A parallel iterated local search for the traveling salesman problem as a multi-core solution using OpenMP

Isaac Kosloski Oliveira[1]

[1]Faculdade de Computação
Universidade Federal de Mato Grosso do Sul (UFMS) Campo Grande – MS – Brazil

isaac.kosloski@ufms.br

Abstract. *This write-up reports the results of an implementation of the Iterated Local Search algorithm (ILS), both sequential and parallel, for the Traveling Salesman Problem (TSP). The results are disposable in two phases. The first phase compares the evaluation measures of ILS on 1 and 12 cores (sequential and parallel, respectively, and optimal). The second phase compares the proposed parallel algorithm with the reported results of metaheuristics algorithms that were used to solve the TSP in the literature.*

1. Introduction

One of the most prominent components in the set of combinatorial optimization problems, dating back to the 19th century, is certainly the *traveling salesman problem* (TSP). Finding the minimum weight cycle in a given graph is one of the few mathematical problems frequently occurring in the most popular scientific press [Reinelt 1994].

2. Literature Review

Research on optimization metaheuristics to design solutions for solving TSP problems has been developed since the early century, but it is still a subject of study.

A recent solution [Esra'a Alhenawi and Hussien 2024] uses the parallel RFD algorithm for solving the TSP, comparing speedup, running time, and efficiency on 1 (sequential), 4, 8, and 16 cores. Then, compare to three parallel water-based algorithms (*Water Flow, Intelligent Water Drops* and *Water Cycle*). Then, the proposed algorithm will be compared with the reported metaheuristics used to solve TSP in the literature. It brings an extensive evaluation measure set like distance, accuracy, running time, and speedup.

Another recent proposal [Scianna 2024] is to solve the TSP with the AddACO algorithm (a version of the Ant Colony Optimization method characterized by a modified probabilistic law at the basis of the exploratory movement of the artificial insects). In particular, the ant decisional rule is set to the amount in a linear convex combination of competing behavioral stimuli. It has an additive form (hence the name of our algorithm) rather than the canonical multiplicative one. The AddACO intends to address two conceptual shortcomings that characterize classical ACO methods:

(i) the population of artificial insects is, in principle, allowed to simultaneously minimize/maximize all migratory guidance cues (which is implausible from a biological/ecological point of view).

(i) a given edge of the graph has a null probability of being explored if at least one of the movement traits is equal to zero, i.e., regardless of the intensity of the others (this, in principle, reduces the exploratory potential of the ant colony).

A proposal of a discrete artificial bee colony algorithm with a fixed neighborhood search for the traveling salesman problem (TSP) called DABC-FNS [Xing Li and Shao 2024], where the solution obtained by the algorithm is expressed by a positive integer coding method. Meanwhile, the local enhancement strategy and the 2-opt strategy with fixed neighborhood search are introduced to improve the ABC algorithm's solution accuracy.

The Discrete Carnivorous Plant Algorithm with Similarity Elimination Applied to the Traveling Salesman Problem is another idea [Pan-Li Zhang and Zhang 2022], where it uses a combination of six steps: first, the algorithm redefines subtraction, multiplication, and addition operations, which aims to ensure that it can switch from continuous space to discrete space without losing information; second, a simple sorting grouping method is proposed to reduce the chance of being trapped in a local optimum; third, the similarity-eliminating operation is added, which helps to maintain population diversity; fourth, an adaptive attraction probability is proposed to balance exploration and the exploitation ability; fifth, an iterative local search (ILS) strategy is employed, which is beneficial to increase the searching precision; finally, to evaluate its performance, DCPA is compared with nine algorithms.

3. Problem description

The traveling salesman problem is one of the most well-known problems in combinatorial optimization and a member of the NP-hard problems [Esra'a Alhenawi and Hussien 2024]. Close related to the Hamiltonian cycle problem, its applications can also be modeled as a graph problem.

Modeling this problem as a complete graph with n vertices, where the set of the vertices represents a group of cities, and the set of the arcs typifies a group of roads interconnecting the cities. The main target is for the salesman to make a tour (hamiltonian cycle), visiting each city exactly once and finishing at the city he has started from [Thomas H. Cormen and Stein 2009].

In the standard classical problem, the salesman incurs a nonnegative cost $c(i,\, j)$ to travel from city i to city j. The desired total cost of this tour should be the minimum distance, where the total cost is the sum of the individual costs along the edges of the tour, i.e., the weight minimum cycle in the graph problem.

In a more formal specification for the TSP, given a complete graph $K_n = (V, E)$, where V is the finite set of vertices (typifying the cities), and E the set of edges (typifying the roads), that has nonnegative cost $c(v_i, v_j)$ (typifying the distance between two cities) associated with each edge $(v_i, v_j) \in E$. An acceptable solution is a permutation σ of V, represented as $S_\sigma = (v_{\sigma(1)}, v_{\sigma(2)}, ..., v_{\sigma(n)})$, where n = $|V|$, which minimizes the cycle cost $C(\sigma)$, then:

$$\mathbf{min} C(\sigma) = \sum_{i=0}^{n-1} c(v_i, v_{i+1}) + c(v_{n-1}, v_0) \tag{1}$$

Given S_n, the set of all symmetric permutations in V of n elements, the optimal solution S_{opt} is a minimum weight Hamiltonian cycle.

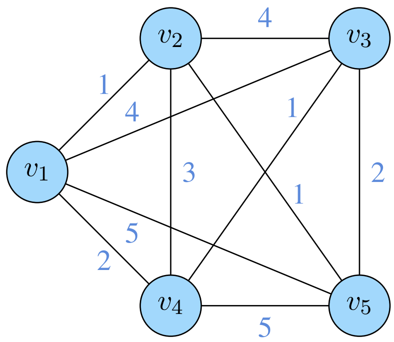

Figure 1. K_5 **graph for example 1.**

For example, given the graph K_5 in figure 1, an solution is the cycle $(v_1, v_2, v_5, v_3, v_4, v_1)$, illustrated in figure 2, which $C = c(v_1, v_2) + c(v_2, v_5) + c(v_5, v_3) + c(v_3, v_4) + c(v_4, v_1)$, as described in 1.

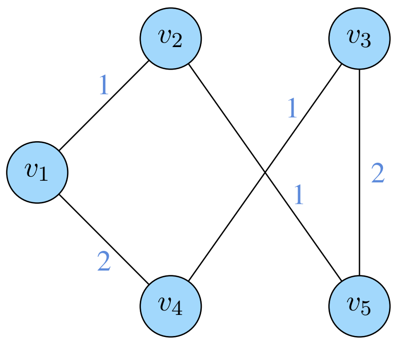

Figure 2. An instance of the traveling-salesman problem, on K_5 **graph, for example 1.**

Now, we can define the TSP as an optimization problem more carefully for implementation. Let a variable $x_{ij} \in \mathbb{B}$ for each edge $(v_i, v_j) \in E$, i.e. for $i = 1, 2, ..., n$ and $j = 1, 2, ..., n$, where $i \neq j$. Let a variable $c_{ij} \in \mathbb{R}$ be a cost (weight) value associated with each edge $(v_i, v_j) \in E$. With those statements, the objective function is formulated, and we can elucidate the constraints for completing the formulation. We desire a Hamiltonian cycle, i.e., include each vertex in our set exactly one time. We must let one edge as a way to "get in" and another to "get out", as the variable x_{ij} already is related to an edge, and for each vertex, there are exactly 2 edges associated. We just set the edges related to the vertex as 1. Other constraints are related to sub-cycles, so an alternative could be to let a set S of all sub-sets from V in which the number of elements is between 2 and $n-2$. For each $S_\sigma \in S$, we can set at least one edge of the selected cycle to "escape" of this set S. Given these statements, we can formulate:

$$z = \min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} X_{ij} :$$

$$\sum_{i=1}^{n} X_{ij} = 1,$$

$$\sum_{j=1}^{n} X_{ij} = 1,$$

$$\sum_{i \in S} x_{ij} \geq 1, \forall S_\sigma \in S,$$

$$x_{ij} \in \mathbb{B}, i \neq j.$$

4. Iterated Local Search

The *Iterated Local Search* (ILS) is a sophisticated metaheuristics algorithm designed to solve optimization problems. It operates by iteratively generating embedded solutions and comparing them to provide an acceptable one.

Imagine an optimization heuristic algorithm, a local search, tailored for a specific problem. This can be implemented as a *LocalSearch* procedure. The question that arises is, 'Can we iteratively optimize?' If the answer is 'yes', then the optimization obtained is significant and invaluable. To comprehend the workings of the local search, we assume it to be deterministic and memoryless.

Let C be the cost function of our optimization problem, a function that we strive to minimize. Let S be the finite set of all solutions s. In figure 3, a high-level block diagram of local search is illustrated, where given an input s, we always generate the same output $s*$ with a cost less than or equal to $C(s)$. The *LocalSearch* defines a n to 1 mapping from set S to a smaller set $S*$ with optimal local solutions $s*$. The main feature of a local search is a neighbor structure; from this, we can understand S as some topological structure, not just a set, where this allows agile to go from one solution to another [Helena R. Lorenço and STUTZLE].

Hence, a helpful concept is the *basin of attraction* of a local minimum $s*$, a set of s mapped to $s*$ under the local search routine. The *LocalSearch* is provided with a solution s that gives a local minimum $s*$ in the *basin of attraction*. To explore solutions in S out of the local *basin of attraction*, we can apply a change or *Perturbation* that leads to an intermediate state s', then, the *LocalSearch* can be applied and a new minimum local solution $s*'$ is given, as represented in the second part of bloc diagram in figure 4.

The next step is, given the two local minimum solutions, i.e., s^* and $s^{*'}$, a criteria that decides each one is our solution can be defined. The *AcceptanceCriterion* implements, which solution is the more feasible for the next step. It can be defined according to the problem and its desired solution, e.g., the less-cost solution or the more diversified solution. Once the base structure is defined, the bloc diagram in figure 4 provides a view of how our ILS system works.

Figure 3. Block diagram of LocalSearch procedure

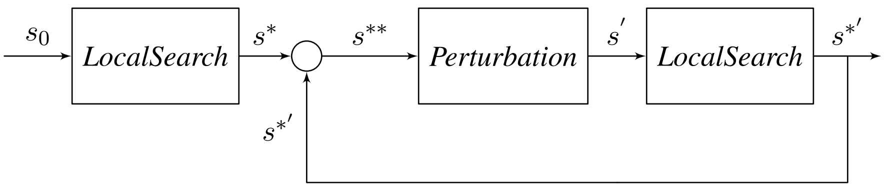

Figure 4. Block diagram of Iterated Local Search procedure

Hence, in the block diagram in figure 3, let us comprehend how ILS can be modularized and how its construction can be implemented procedure by procedure alone, as long we provide a type of ***s*** as input signal in each bloc of our linear system. This idea leads us to define the iterated local search algorithm. The algorithm 1 provides a high-level architecture proposed by [STUTZLE and DORIGO 1999].

Algorithm 1 Iterated Local Search

PROCEDURE *Iterated Local Search*
  $\mathrm{s_0}$ = GenerateInitialSolution()
  s^* = LocalSearch($\mathrm{s_0}$)
  REPEAT
    s' = Perturbation($\mathrm{s}^*, history$)
    $\mathrm{s}^{*'}$ =LocalSearch(s')
    s^* = AcceptanceCriterion($\mathrm{s}^*, \mathrm{s}^{*'}, history$)
  UNTIL termination condition met
END

The procure *GenerateInitialSolution()* runs only one time and provides the initial solution s_0, so that can be the input set for *LocalSearch()* procedure. Then, a *Perturbation()* is applied in s^* and gives a new set with other basin attraction in space S, providing a $s^{'}$ which is the new input set for another call of *LocalSearch()*. The *AcceptanceCriterion()* procedure defines which solution is better. To provide a feedback structure each time, the solution is set to s^*, and the algorithm repeats the last steps until a defined condition is met. These four components, i.e., *GenerateInitialSolution()*, LocalSearch(), textitPerturbation() and *AcceptanceCriterion()*, performing together, implements the ILS metaheuristic.

5. Implementing the TSP Using ILS

In this paper, a sequential and parallel version of the ILS algorithm has been implemented with $C++$. OpenMP has been used as a multi-core implementation for parallel TSP. The following sections list high-level algorithms and their steps.

5.1. Sequential ILS

Algorithm 2 Sequential ILS algorithm for solving the TSP

INPUT: nodes(n_{id}, X, Y), nodesDimension
OUTPUT: bestSolution[], Cost(bestSolution[]), elapsedTime()

for all nodes(n_{id}, X, Y) **do**
    searchGraph[] ← CalculateEclidianDistance(nodes(n_{id}, X, Y), nodesDimension)
end for
initialTime ← GetTime()
initialSolution[] ← greedyProcedure(Graph[])
bestSolution[] ← 2OptimizationProcedure(initialSolution[])
repeat
    perturbedSolution[] ← DoubleBridgeMoveProcedure(bestSolution[])
    optimizedSolution[] ← 2OptimizationProcedure(perturbedSolution[])
    bestSolution[] ← BetterCostProcedure(optimizedSolution[], bestSolution[])
until iteration > interationDimension - 1
finalTime ← GetTime()
return bestSolution[], Cost(bestSolution[]), elapsedTime(initialTime, finalTime)

5.1.1. The Input

To operate local search algorithms, the algorithm 2 requires a graph data structure for this implementation. Given the graph is complete, an adjacency matrix was chosen as a data structure. To provide a correct input with a list of nodes(n_{id}, X, Y) and a value with the nodes' dimension, a preprocessing step is necessary, where the input is extracted from a text file with this data.

A provided list nodes(n_{id}, X, Y) includes the *id* of a node, which is necessary to identify the vertex of our graph. A cartesian coordinate (X,Y) provides a position, and with the Euclidian method, our procedure calculates the distance between two nodes. A value of the n_{max} where it is the maximum value for the *id* n.

5.1.2. The Algorithm

After a correct provided input, the procedure continues:

- Recording the initial time to provide a final elapsed time;
- Generating a greedy tour as an initial solution to allow a start state for our search;
- Optimizing our initial tour with the 2-Opt local search algorithm and set as our best solution;
- Starting the main loop and repeating for interation value, predefined;
- Perturbating our best tour, for provides a way to get out of the basin of attraction of the previews best solution;
- Optimizing our perturbed tour, again with the 2-Opt local search algorithm and set as our intermediate tour, an optimized solution;
- Getting the less cost tour, between the previous and the best tour, and setting as our current best solution;

- Iterating the last steps until the predefined iteration value;
- Recording the final time to provide a final elapsed time;
- Returns the output records;

5.1.3. The Output

After running the metaheuristics algorithm implementation, we should expect as output the best Hamiltonian cycle (tour) for our problem, the cost value of this cycle, and the elapsed time.

5.2. Parallel ILS

Algorithm 3 Parallel ILS algorithm for solving the TSP

INPUT: nodes(n_{id}, X, Y), nodesDimension
OUTPUT: bestSolution[], Cost(bestSolution[]), elapsedTime()
for all nodes(n_{id}, X, Y) **do**
    searchGraph[] ← CalculateEclidianDistance(nodes(n_{id}, X, Y), nodesDimension)
end for
initialTime ← GetTime()
initialSolution[] ← greedyProcedure(Graph[])
bestSolution[] ← 2OptimizationProcedure(initialSolution[])
parallel // Each thread executes each command.
threadSolution[] ← bestSolution[]
repeat
    perturbedSolution[] ← DoubleBridgeMoveProcedure(bestSolution[])
    optimizedSolution[] ← 2OptimizationProcedure(perturbedSolution[])
    threadSolution[] ← BetterCostProcedure(optimizedSolution[], threadSolution[])
    threadSolutionWeight ← Cost(bestSolution[])
until iteration > interationDimension - 1 // Each thread executes $\frac{interationDimension\text{ - }1}{threadsDimension}$.
for each thread
if threadSolutionWeight < bestSolutionWeight **then**
    bestSolution[] ← threadSolution[]
end if
end parallel
finalTime ← GetTime()
return bestSolution[], Cost(bestSolution[]), elapsedTime(initialTime, finalTime)

Given the same problem, the input and output are equal to the sequential ILS, only differing in the algorithm itself.

5.2.1. The Algorithm

After a correct provided input, the procedure continues:

- Recording the initial time to provide a final elapsed time;
- Generating a greedy tour as an initial solution to allow a start state for our search;

- Optimizing our initial tour with the 2-Opt local search algorithm and set as our best solution;
- Starting the parallel region, where each thread executes each step simultaneously unless there is a specific region where each thread executes thread by thread alone;
- Setting for each thread tour the best initial tour. Each thread operates the procedures only in its particular tour list;
- Starting the main loop and repeating for each thread $\frac{iterationsDimension}{threadsDimension}$ times, where iterationsDimension is predefined;
- Perturbating our best tour, for provides a way to get out of the basin of attraction of the previews best solution;
- Optimizing our perturbed tour, again with the 2-Opt local search algorithm and set as our intermediate tour, an optimized solution;
- Getting the less cost tour, between the previous and the best tour, and setting as our current best solution;
- Iterating the last steps until the predefined value for each thread;
- Setting the best tour between all the threads as the best solution;
- Recording the final time to provide a final elapsed time;
- Returns the output records;

6. Experiment Setup

6.1. Implement Language and Frameworks

The implementation uses C++20 and OpenMP 5.2, compiled using GCC 12.2.0.

6.2. Testing Enviroment

The code was compiled and run on a machine x86_64, AMD Ryzen 5 5600GT with Radeon Graphics, 6 cores, and 12 threads. With Debian 12.2.0-14. The program was compiled using *g++ component.cpp node.cpp scanner.cpp functions.cpp mainExec.cpp -o ../bin/TspPar -fopenmp -Wall -pedantic*, and tested using a shell script.

6.3. Datasets - Inputs

For the datasets, nine TSP benchmarks were downloaded from TSPLIB [TspLib], which includes d198, a280, lin318, pcb442, rat783, u1060, pcb1173, d1291, and fl1577 have been used in this write-up for evaluating the proposed algorithm performance. The selected benchmarks varied in several cities where each city was represented by a 2D-Euclidian coordinate. The benchmarks' input was a *.tsp* file, with the format:

NAME : <benchmark name>

COMMENT : <benchmark description>

TYPE : TSP

DIMENSION : <dimension>

EDGE_WEIGHT_TYPE : EUC_2D

NODE_COORD_SECTION

<*i*> <coordinate x_i > <coordinate y_i >

EOF

The table 1 displays benchmarks properties.

Benchmark name	Dimension	Description
d198.tsp	198 city	Drilling problem (Reinelt)
a280.tsp	280 city	drilling problem (Ludwig)
lin318.tsp	318 city	318-city problem (Lin/Kernighan)
pcb442.tsp	442 city	Drilling problem (Groetschel/Juenger/Reinelt)
rat783.tsp	783 city	Rattled grid (Pulleyblank)
u1060.tsp	1060 city	Drilling problem problem (Reinelt)
pcb1173.tsp	1173 city	Drilling problem (Juenger/Reinelt)
d1291.tsp	1291 city	Drilling problem (Reinelt)
fl1577.tsp	1577 city	Drilling problem (Reinelt)

Table 1. TSP benchmarks' properties

6.4. Data Analysis - outputs

The output data, *.sol* file, has the format:

Iterations: 1000

Time: <time in seconds> sec - <time in minutes> min - <time in hours> horas

<fraction of time in hours> h <fraction of time in minutes> min

Problem dimension: <dimension>

Total distance: <total path distance>

$[v_1]$ $[v_2]$ $[v_3]$ ... $[v_i]$

Where v_i it's the city in the position i on the tour sequence. For each benchmark, there are 30 different output *.sol* files, to define the trial result statistically accurately.

6.5. Evaluation Measures

This section presents the evaluation metrics that are used for evaluating the proposed method.

1. Distance: the value of the best route found.
2. Accuracy: the percentage of retrieving the best route correctly. The optimal solution is provided in TSPLIB [TspLib].
3. Running time: is the duration between the end and the beginning of the main part (ILS algorithm itself) of the program running, using **chrono::high_resolution_clock::now()** for the sequential program and **omp_get_wtime()** for the parallel program.

$$RT = \text{finish\_time - start\_time} \tag{2}$$

4. Speedup: is the improvement in speed of execution of a task executed on two similar architectures with different resources:

$$SP = \frac{T_s}{T_p(n)} \tag{3}$$

- T_s: Sequential running time;
- $T_p(n)$: Parallel running time in function of n;
- n: Number of cores.

5. Efficiency: represents the speedup divided by the number of cores

$$EF = \frac{SP}{n} \tag{4}$$

7. Experimental results

The results are taken from an average of 30 trials for each instance.

7.1. Comparison of Sequential and Parallel ILS

Table 2. Avarege distance of each symmetric TSP benchmark, the Sequential and Parallel ILS, then the optimal solution.

Instance	Sequential Distance	Parallel Distance	Optimal
d198	1.628×10^4	1.721×10^4	1.578×10^4
a280	2.643×10^3	2.729×10^3	2.579×10^3
lin318	4.382×10^4	4.763×10^4	4.203×10^4
pcb442	5.172×10^4	5.348×10^4	5.078×10^4
rat783	9.271×10^3	9.659×10^3	8.806×10^3
u1060	2.467×10^5	2.416×10^5	2.241×10^5
pcb1173	6.066×10^4	6.498×10^4	5.689×10^4
d1291	——	5.690×10^4	5.080×10^4
fl1577	2.342×10^4	2.966×10^4	2.2249×10^4

Table 3. Accuracy of each symmetric TSP benchmark for Sequential and Parallel ILS.

Instance	Sequential Accuracy (%)	Parallel Accuracy (%)
d198	96.91	91.97
a280	97.58	94.50
lin318	95.91	88.25
pcb442	98.18	94.95
rat783	94.98	91.17
u1060	90.84	92.75
pcb1173	93.79	87.55
fl1577	95.00	75.02

Table 4. Running time, speedup, and efficiency of each symmetric TSP benchmark for Sequential and Parallel ILS.

Instance	Sequential Time (s)	Parallel Time (s)	Speedup	Efficiency (%)
d198	6.114×10^1	8.649	7.069	58.91
a280	1.755×10^2	2.411×10^1	7.279	60.66
lin318	2.775×10^2	3.941×10^1	7.041	58.68
pcb442	7.463×10^2	1.018×10^2	7.331	61.09
rat783	4.620×10^3	5.999×10^2	7.701	64.18
pcb1173	1.676×10^4	2.183×10^3	7.679	63.99
fl1577	4.293×10^4	5.797×10^3	7.406	61.71

References

Esra'a Alhenawi, Ruba Abu Khurma, R. D. and Hussien, A. G. (2024). Solving traveling salesman problem using parallel river formation dynamics optimization algorithm on multi-core architecture using apache spark. *International Journal of Computational Intelligence Systems*.

Helena R. Lorenço, O. M. and STUTZLE, T. Iterated local search.

Pan-Li Zhang, Xiao-Bo Sun, J.-Q. W. H.-H. S. J.-L. B. and Zhang, H.-Y. (2022). The discrete carnivorous plant algorithm with similarity elimination applied to the traveling salesman problem. *mathematics*.

Reinelt, G. (1994). *The Traveling Salesman: Computational Solutions for TSP Applications*. Springer-Verlag Berlin Heidelberg, 1º edition.

Scianna, M. (2024). The addaco: A bio-inspired modified version of the ant colony optimization algorithm to solve travel salesman problems. *Mathematics and Computers in Simulation*.

STUTZLE, T. and DORIGO, M. (1999). Aco algorithms for the traveling salesman problem.

Thomas H. Cormen, Charles E. Leiserson, R. L. R. and Stein, C. (2009). *Introduction to Algorithms*. The MIT Press, 3º edition.

TspLib. Disponível em: `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`. Acesso em: 11 de setemrbo 2023.

Xing Li, S. Z. and Shao, P. (2024). Discrete artificial bee colony algorithm with fixed neighborhood search for traveling salesman problem. *Engineering Applications of Artificial Intelligence*.