Algorithmic Analysis of Traveling Salesman Problem

CS 255 Project Report

Under the guidance of

Prof. Aikaterina Potika

By

Prabaljit Walia

Venkata Satya Swathi Mattaparthi

Table of Contents

1. Introduction 3

2. Dataset Description 3

3. Algorithms 4

4. Methodology/Experiments 6

5. Results & Conclusion 10

6. Execution Instructions 13

7. References 14

1. Introduction

The Traveling Salesman Problem (TSP) is an optimization problem widely studied in computer science and operations research. The intuition for TSP comes from a board game developed by W.R. Hamilton and Thomas Kirkman (mathematicians) which involves finding a Hamiltonian cycle.

TSP aims to find the shortest path that visits each city exactly once and returns to the starting city [1] [6]. It is a renowned combinatorial problem. TSP is a combinatorial Non-deterministic Polynomial hard (NP-hard problem) as there is no optimal solution till date [7].

The Traveling Salesman Problem asks,
"*Given a list of cities with pairwise distances, what is the shortest possible route to visit each city exactly once and return to the starting city?*"

Our objective is to explore, implement and analyze different algorithms for solving this problem.

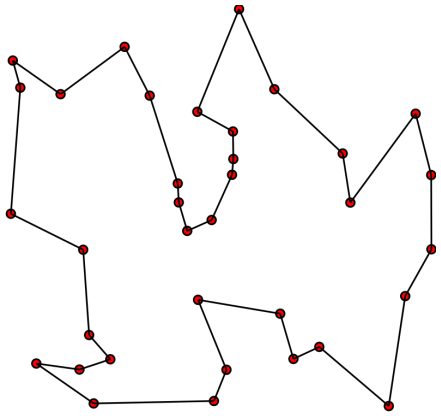

Fig 1: Solution to Traveling Salesman Problem

2. Dataset Description

We utilized the ATT48 dataset from TSPLIB, which is a public collection of Traveling Salesman Problem datasets [8]. ATT48 contains a set of 48 cities that represent the US State capitals. The solution tour has a length of 33523 units. This dataset didn't require much preprocessing since it was specifically designed for this problem.
It consists of three files:
att48_d.txt - the pairwise distances between each pair of cities
att48_xy.txt - the x and y coordinates of each city

att48_s.txt - an itinerary that minimizes the total distance i.e the solution tour

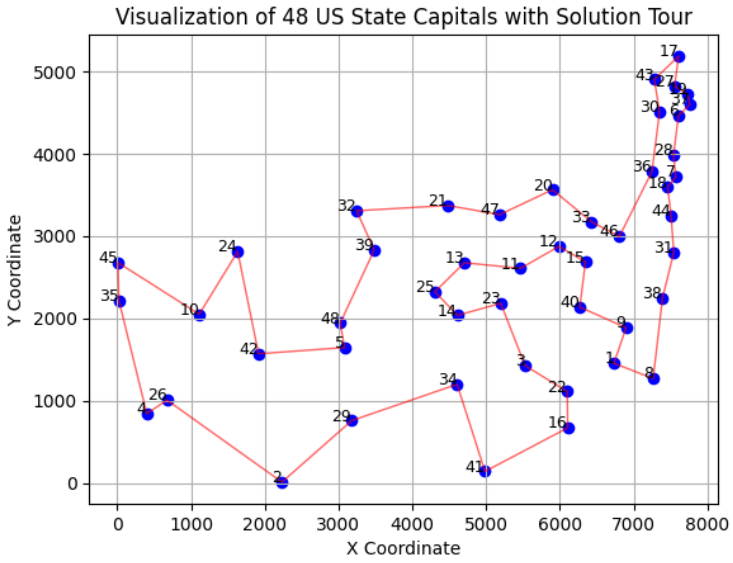

Fig 2: Visualization of solution tour

3. Algorithms

1) Brute Force

This is a naive approach. Here, all possible combinations of cities are visited. All the permutations of 48 cities are computed, each permutation determines the different possible route. For each of these routes, distance is calculated from the starting city visiting all cities and ending at the starting city. The minimum distance of these routes is calculated and the best tour is updated. This gives the tour in brute force method. The time complexity of this algorithm is O(n!) [7].

2) Nearest Neighbour Heuristic

This is a greedy algorithm that constructs a path by repeatedly visiting the nearest neighbor. It is an approximate solution to the Traveling Salesman Problem. We begin at a randomly selected starting city and in each step select the nearest unvisited city as the next city in the tour. This process continues until all the cities have been visited. While it is easy to understand, the tour is not guaranteed to be optimal. The result can significantly vary depending on the starting city chosen. The time complexity of this algorithm is O(n^2) [5].

3) Dynamic Programming

Dynamic Programming is an optimization technique to solve the Traveling Salesman Problem. It breaks down the problem into small subproblems and utilizes the solutions to these subproblems to form the solution to the actual problem. A memoization table is

used to store the solution to subproblems. We iterate through all the cities in the subset from the starting city and find the minimum cost for each city in a subset. Minimum cost is stored in the memoization table. Backtracking is used to find the shortest path from the final city to the starting city. The optimal route is determined by the memoization table. The time complexity of this algorithm is O(2^n. n^2)

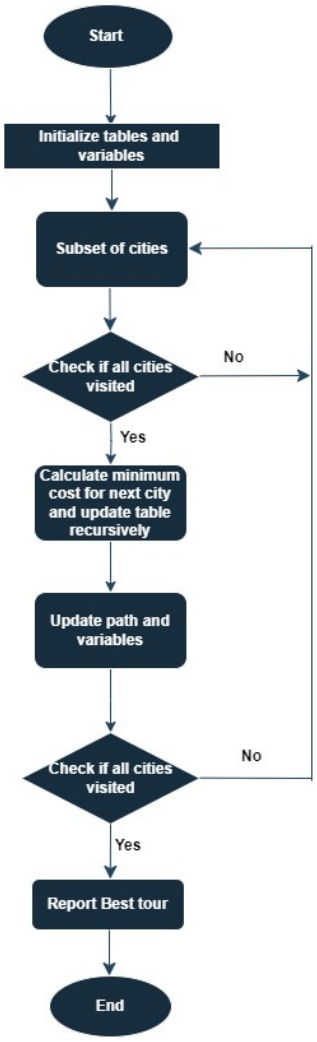

Fig 3: Dynamic Programming Flowchart

4) Ant Colony Optimization

Ant Colony Optimization is inspired by the ability of ants to find the shortest paths between their nest and sources of food by laying down pheromones that guide other ants. The idea is that virtual ants traverse the Traveling Salesman Problem graph and lay down pheromones. Paths with higher pheromone levels are more likely to be chosen and lead to a feedback loop that finds efficient routes. Ant Colony Optimization is typically not favored for its computational efficiency but for its ability to find good solutions to complex problems.

4. Methodology/Experiments

1) Brute Force

- The brute force solution to the Traveling Salesman Problem finds the shortest possible path that visits each city exactly once and returns to the starting city. The below code uses itertools to generate all permutations of the subset of cities, excluding the starting city. 'best_tour' will store the order of cities for the best tour found, and 'len_best_tour' will store the length of the best tour.
- In a for loop we iterate through each permutation and construct a tour. The length of the tour is calculated using Manhattan distance. Each time the current tour is compared to the previous tour and 'len_best_tour' is updated with the minimum length tour.
- For 48 cities, the program executed for a while and crashed as calculating 48! permutations require a lot of space. We performed trial and error and were able to execute the code for ¼ of the cities i.e. 12 cities. The length of the tour is 24854. The tour is not optimal as all cities were not considered. This modified code took 30 minutes to execute.

```
import itertools


n_cities = len(coordinates)

# Calculate all possible permutations of cities (excluding the starting city)
# NOTE: considering 1/4 no. of cities as program crashes for n or n/2 cities
city_permutations = list(itertools.permutations(range(1, n_cities//4)))


best_tour = None
len_best_tour = float('inf')

# Iterate through all permutations and calculate their tour lengths
for tour_permutation in city_permutations:
    # Start and end at 1st city (0th index)
    tour = [0] + list(tour_permutation) + [0]

    # Calculate the length of the tour
    tour_length = 0
    for i in range(n_cities//4):
        x1, y1 = coordinates[tour[i]]
        x2, y2 = coordinates[tour[i + 1]]
        #manhattan distance
        tour_length += abs(x2 - x1) + abs(y2 - y1)

    # Check if this tour is the best found so far
    if tour_length < len_best_tour:
        len_best_tour = tour_length
        best_tour = tour
```

Fig 4: Brute Force Code

2) Nearest Neighbour

Initialization:

- The algorithm starts at the first city.
- It keeps track of the number of cities in the problem (numCities) and maintains a set of unvisited cities (unvisited).

Tour Construction:

- The tour begins with the starting city and then iteratively selects the next city to visit.
- In each step, the nearest unvisited city to the current city is chosen as the next city (nextCity).
- The selected city is then marked as visited by removing it from the unvisited set, and the process is repeated until all cities are visited.

Completing the Tour:

- After visiting all other cities, the tour returns to the starting city to complete the circuit.
- The total distance of the tour is calculated by summing the distances between consecutive cities in the tour (totalDistance).

Distance:

- The computeTourDistanceAdjusted function calculates the total distance of a given tour using the distance matrix.

3) **Dynamic Programming**
   - The Dynamic Programming solution uses a memoization table to store solutions to subproblems. In the below code, ‘memo_table’ stores solutions to subproblems. It uses bitmask to represent the subset of visited cities and the second dimension represents the current city.
   - ‘next_city_table’ stores the next city to visit for each subset and current city. ‘tsp_dyn()’ is a recursive function that calculates the minimum cost of visiting all cities in the subset starting from city ‘i’.
   - In the for loop we iterate over all the cities in the subset. If a city is not already visited, we calculate the cost of visiting the city and recursively call ‘tsp_dyn()’ for the updated subset.

```
def TSP_DP(distanceMatrix):
  n = distanceMatrix.shape[0]


  #table to store subproblem solution
  memo_table = np.full((1 << n, n), -1, dtype=int)
  #table to store next city to visit
  next_city_table = np.zeros((1 << n, n), dtype=int)

  def tsp_dyn(subset, i):

    #all cities visited
    if subset == (1 << n) - 1:
      return distanceMatrix[i][0]

    #look up already stored values
    if memo_table[subset][i] != -1:
      return memo_table[subset][i]

    minimum_cost = float('inf')

    for next_city in range(n):
      #city not already visited
      if (subset >> next_city) & 1 == 0:
        new_subset = subset | (1 << next_city)
        cost = distanceMatrix[i][next_city] + tsp_dyn(new_subset, next_city)

        #get minimum cost and store the next city to visit
        if cost < minimum_cost:
          minimum_cost = cost
          next_city_table[subset][i] = next_city

    #store the calculated cost
    memo_table[subset][i] = minimum_cost
    return minimum_cost

  #start with first city and mark as visited
  minimum_tour_cost = tsp_dyn(1, 0)
```

```
  # Reconstructing the tour using backtracking
  tour = [0]
  subset = 1
  current_city = 0
  while subset != (1 << n) - 1:
    next_city = next_city_table[subset][current_city]
    tour.append(next_city)
    subset |= 1 << next_city
    current_city = next_city
  tour.append(0)

  return minimum_tour_cost, tour

Min_Cost, tour = TSP_DP(distanceMatrix)
print(f"Length of Shortest Path: \n", Min_Cost)
print(f"Tour Path: \n",tour)
```

Fig 5: Dynamic Programming Code

- We update the minimum cost later in the memoization table and store the next city to visit. Finally, we reconstruct the tour using backtracking. This loop runs until all the cities are visited in the subset.
- The program crashed for 48 cities because the size of the memoization table goes to 2^48 x 48. This requires 96 pebi bytes of memory which is very huge. After trial and error, we could execute the program for 20 cities.
- The length of the tour is 22970. As all cities were not considered, this is not the optimal tour.

4) Ant Colony Optimization

Ant Class:

- Each Ant object represents a single ant in the colony, maintaining a list of visited cities, its current city, and the total length of its tour.
- Methods allow the ant to visit a city, adding it to its visited list and updating the total tour length (visit_city function).

Pheromone Initialization:

- The pheromone matrix, representing pheromone levels on paths between cities, is initialized with a constant value (initialize_pheromone_matrix function).

Probability Calculation for Movement:

- For each ant, probabilities of moving to the next city are calculated based on pheromone levels and distance to each city (calculate_move_probabilities function).
- These probabilities are influenced by pheromone (alpha parameter) and distance (beta parameter).

Ant Movement and Tour Construction:

- Ants build their tours by choosing the next city based on calculated probabilities until all cities are visited. This is part of the main loop in the ant_colony_optimization function.

Pheromone Update:

- After all ants complete their tours, the pheromone matrix is updated. Pheromone levels decrease due to evaporation, and new pheromones are deposited (update_pheromones function).

Algorithm Parameters:

- Parameters include the number of ants, iterations, initial pheromone level, pheromone evaporation rate, and factors influencing pheromone and distance.

Iterations:

- The algorithm runs through a set number of cycles, constructing tours and updating pheromones in each cycle. The best tour found is recorded (ant_colony_optimization function).

5. Results & Conclusion

1.) Ant Colony Optimization

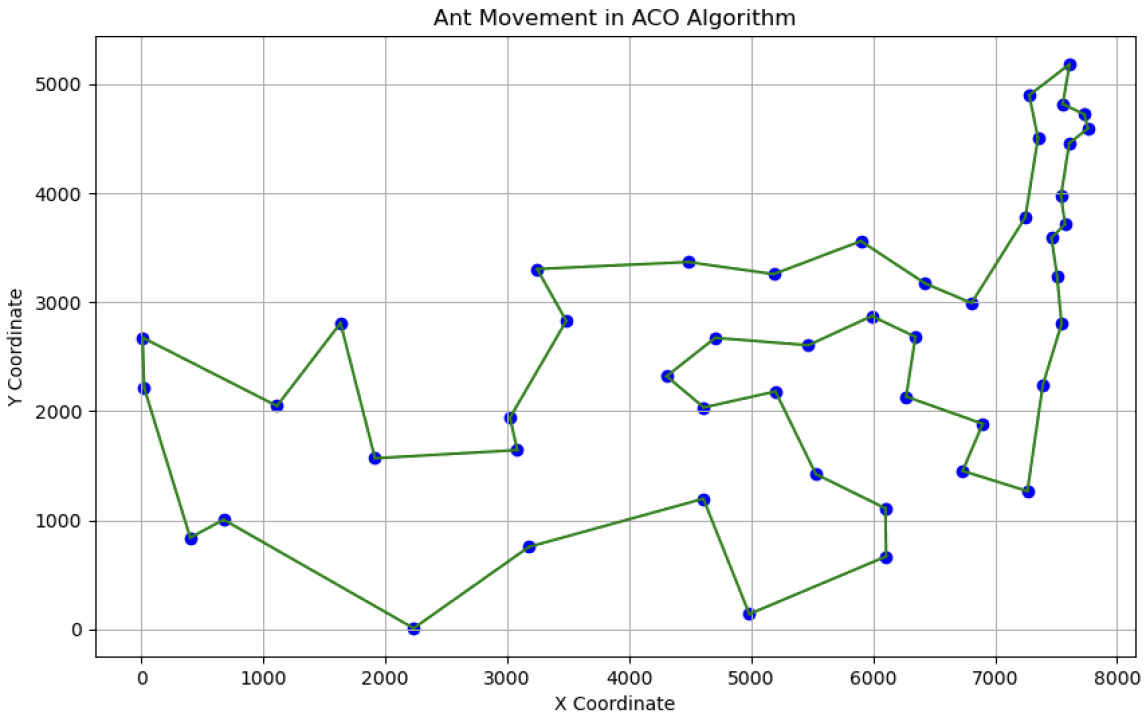

Fig 6: ACO Tour

Our best tour through ACO was 34,279 units. Some key insights:

- The tour, while not completely minimizing the travel distance between each pair of subsequent cities, effectively avoided the trap of local optima
- Despite the inherent randomness in the ACO, resulting from probabilistic transitions guided by pheromone levels, the algorithm consistently found tours that were competitive in terms of total distance. After changing the parameters like number of ants ranging between 10-30, the worst solution was still around 36,000 units.
- Comparing with the solution tour (Fig 2), the optimal solution has a more direct approach to certain city clusters, whereas the ACO path shows a more exploratory and distributed pattern
- Apart from a few occasional longer jumps, we got a tour that was cohesive and efficient.

2.) Nearest neighbor heuristic

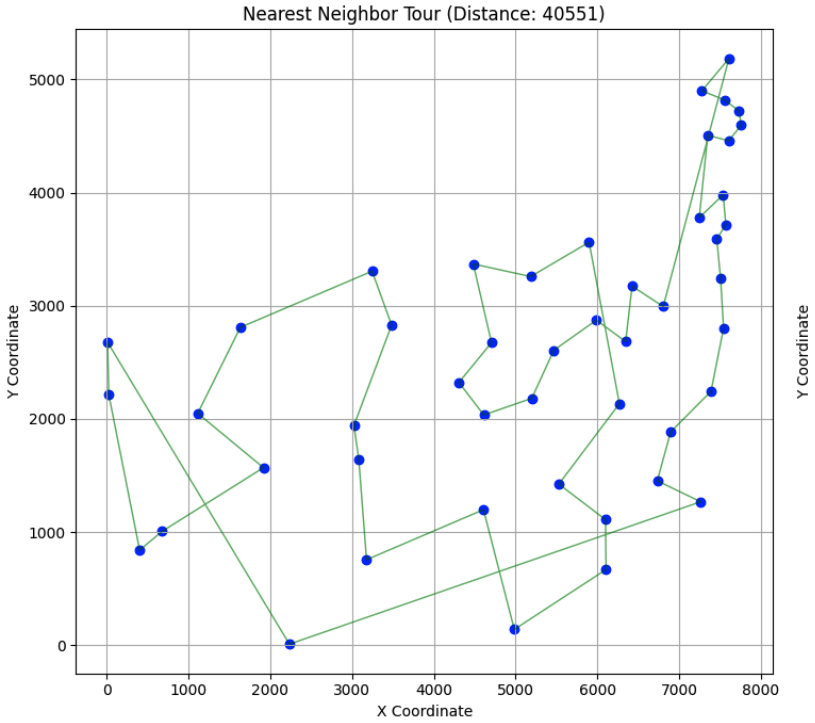

Fig 7: Nearest Neighbor Tour

Tour length through Nearest Neighbor was 40,551 units.
The path shows multiple long-distance links, as the heuristic does not account for the overall tour optimization. This algorithm is designed for speed rather than optimality, and this is reflected in the tour length, which is longer than those generated by both the ACO algorithm (34,279 units) and the given solution (33,551 units).

3.) Brute Force, Dynamic Programming and exploratory algorithms

Due to the high time complexity and huge memory requirements, we couldn't execute Brute force and DP on our dataset, but we were able to execute those implementations on a small subset of our dataset.

We also explored an existing implementation of Christofides algorithm for comparison only as this algorithm guarantees a solution that is 1.5 times the optimal solution.

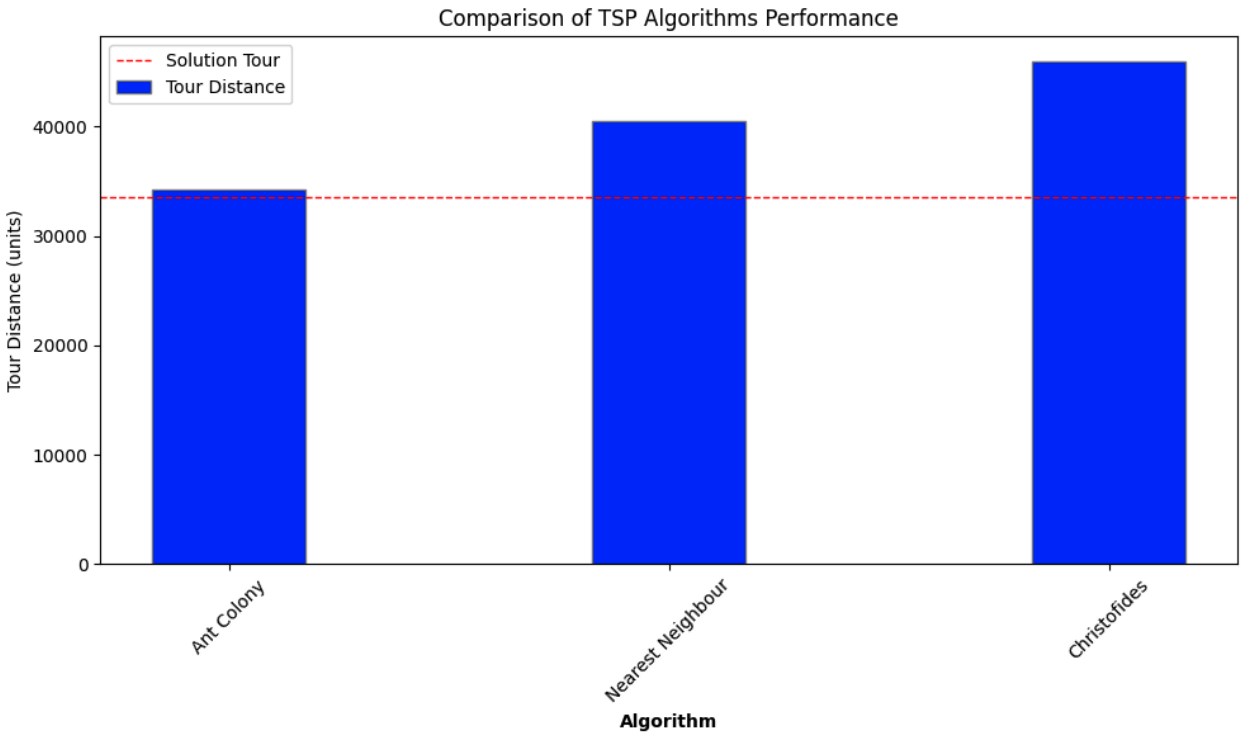

Fig 8: Comparison of Algorithm performance

Algorithm	**Tour Distance** (Solution Tour: 33523 units)	**Execution Time**
Ant Colony Optimization	34,279	1 min 1 sec
Nearest Neighbour	40,551	471 ms
Christofides Algorithm	45,929.2	4.04 ms
Dynamic Programming (only for subset of the dataset)	22,970*	1 min 40 sec
Brute Force (only for subset of the dataset)	24,854*	29 mins 55 sec

Table 1:Results

Conclusion

- Even with its simplicity, the brute force method becomes infeasible when dealing with larger datasets because of the exponential computing capacity.
- Because of its greedy nature, the nearest neighbor heuristic provides fast approximations but is not optimal.
- With its polynomial time complexity, dynamic programming appears promising for dealing with larger datasets but its high memory requirements prevent that.
- Christofides algorithm is an efficient approximation technique, guaranteeing a solution within 1.5 times the optimal length. **However, the Ant Colony Optimization method stands out because it can find effective paths via a feedback mechanism that is closest to the solution tour. Even with its inherent random nature, the ACO solutions with varying parameters were the closest to the solution tour.**
- Hence, solving TSP by the Ant Colony Optimization algorithm is the best solution proposed from our analysis.

6. Execution Instructions

Link to notebook: https://colab.research.google.com/drive/1ZGfP_dwlrCAfQuPKaNUKTPgAxc_-kmWc?usp=sharing

Github: https://github.com/prabal4546/Algorithmic-Analysis-of-TSP

The notebook can be executed on Google Colab after adding the dataset without any special requirements. For other environments, just delete the first cell which is specific to uploading the dataset on colab.

7. References

[1] C. Chase, A. N. Harrison, and M. Wilder-Smith, "An Evaluation of the Traveling Salesman Problem," Dept. of Computer, California Polytechnic Univ., Pomona Univ., 2020.

[2] M. Hahsler and K. Hornik, "TSP – Infrastructure for the Traveling Salesperson Problem," Journal of Statistical Software, vol. 23, no. 2, pp. 1-21, 2007.

[3] X. Bresson and T. Laurent, "The Transformer Network for the Traveling Salesman Problem," School of Computer Science and Engineering, Nanyang Technological University, Singapore, and Dept. of Mathematics, Loyola Marymount University, 2021.

[4] A. Salman, F. Ekstedt, and P. Damaschke, "A comprehensive survey on the generalized traveling salesman problem," ScienceDirect, 2020.

[5] J. Crouser, "Traveling Salesman Problem," 2020 : https://jcrouser.github.io/CSC250/projects/TSP.html

[6] "Traveling Salesman Problem," Wikipedia : https://en.wikipedia.org/wiki/Travelling_salesman_problem

[7] "Algorithms for Traveling Salesman Problem," Routific Blog, 2021: https://www.routific.com/blog/travelling-salesman-problem

[8] G. Reinelt, "TSPLIB: A traveling salesman problem library," ORSA Journal on Computing, vol. 3, no. 4, pp. 376-384, 1991.