are either removed or solved. BnB is guaranteed to reach the optimal solution, provided that it exists.

The Traveling Salesman Problem (TSP) is a graph theory problem of finding the shortest path a salesman can take through each of n cities visiting each city only once. This path is also referred to as the most efficient Hamiltonian circuit.

Cite this article as follows: Pawel Kalczynski: "A Java Implementation of the Branch and Bound Algorithm: the Asymetric Traveling Salesman Problem", in *Journal of Object Technology*, vol. 4, no. 1, January-February 2005, pp. 155-163. http://www.jot.fm/issues/issue_2005_01/article5

In the traditional TSP, the cost of traveling (e.g. distance) between any two cities does not depend on the direction of travel. Hence, the cost (distance) matrix C(n×n) representing the parameters of the problem is symmetric, i.e., the elements of the matrix $c_{ij} = c_{ji}$ for all i=1,…,n and j=1,…,n.

A generalized version of the TSP, known as Asymmetric TSP or ATSP, assumes the asymmetric cost matrix. To illustrate the practical application of ATSP let us imagine a mailman who works in the mountains and must visit all his customers in the shortest time. Each element of the cost matrix c_{ij} represents the time it takes to get from home i to home j. Depending on whether he goes uphill or downhill, the traveling time between the two homes may be significantly different.

Solving larger instances of TSP or ATSP optimally has fascinated researchers since the early computers first appeared at universities. Because the problem is combinatorially explosive in nature (there is n! possible solutions), efficient mathematical models had to be developed first.

The application of BnB to TSP was originally proposed by Little et al. [Little1963]. Later, the techniques of patching cycles have been refined by many researchers, including Zhang, whose BnB algorithm [Zhang1993] is the most efficient, to the best of our knowledge. An excellent overview of the existing heuristics for ATSP can be found in [Johnson2002]

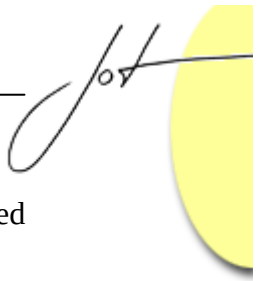
This paper revisits some 20-year-old algorithms and contributes to the field of object technology by offering a description of the model and implementation of a generic package for solving discrete minimization problems with the branch and bound method. In particular, the package is implemented for and tested on the Traveling Salesman Problem with asymmetric cost matrices.

The following section of this paper offers the description of the generic model of the branch and bound method. Section 3 presents the BnB framework for solving ATSP. Section 4 contains computational results for different problem sizes of ATSP followed by a brief summary.

2 BRANCH AND BOUND PACKAGE

The proposed Branch and Bound Package implemented with Java can be used to solve various discrete minimization problems. The package was modeled according to the general description of BnB [Balas1985] adopted to the OOP approach.

Let P denote any (abstract) problem. Let P_i.value denote the optimal solution to the instance P_i of problem P. Further, let R be the relaxation of P such that the R_i.value bounds P_i.value from below. It is important that R_i.value may be computed (or approximated from below) at all stages. Further, let us define a branching rule for breaking up the feasible set of the currently analyzed instance of the relaxation R_i.



Finally, let us define a rule for choosing the next relaxed problem to be processed (selection rule). Then the branch and bound algorithm [Balas1985] is given by:

Step 1. Put P_i on the list of active problems. Initialize the upperbound at $u=\infty$. Set CurrentBestSolution = null.

Step 2. If the list is empty, stop: the solution associated with u is optimal (or if $u=\infty$, P_i has no solution). Otherwise choose a subproblem P_i according to the subproblem selection rule and remove P_i from the list.

Step 3. Solve the relaxation R_i of P_i and let $l_i=R_i$.value. If $l_i \geq u$ then return to Step 2. If $l_i<u$ and the solution of R_i is also a valid solution of P_i then set CurrentBestSolution=R_i.solution and u=R_i.value and goto Step 5.

Step 4. (optional) Use a heuristic to make R_i.solution feasible for P_i. If the value found is lower than u then set CurrentBestSolution=R_i.solution and u=R_i.value.

Step 5. Apply the branching rule to P_i, i.e. generate new subproblems P_{i1}, P_{i2}, … , P_{iq} place them on the list and go to Step 2.

In general, the better (more constrained) the relaxation R and the branching method, the better is the performance of the branch and bound method. Simple approaches prove inefficient for larger instances of TSP (see [Wiener2003] for instance).

Based on the above algorithm we propose an object model of a generic branch and bound minimization method based on two classes i.e., `BnB` and `OptimizationProblemComparson`, and two interfaces i.e., `OptimizationProblem` and `OptimizationProblemSolution`. Figure 1 presents the UML diagram of the model with its ATSP extension.

`OptimizationProblem` interface is the central part of the package. In the proposed model all problems to be solved, i.e., the original (root) problem and its relaxations, are required to implement this interface. The methods do not need further explanation. Listing 1 presents the code of the interface.

Listing 1. OptimizationProblem Interface

```
public interface OptimizationProblem
{
   public OptimizationProblem getRelaxation();
   public int getProblemSize();
   public double getValue();
   public OptimizationProblemSolution getSolution();
   public boolean isValid(OptimizationProblemSolution ops);
   public OptimizationProblem[] branch();
   public void performUpperBounding(double upperbound);
} // of interface
```

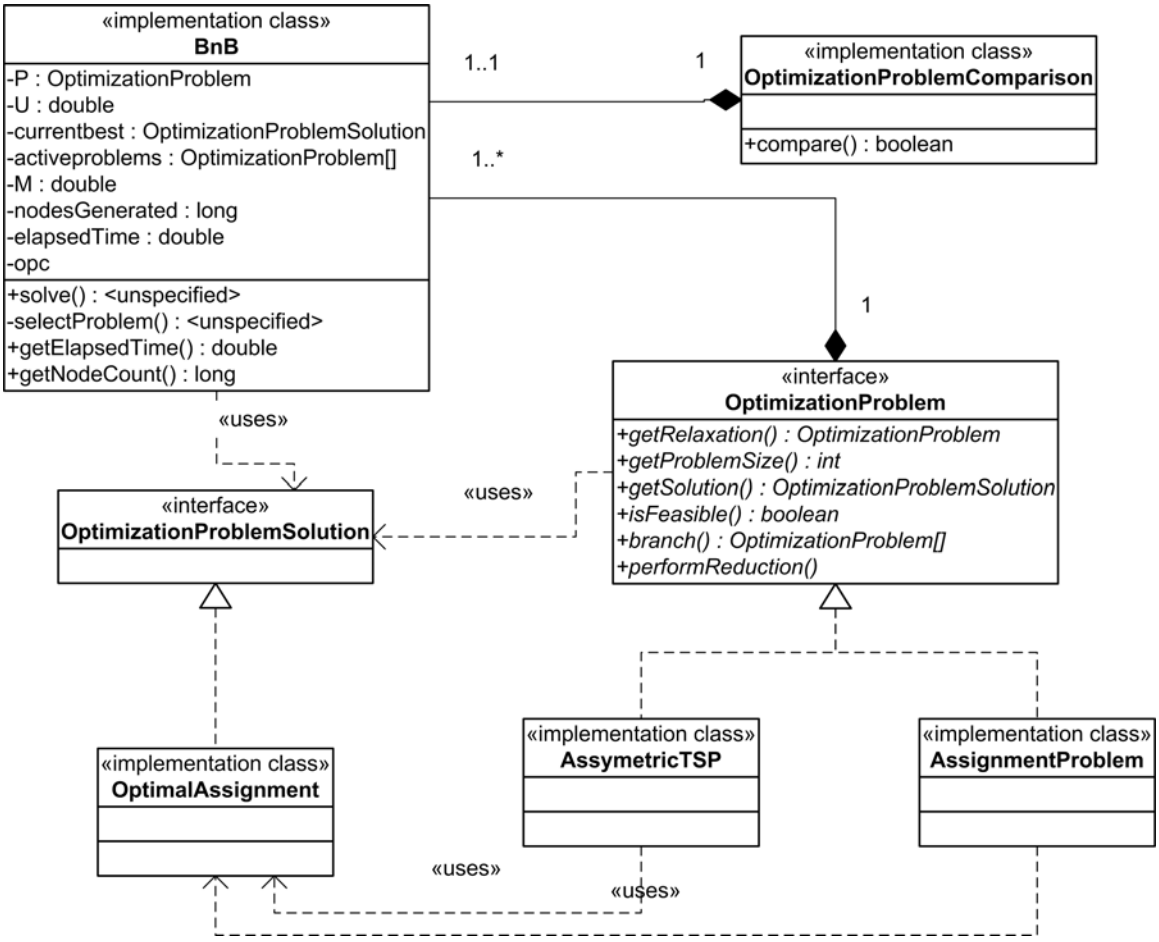
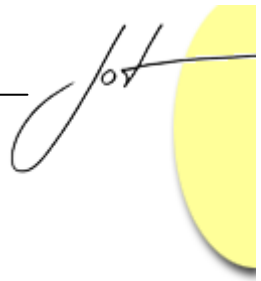


Figure 1. UML Class Diagram for BnB for ATSP

`OptimizationProblemSolution` is a "dummy" interface that does not contain any methods. It was introduced for the purpose of consistency of the package. The solution to the optimization problem may have different forms. For example, in the case of TSP or ATSP it is an array of integers representing the optimal assignment. Hence the object that contains the solution to the specific problem must implement `OptimizationProblemSolution`.

The Java code for BnB class is listed in Appendix 1. The private method selectProblem() uses the `OptimizationProblemComparison` object to compare two optimization problems and sort the `activeproblems` Vector. Because the `Comparison` interface implemented by the `OptimizationProblemComparison` is not part of standard Java code we show the code of `OptimizationProblemComparison` in Listing 2, in order to enable the reader to implement its counterpart for a similar vector-sorting class, if Microsoft's `util` package is not available.



Listing 2. OptimizationProblemComparison Class

```
public class OptimizationProblemComparison
   implements com.ms.util.Comparison
{
   public int compare(Object problem1, Object problem2){
         OptimizationProblem p1 =
                        (OptimizationProblem)problem1;
         OptimizationProblem p2 =
                        (OptimizationProblem)problem2;
         if(p1.getValue()<p2.getValue()) return 1;
         if(p1.getValue()>p2.getValue()) return -1;
         return 0;
   }
} // of class
```

3 BRANCH AND BOUND FOR ATSP

The AP may be solved in polynomial time $O(n^3)$ by a well-known Hungarian algorithm[Carpaneto1980a]. Because the branching scheme used in this implementation forces some arcs to the solution and excludes others, the original assignment problem is converted to the Modified Assignment Problem (M.A.P.), which can be solved with the Hungarian algorithm provided some changes are (temporarily) applied to the cost matrix. In particular [Carpaneto1980b], in order to:

- force inclusion of arc (i,j) to the solution, all elements in jth column of matrix C (except for item in row i) are substituted with a very large number
- force exclusion of arc (i,j) from the solution, the element c_{ij} is substituted with a very large number.

Because solving ATSP with BnB requires finding multiple solutions to the Modified Assignment Problem, efficient implementation of M.A.P. is a critical issue. The modification proposed by Carpaneto and Toth [Carpaneto1980b] decreases the complexity of solving M.A.P to $O(n^2)$ in approximately 40% of cases analyzed. However, in order to keep the BnB class generic, we used the original Hungarian algorithm with the modified cost matrix in the implementation.

4 COMPUTATIONAL EXPERIMENTS

Without the loss of generality, we assume that M (a large positive number) is at each element of the diagonal of the cost matrix C. This assumption ensures that the relaxed problems (M.A.Ps) are more constrained, and this makes BnB converge faster.

For the purpose of the experiment, the elements of the cost matrix C are real numbers randomly selected from [0,10] with the Ranlux random number generator [James1996]. All computations are performed on a single-processor PC (Intel Celeron

1.8MHz, 256MB RAM). The computational results presented in **Table 1** were averaged over 100 solved instances.

Table 1. Computational Results for ATSP

n	avg (max) time [s.]	avg (max) M.A.P. calls
50	0.3 (1.3)	44 (197)
100	6.1 (32.9)	93 (538)
150	44.8 (147.4)	191 (853)
200	150.1 (925.6)	261 (1245)
250	327 (3027.8)	285 (3212)
300	606.6 (3251.7)	304 (2382)

Figure 2 illustrates an attempt to describe the time complexity of the proposed method with a function. Although the problem is NP-hard, the average computing time appears to increase with the square function of n.

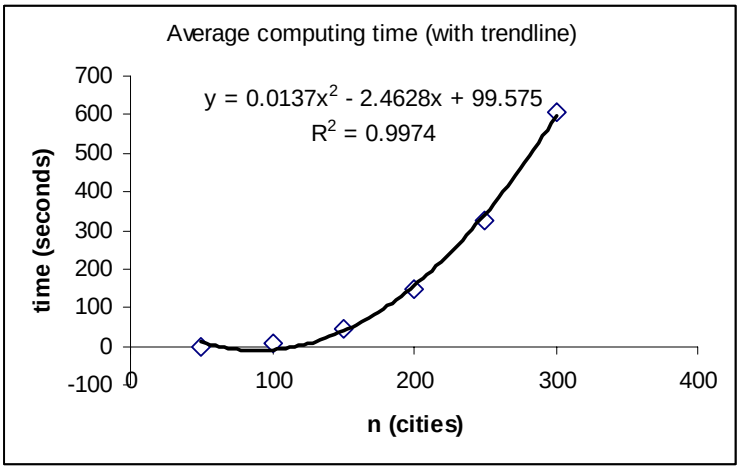
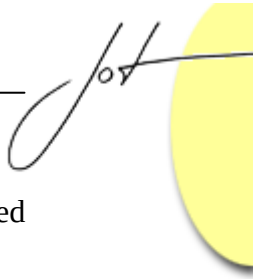


Figure 2. Average ATSP Computing Time

Notice that the Assignment Problem serves as a good lower bound for the ATSP. However, it is a poor relaxation for the traditional (symmetric) TSP [Balas1985].

5 SUMMARY

The original implementation of the presented BnB method for ATSP was done with Fortran back in the 1980s. With object oriented features offered by contemporary



programming environments, such as Java, the implementation becomes self-contained and sharable, still remaining very efficient even on a personal computer.

It is relatively easy to extend the presented single-threaded version to a multi-threaded one, since the only shared-for-write element is the `activeproblems` Vector in the BnB class. This is because each instance of M.A.P. stores its local copy of the (modified) cost matrix. For the maximum problem size that can be potentially handled with the proposed implementation, the amount of memory allocated for data is not a critical issue.

Finally, the proposed implementation of BnB may be applied to other problems, not necessarily related to TSP.

REFERENCES

[Balas1985] E. Balas, P. Toth: *Branch and Bound Methods*, in *The Traveling Salesman Problem*, E.L. Lawler, et al., Editors. 1985, John Wiley & Sons Ltd.: Chichester. p. 361-401.

[Carpaneto1980a] G. Carpaneto, P. Toth: "Solution to the Assignment Problem", *ACM Transactions on Mathematical Software*, vol. 6, pp. 104-111, 1980a.

[Carpaneto1980b] G. Carpaneto, P. Toth: "Some New Branching and Bounding Criteria for the Asymmetric Travelling Salesman Problem", *Management Science*, vol. 26, no. 7, pp. 736-743, 1980b.

[James1996] F. James: "RANLUX: a Fortran implementation of the high-quality pseudorandom number generator of Luscher", *Computer Physics Communications*, vol. 97, no. 3, pp. 357, 1996.

[Johnson2002] D.S. Johnson, et al.: *Experimental Analysis of Heuristics for the ATSP*, *Chapter 10*, in *The Traveling Salesman Problem and its Variations*, G. Gutin and A.P. Punnen, Editors. 2002, Kluwer Academic Publishers: Dordrecht.

[Little1963] J.D.C. Little, et al.: "An algorithm for the traveling salesman problem", *Operations Research*, vol. 11, pp. 972-989, 1963.

[Wiener2003] R. Wiener: "Branch and Bound Implementations for the Traveling Salesperson Problem Part 2: Single threaded solution with many inexpensive nodes", *Journal of Object Technology*, vol. 2, no. 3, pp. 65-76, May-June 2003. http://www.jot.fm/issues/issue_2003_05/column7

[Zhang1993] W. Zhang: "Truncated branch-and-bound: A case study on the asymmetric traveling salesman problem", *Proceeding of the AAAI 1993 Spring Symposium on AI and NP-Hard Problems*. Stanford, CA, 1993

About the author

Dr. **Pawel J. Kalczynski** is an Assistant Professor of Information Systems in the College of Business Administration at the University of Toledo. He is a co-author of the monograph "Filtering the Web to Feed Data Warehouses," Springer-Verlag London, 2002. He can be reached at Pawel.Kalczynski@utoledo.edu.

APPENDIX 1

```
import java.util.*;
import com.ms.util.VectorSort;

public class BnB
{
   private OptimizationProblem P;
   private double U;
   private OptimizationProblem currentbest=null;
   private Vector activeproblems;

   static double  M = Double.MAX_VALUE/1000;
   private long nodesGenerated = 0;
   private double elapsedTime = 0;
   private OptimizationProblemComparison opc;

   public BnB(OptimizationProblem problem){
         this.P = problem;
         int n = P.getProblemSize();
         activeproblems = new Vector(n*n,n);
         activeproblems.addElement(P);
         U = M;
         this.opc = new OptimizationProblemComparison();
   }

   public OptimizationProblem solve(){
         OptimizationProblem Pi;
         OptimizationProblem Ri;
         double Li;
         Date d0 = new Date();

         while(activeproblems.size()>0){
               Pi = selectProblem();
               Ri = Pi.getRelaxation();
               Li = Ri.getValue();
               if(Li<U){
```

```
                    if(P.isValid(Ri.getSolution())){
                          U = Li;
                          this.currentbest = Ri;
                    } else {
                          // optional upper bounding
                          Ri.performUpperBounding(U);

                          // Branching
                          OptimizationProblem[] subPs =
                                                Ri.branch();
                          for(int k=0;k<subPs.length;k++){
                    this.activeproblems.addElement(subPs[k]);

                                this.nodesGenerated++;
                          } // of for(k)
                    }
               } // of if better lower bound
          } // of while(non-empty)

          Date d1 = new Date();
          this.elapsedTime =
                    (double)(d1.getTime()-d0.getTime())/1000;
          return currentbest;
     } // of solve

     private OptimizationProblem selectProblem(){
          OptimizationProblem selected;

          //Sort the vector by the value
          VectorSort.sort(this.activeproblems, opc);

          //Select the best element and remove it from the list
          selected =
     (OptimizationProblem)this.activeproblems.lastElement();

     this.activeproblems.removeElementAt(this.activeproblems.size
  ()-1);

          return selected;
     } // of selectProblem()

     public double getElapsedTime(){
          return this.elapsedTime;
     }

     public long getNodeCount(){
      return this.nodesGenerated;
     }

  } // of class
```