An Empirical Analysis of Approximation Algorithms for Euclidean TSP

Bárbara Rodeker[1], **M. Virginia Cifuentes**[1,2], **and Liliana Favre**[1,2]
[1]Universidad Nacional del Centro de la Provincia de Buenos Aires, Tandil, Argentina
[2]Comisión de Investigaciones Científicas de la Provincia de Buenos Aires, Argentina

Abstract - *The Traveling Salesman Problem (TSP) is perhaps the most famous optimization problem in the set NP-hard. Many problems that are natural applications in computer science and engineering can be modeled using TSP and therefore, researchers are searching implementations focusing on the quality of the produced solution and their computational time. An innovative Polynomial-Time Approximation Scheme (PTAS) for the Euclidean TSP was given by Sanjeev Arora. To date, there is no evidence that it can be implemented to be practically useful. In light of this, we propose an implementation of the Euclidean TSP that is based on the essential steps of the Arora's PTAS and adds some heuristics to improve the running time. The paper includes the description of a C++ Arora-based implementation and a comparative analysis of this implementation with various competitive algorithms for the Euclidean TSP.*

Keywords: Geometric Optimization, Approximation Algorithms, Heuristic Algorithms, Traveling Salesman Problem (TSP).

1 Introduction

The Traveling Salesman Problem (TSP) is perhaps the most well known combinatorial optimization problem in the set NP-Hard. The significance of TSP is that includes many problems that are natural application in computer science and engineering.

There are different variations of TSP. The book "The Traveling Salesman Problem and Its Variations" provides the state of the art of TSP up to 2004 [6]. Several works are linked to a restricted version of TSP, for instance, [1] [2] [3] and [4] propose approximation algorithms for geometric TSP in general, and Euclidean TSP in particular.

Given n nodes in R^2 (more generally in R^d), TSP is to find the minimum length path that visits each node exactly once. If distance is computed using distance between nodes then the problem is called Euclidean TSP. This problem can be used as a model for the study of general methods that can be applied to a wide range of geometric optimization problems.

An innovative Polynomial-Time Approximation Scheme (PTAS) for the Euclidean TSP was discovered by Arora [2]. The main idea is to recursively divide the problem into weakly dependent subproblems that are solved bottom-up, using dynamic programming to find a tour.

Arora's PTAS is asymptotically the most efficient known PTAS, but have never been used in practice due to practical implementation aspects. There is no evidence that it can be implemented to be competitive with other algorithms such as Lin-Kernighan algorithm that produces good quality solutions in near lineal time [5]. In light of this, this paper describes an implementation of the Euclidean TSP that is based on the essential steps of Arora´s PTAS and adds some heuristics to improve the running time. Next, we describe the essential steps in the context of our approach.

First we perform a simple perturbation of the input instance. A proposed regular grid classifies geometrically nodes in the plane. The cell size c is the minimum distance between nodes. In that case, the node coordinates and cell size are modified by a factor equal to 2c to ensure that the minimum distance is two. Finally, to avoid overlaps in the next phase, a new shift place coordinates in odd positions. Following, the dissection step divides the bounding box: the smallest square containing all nodes whose size is 2k. The successive subdivisions generate a quadtree hierarchy until every quadtree region has only one node inside.

Next, the adjacent quadtree regions are communicated via portals. The portals for a dissection are a set of points on the square edges. On each edge a square has at least one portal to communicate with every adjacent terminal square (neighbor). This portals configuration improves the execution time. Finally the path is computed using dynamic programming and the reconstruction is made in the path´s trimming step, starting from the information registered by the dynamic programming table.

A TSP implementation in C++ is presented and discussed along with some results on several moderate sized problems. We have compared the performance of our implementation with other algorithms included in Concorde Solver Software [7]. For comparative analysis we have used TSP instances from TSPLIB [8].

This paper has the following structure. Section 2 describes the proposed variant of the Arora´s PTAS for TSP. Section 3 describes the design of a TSP software based on Arora´s proposal. Section 4 includes a comparative analysis between our implementation and those implemented in the

Concorde software. Finally Section 5 highlights conclusions and future work.

2 Our Arora-based algorithm

Arora discovered the first solution to find PTAS for the Euclidean TSP. Given n nodes in R^2, and for every fixed c > l, a randomized version of the scheme finds a (1+1/c)-approximation to the optimum traveling salesperson tour in $O(n (\log n)^{O(c)})$ time[2]. The main idea of this approach is to divide recursively the plane, and next apply dynamic programming to find a tour that crosses each of the partition at most O(c) times.

Arora´s PTAS algorithm is based on a theorem called "Structure Theorem"[2]:

"Let the minimum nonzero internode distance in a TSP instance be 8 and let L be the size of its bounding box. Let shifts 0 ≤ a, b ≤ L be picked randomly. Then with probability at least 1/2, there is a salesman path of cost at most (1 + 1/c)-OPT that is (m, r)-light with respect to the dissection with shift (a, b), where m = O(c log L) and r = O(c)"

Although our implementation includes the main steps of the Arora´s PTAS, each of them has been modified at implementation level focusing on the computational time. Like Arora´s PTAS, our algorithm includes the following steps: perturbation, random partition, portal-respecting tour, dynamic programming and tour reconstruction. Next, we describe in detail the essential steps.

2.1 Perturbation

At the Arora's perturbation step, the instance must be prepared to reflect the structure theorem. Perturbation ensures that each node lies on the unit grid (i.e has integer coordinates) and every minimum internode distance is 2. The smallest axis-parallel square containing the nodes is called *bounding box*. The perturbation will ensure that its sidelength is at most $n^2/2$. The partitioning is defined using some randomized variant of quadtree.

We can observe empirically that computation does not improve starting from a threshold u, then values of c > u are not convenient due to both the size of the bounding box and the number of squares in the quadtree must be increased influencing the efficiency of the following steps. The constant that reflects implementation factors in the big-oh expressions is so large in comparison to the theoretical advantages of selecting c > u.

Our implementation proposes to classify geometrically n nodes through a regular grid composed by square cells whose side length is equal to the minimum distance among nodes. In that case, the node coordinates and cell size are modified by a factor equal to 2c to ensure that the minimum distance is two. Finally, to avoid overlaps in the next phase, a new shift places coordinates in odd positions. Fig. 1 and Fig. 2 summarize the differences between the perturbation proposed by Arora and our implementation respectively regarding the steps described above.

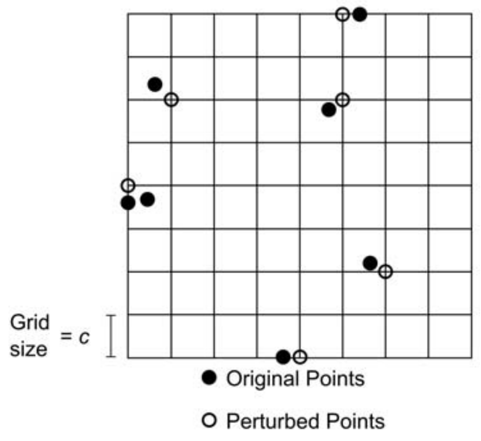


Figure 1. Arora´s PTAS perturbation

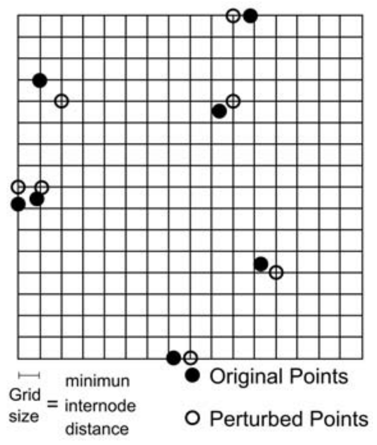


Figure 2. Arora-based perturbation

2.2 Shifted Quadtree

At the Arora's dissection step, a recursive partition of the bounding box is made using a randomized variant of quadtree. Random integer values for a and b are chosen, and a (a,b)-shift of the bounding box is defined by shifting x- and y-coordinates by a and b respectively. The shifted quadtree is created during the dissection of the bounding box (Fig.3).

Dissection step ends when each square has a size <=1, which means that each square will have at most one node inside. No node is in edges because their coordinates have been scaled by a factor of two, so nodes have odd coordinates while partition lines have even coordinates. There are $O(L^2)$ squares in the partition and the maximum number of level is log (L) Shift of the implementation chooses a and b values equals to one unit (see Fig. 4).

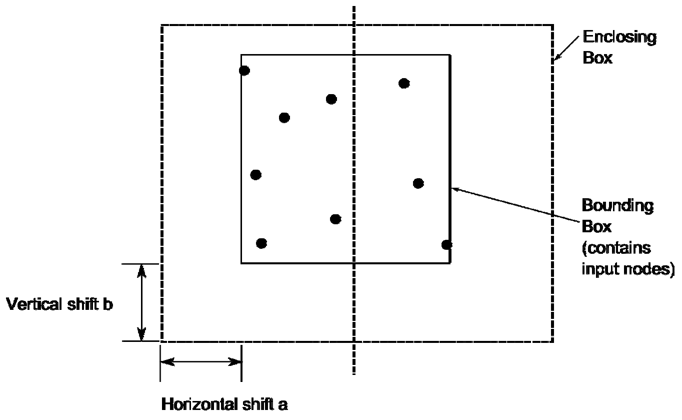


Figure 3. Arora´s PTAS shift

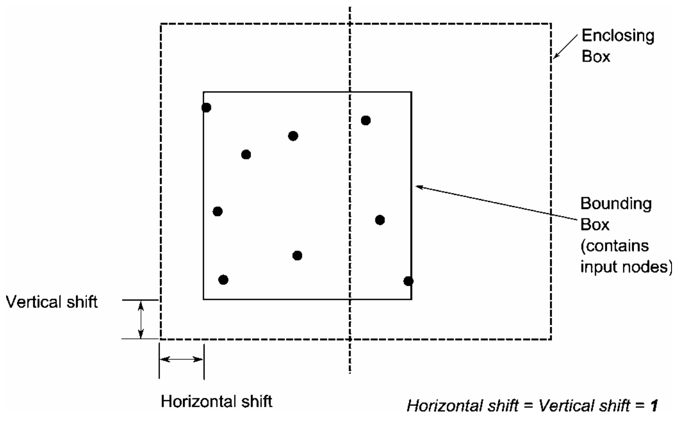

Enclosing Box
Bounding Box (contains input nodes)
Vertical shift
Horizontal shift
Horizontal shift = Vertical shift = 1

Figure 4. Arora-based shift

2.2 Portal-Respecting Tour

Next, a set of portals are placed in the grid lines. Squares share some portals which are the only way of communication between two adjacent neighbors. Portals are predefined and the execution time depends of the portal configuration. Salesman is allowed to enter and exit squares only by crossing these specified points. A possible path from *i* to *j* nodes can be expressed as $(i,P_1)(P_1,P_2)(P_2,P_3)(P_3,j)$ sequence where P_1, P_2 and P_3 are portals. And, a (m,r)*-light* path is a path whose edges can be crossed at most r times and always over portals.

According to Arora´s PTAS, each square has four portals, one at each corner, and additionally m portals equally spaced on each side of the square. A path is one that visits each node in the plane once, but can pass through portals more than once (Fig. 5).

To increase the number of portals implies to increase the number of combinations of them and then, the pairs of enter/exit portal. Because of this practical reason we decided to place the portals in such a way that there is at most one portal communicating neighbors' squares (see Fig. 6). That decision is based on the hypothesis that one only way of visiting nodes from a square to another one is sufficient for computing the path due to portals are not part of the final solution path. The structure of the generated quadtree is partially depicted in Fig. 7.

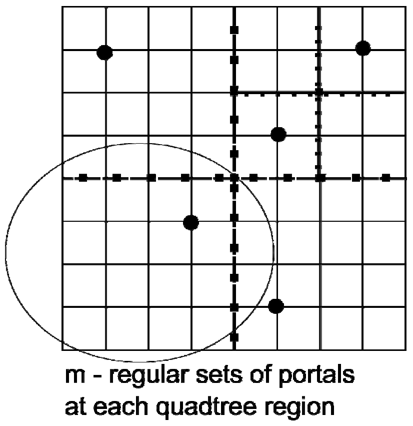


Figure 5. Arora´s PTAS portalization

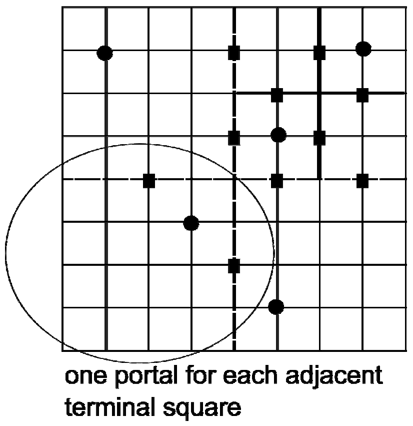


Figure 6. Arora-based portalization

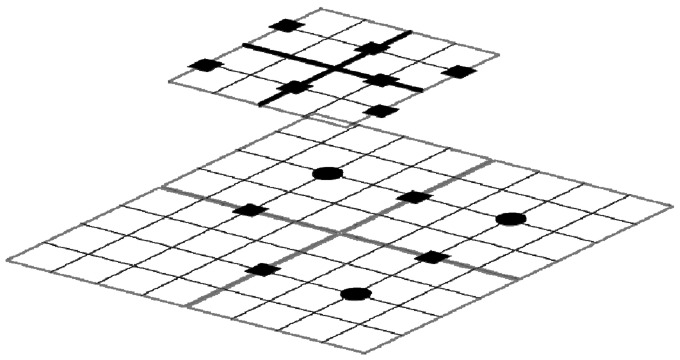


Figure 7. Two levels in the quadtree: 4-way recursive partition

2.3 Finding the Min-Cost k-light Portal-Respecting Tour

Finally, the optimal (m,r)*-light* salesman path is founded using *dynamic programming* over the shifted quadtree. Considering that we have interface information, i.e. the portals used by the tour to enter/exit a quadrant and the order in which the tour uses these portals, the subproblem inside each quadrant can be solved independently of the subproblems outside the quadrant. Starting from quadtree leaves, the solution is constructed in a bottom-up way, sub-problems allow solving the problem at the parent level. We solve the leaf subproblems first and use their solutions to calculate the solutions of internal node subproblems. The following pseudocode summarizes the essential steps:

```
1: DoDynamicProgramming(q: quadtree) : matrix
2:     leaves = q.getLeaves()
3:     squares= q.getSquares()
4:     for i=0 to leaves.getSize()
5:         calculateMatrix(leaves[i])
6:     for i=0 to squares.getSize()
7:         calculateMatrix(squares[i])
```

Following, we show the pseudocode of calculateMatrix:

```
1: calculateMatrix( c:square) : Matrix
2:     distance = MAXINT
3:     portals = c.getPortals()
4:     pairs = doPairings(portals)
5:     for i=0 to pairs.getSize()
6:         lastDistance = calculateDistance(c,pairs[i])
7:         setBestDistance(distance, lastDistance)
```

We store information in a lookup table that, for each square and for each choice of the interface contains the optimum solution for the subproblem inside the square. In Fig. 8 square identifies the quadrant processed to obtain the best solution. Pairings refer to the initial and final portal of the respective tour. Each table entry stores a partial solution together with the manner in which it can be reconstructed in the quadrant.

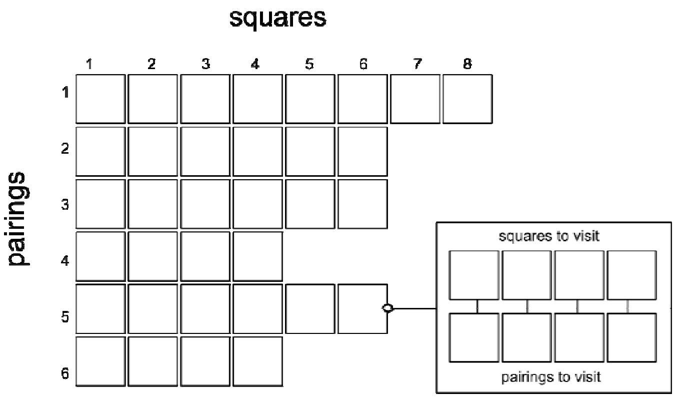


Figure 8. Dynamic Programming Table Structure

We consider only interfaces related to tours that do not cross each other. The number of possibilities is given by the Catalan number. Maybe more than one tour inside a square is needed to cover all the nodes that are inside the square. As a final remark, the solution can be reconstructed from the dynamic programming table looking up the decisions made at each step and choosing the shortest path stored in each child square entry. Dynamic programming is used to find the (m,r)-*light* salesman path using the quadtree generated previously. When the table is completed, a recursive algorithm that does not take into account portals and borders is used to reconstruct the path. The following pseudocode summarizes the trimming process:

```
1: doPathTrimming(q:quadtree, m:matrix,
                  index:integer, a:array):void
2:     if   isLeaf(q,i)
3:         a.add(q[index].getNode())
4:     else
5:         squares = m.getSubpath()
6:     for i=0 to squares.getSize()
7:         doPathTrimming(q,m,squares[i+.getIndex(), a)
```

The reconstruction is made in the *path's trimming* step, starting from the information stored by the dynamic programming table. We give the dynamic programming algorithm more hints about which portals are used by the tour to enter/exit each dissection square. Instead of defining the number of portals to be placed considering the constant proposed by Arora PTAS, we decide to place portals in lines belonging to quadrants that come into contact with some neighbor quadrant. Fig. 9 and Fig. 10 show the path before trimming step and after trimming step respectively.

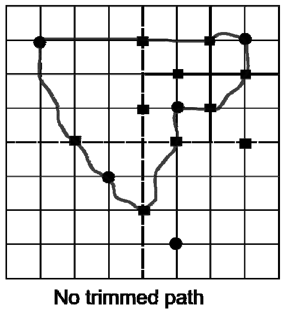


Figure 9. Path before trimming step

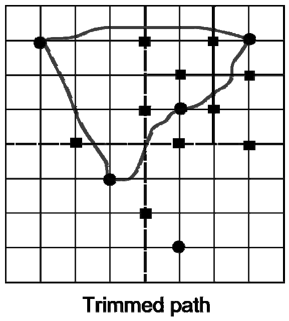
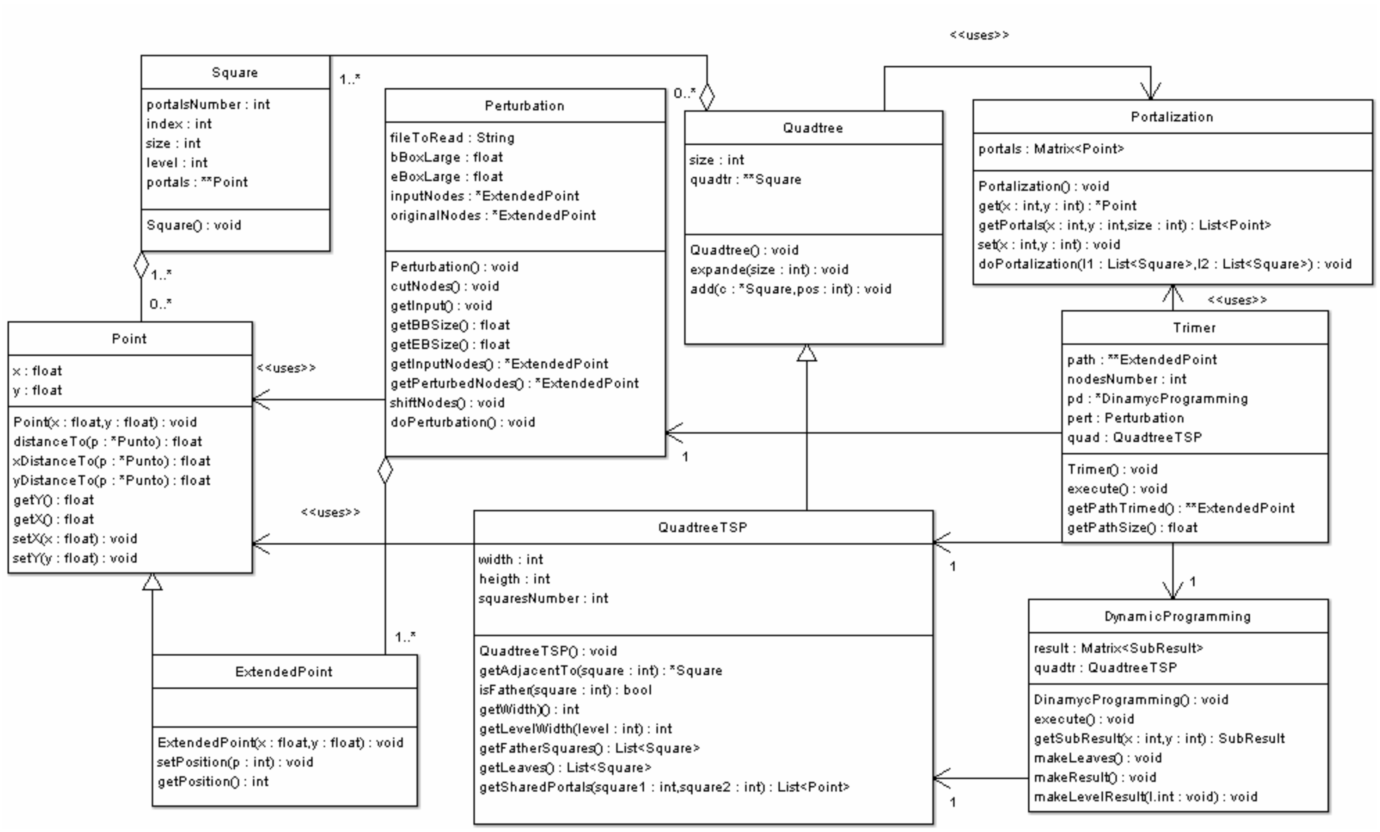


Figure 10. Path after trimming step



<<uses>>
Square
portalsNumber : int
index : int
size : int
level : int
portals : **Point
Square() : void
1..*
0..*
Perturbation
fileToRead : String
bBoxLarge : float
eBoxLarge : float
inputNodes : *ExtendedPoint
originalNodes : *ExtendedPoint
Perturbation() : void
cutNodes() : void
getInput() : void
getBBSize() : float
getEBSize() : float
getInputNodes() : *ExtendedPoint
getPerturbedNodes() : *ExtendedPoint
shiftNodes() : void
doPerturbation() : void
Quadtree
size : int
quadtr : **Square
Quadtree() : void
expande(size : int) : void
add(c : *Square,pos : int) : void
Portalization
portals : Matrix<Point>
Portalization() : void
get(x : int,y : int) : *Point
getPortals(x : int,y : int,size : int) : List<Point>
set(x : int,y : int) : void
doPortalization(l1 : List<Square>,l2 : List<Square>) : void
1..*
0..*
Point
x : float
y : float
Point(x : float,y : float) : void
distanceTo(p : *Punto) : float
xDistanceTo(p : *Punto) : float
yDistanceTo(p : *Punto) : float
getY() : float
getX() : float
setX(x : float) : void
setY(y : float) : void
<<uses>>
<<uses>>
Trimer
path : **ExtendedPoint
nodesNumber : int
pd : *DinamycProgramming
pert : Perturbation
quad : QuadtreeTSP
Trimer() : void
execute() : void
getPathTrimed() : **ExtendedPoint
getPathSize() : float
1
<<uses>>
QuadtreeTSP
width : int
heigth : int
squaresNumber : int
QuadtreeTSP() : void
getAdjacentTo(square : int) : *Square
isFather(square : int) : bool
getWidth() : int
getLevelWidth(level : int) : int
getFatherSquares() : List<Square>
getLeaves() : List<Square>
getSharedPortals(square1 : int,square2 : int) : List<Point>
1
1
1..*
ExtendedPoint
ExtendedPoint(x : float,y : float) : void
setPosition(p : int) : void
getPosition() : int
DynamicProgramming
result : Matrix<SubResult>
quadtr : QuadtreeTSP
DinamycProgramming() : void
execute() : void
getSubResult(x : int,y : int) : SubResult
makeLeaves() : void
makeResult() : void
makeLevelResult(l.int : void) : void
1

Figure 11. Arora-based software: Class Diagram

3 An Implementation of the Arora-based algorithm

The Arora-based software was modeled in UML in terms of class diagrams (Fig. 11) and sequence diagrams (Fig. 12).

Fig. 11 shows a class diagram of this software that includes the main classes and their interrelationships. Perturbation class implements functionality for transforming the main input into a valid input for the Arora-based implementation. It takes an important role processing node coordinates and computing mappings between original nodes and perturbed nodes, in order to reverse changes made in the node position in the plane. This last responsibility allows us to calculate the actual tour size of the final solution, contrasting it with the original distribution of the nodes.

ShiftNodes and cutNodes are two important methods in the perturbation step. They contribute to geometrical classification of input points, calculating the minimum internode distance in both axis and applying this distance to compute the classification by shifting the coordinates, and next obtaining odd coordinates.

Quadtree and QuadtreeTSP manage the hierarchy of squares in which the plane is partitioned. We choose an implementation of the quadtree based on linear array, to achieve a good performance in both, execution times and memory usage. This quadtree representation allows us a quick indexing and then an efficient path reconstruction.

We add an optimization in the quadtree memory usage by assigning an initial number of square positions and incrementing it, if necessary, in runtime execution.

QuadtreeTSP provides a set of useful methods for solving the problem such as getSharedPortals which returns a list of common portals from a pair of squares, or getLeaves and getFathers, which return a list of leaves and father squares, respectively, and isAdjacent which determines whether or not a pair contains squares that are adjacent.

After perturbing the input nodes and building a quadtree, portals can be placed in each one of the quadrants or dissection square. This functionality is also provided by Portalization class.

The dynamic programming has to enumerate all possible “interfaces”. This involves enumerating ways of choosing crossing points among the portals. This functionality is provided by the class DynamicProgramming. It is associated with the class QuadtreeTSP, a relevant collaborator and the Trimer class which provides functionality for reconstructing the resulting path in the method getPathTrimmed, and calculating the actual size of the path by using the Portalization class and the mapping of original points saved in this class.

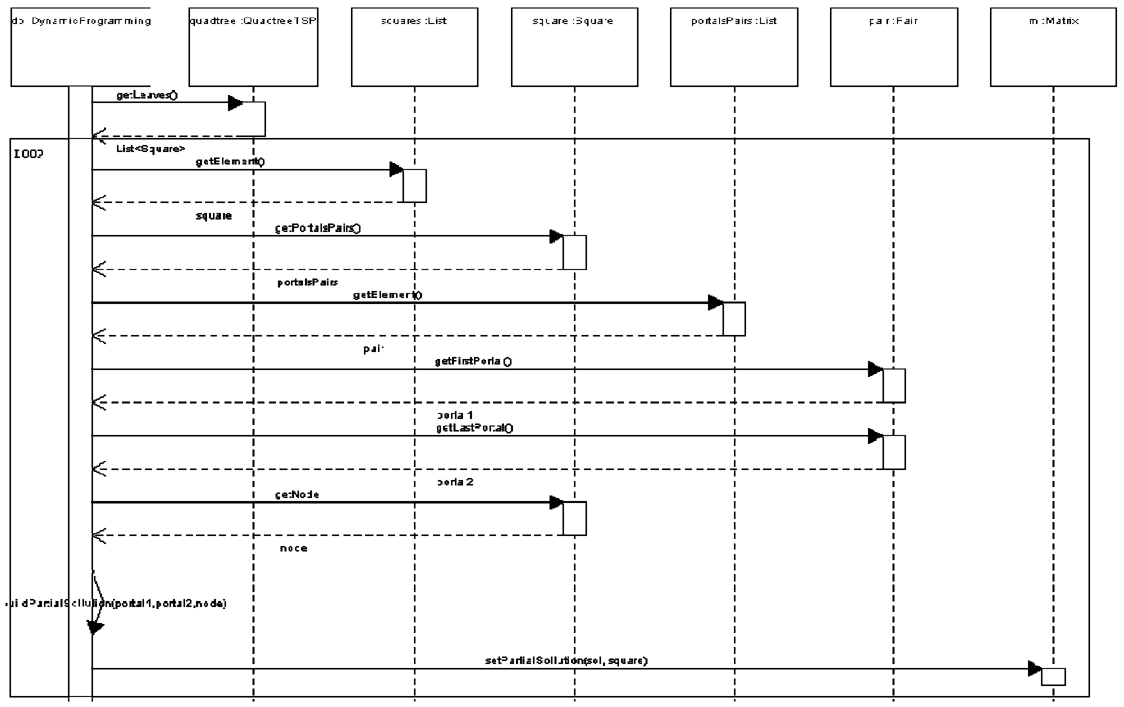


Figure 12. Arora-based software: sequence diagram

We give to the dynamic programming algorithm more hints about which portals are used by the tour to enter/exit each dissection square. We define an order to cover the lines, going from top to bottom and from right to left that allows us to know how portals enter in each one of the quadrants. Next it is possible to execute the compression method.

4 An Empirical Analysis

This section includes a comparative analysis of the Arora-based implementation with other implementations provided by the Concorde software [7]. Concorde is a C code that implements the symmetric TSP. It provides both exact algorithms such as a solver for a variant of the TSP called multiple TSP and, other algorithms. Table I shows performance ratios for TSPLIB instances (first column) whose optimal solution is known (second column) and our Arora-based implementation (third column).

We run our program on an AMD Athlon processor 3500+ with 1 GB of RAM. The instances were selected from TSPLIB, a library of sample instances for the TSP (and related problems) from various sources and of various types. Concorde's TSP solver has been used to obtain the optimal solutions to all TSPLIB instances having up to 15,112 nodes. Fig. 13 depicts the same information graphically.

Instance	*Concorde Solver*	*Arora PTAS*
att48	0,191	0,17643248
berlin52	0,08	0,09440166
burma14	0,04	0,00924726
dantzig42	0,26	0,08523932
rd18	0,06	0,02977082
rd22	0,03	0,03742933
rd25	0,06	0,03518771
rd27	0,19	0,05421247
rd31	0,11	0,06634306
rd34	0,14	0,070198307
rd39	0,16	0,091789599
rd75	2,47	0,515969208
ulysses16	0,05	0,01085529
kroC100	0,34	0,332153947
kroE100	1,47	1,203245664
ch130	1,23	1,208151874
pr144	1,14	2,048850063
kroA150	4,94	1,039160005
rat195	21,02	19,84023863

Table I

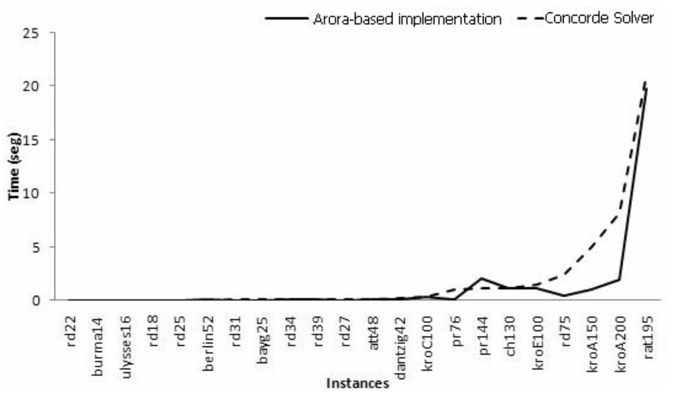


Figure 13. Arora-based implementation versus Concorde

Table II shows the combined length of the tour computed for different heuristic algorithms and our implementation for TSPLIB instances whose shortest tour is known and shown in the second column (Concorde solver). Arora implementation is compared with other implementation of algorithms such as Chained Lin-Kernighan, greedy and nearest neighbor [5].

Experimental results show that Arora´s PTAS is practically feasible. Table I and Table II show that in spite of its good performance, it seems that our approach must be improved to generate more approximate solutions. In most cases the significant theoretical results are lost due to implementation decisions. We think the quality of the solutions had to do with implementation aspects linked to data structures and the need to give more hints about which portals must be used by the tour.

5 Conclusions

This paper describes an implementation in C++ for the Euclidean TSP that is based on the novel Arora´s PTAS. We describe how to implement the essential steps of this algorithm in a way that improves the running time. The paper includes experimental results using TSPLIB instances and the Concorde Software to compare our implementation with other ones. Our implementation had shown the practical feasibility of Arora´s PTAS.

The existing TSP software [5] provides programs in the form of a “callable library”. Besides, the available TSP software does not allow us to experiment with Arora´s PTAS. Considering this, we foresee to define an architectural framework that assists, on the one hand, in the experimentation with the combination of different implementations of various steps of an algorithm and on the other hand, in the construction of solutions for higher dimension TSP and other geometric NP-hard problems. This framework should facilitate to experiment with different platforms and programming languages.

Instance	*Concorde Solver*	*Greedy*	*Nearest Neightbour*	*Kernighan*	*Arora*
A172	1673	1958	2186	1677	2783,06
a190	1847	2302	2256	1848	3086,66
a195	1922	2225	2400	1940	3134,183
att48	33522	40159	42597	33522	49466,1
bayg21	8114	9769	9969	8114	10035,92
bayg25	8798	10541	11396	8798	10287,61
bayg28	9055	9800	12092	9055	10637,62
berlin52	7542	9951	8848	7542	8254,06
Burma14	30	36	33	30	26,4974
Dantzig42	675	894	883	675	861,478
rd18	34	37	38	34	63,9063
Ulysses16	52	53	56	52	50,0808
Ulysses22	72	84	77	72	65,7936
st70	675	783	831	675	777,161
pr76	108159	140349	150800	108159	170924,7
kroE100	22068	24846	27823	22068	24814,79
gr120	1609	2085	2131	1609	2689,556
ch130	6110	7167	7656	6139	11677,6
pr144	58537	65844	63740	59274	63943,3
kroA150	26524	31892	32636	26525	45731,14
rat195	2323	2648	2658	2329	4056,60
kroA200	29368	34554	36106	29368	45108,27
eil101	629	794	826	629	760,835
a280	2579	3099	3229	2606	4182,06

Table II

6 References

[1] S. Arora. “Approximation schemes for NP-hard geometric optimization problems: A survey”. Math Programming, 97 (1,2) 2003.

[2] S. Arora. “Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems”; Journal of the ACM Vol. 45, Issue 5, 753-782, 1998.

[3] S. Arora, C. Lund, R. Motwani, M. Sudan and M. Szegedy. “Proof Verification and Hardness of Approximation Problems”; Journal of the ACM Vol. 45, Issue 3, 501-555, 1998.

[4] Dumitrescu, Adrian and Joseph S. B. Mitchell. “Approximation algorithms for TSP with neighborhoods in the plane”; Journal of Algorithms, Vol. 48, Issue 1, 135-159, 2003.

[5] Gutin, G., Punnen, A. (Eds.) “The Traveling Salesman Problem and Its Variations”. Kluwer Academic Publishers, 2004.

[6] Mitchell, J. S. B. “Guillotine subdivisions approximate polygonal subdivisions Part II - A simple polynomial-time approximation scheme for geometric k-MST, TSP, and related problems.” State University of New York, Stony Brook, 1996.

[7] Concorde Software page. http://www.tsp.gatech.edu/concorde/index.html , 2009

[8] TSPLIB, Traveling salesman problem library, www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/ 2009