1 Introduction

This report compares variations of the 2-opt swap algorithm, considering various ways to implementations. This report focuses on two primary tasks undertaken for this purpose.

- The development of Julia functions for various 2-opt swap algorithms mentioned in the assignment guide.
- The execution of experiments on different instances to compare the variations.

No AI tools were utilized for the development and experimental analysis. However, AI assistance was sought for refining the report's language and expression, specifically employing GPT-4.0 for sentence structure and expression enhancement and Grammarly for grammar checks.

Two Julia scripts are submitted:

- **TSP.jl**: Key functions are defined in this script.
- **main.jl**: This script is the main file performing the experiment by importing functions defined in 'TSP.jl.' It generates instances, solves them, and saves the results. The generated instances are saved in the 'instances' folder as .csv files, and the results are saved in the 'results' folder as .csv and .gif files.

The remainder of this report is organized into four sections. Section 2 introduces the Julia scripts and the methodologies they implement. Section 3 delves into the experimental design and presents the results of applying the algorithms. Section 4 wraps up the report with a conclusion that synthesizes the key findings.

2 Algorithms & codes summary

Two functions are defined in 'TSP.jl' to implement the algorithms in the assignment guide. First, 'TwoOptSwapSlow' function is defined as follows:

```julia
function TwoOptSwapSlow(C, init_tour)

global eps
sitesN = size(C)[1]

opt_tour = copy(init_tour)
opt_cost = CalculateCost(C, opt_tour, sitesN)

flag = false
while !flag
    flag = true
```

```
494     for sInd in 1:1:(sitesN-1), ssInd in (sInd+1):1:sitesN
495         new_tour = Swap(opt_tour, sInd, ssInd)
496         new_cost = CalculateCost(C, new_tour, sitesN)
497         if (new_cost - opt_cost) < -eps
498             opt_tour = new_tour
499             opt_cost = new_cost
500             flag = false
501         end
502     end
503 end
504
505 return opt_tour, opt_cost
506
507 end
```

The inputs are 'C' and 'init_tour', where C is the cost matrix where c_{ij} represents the length of the arc (i, j) and init_tour is the initial tour from which the algorithm begins.

This function adopts the strategy of calculating the length of a new tour after creating it, which is implemented in lines 494 to 495. In addition, the function swaps whenever an improvement is identified (lines 497-501). Here, the variable 'eps' is set to 1e-10, which is the minimum gap separating two floating numbers.

The outputs of this function are 'opt_tour' and 'opt_cost.' opt_tour represents the optimal tour, which is a sequence of sites visited in the most efficient order obtained by the algorithm. opt_cost denotes the optimal cost, calculated as the sum of the arc lengths that comprise the tour.

Next, 'TwoOptSwap' function is defined as follows:

```
517 function TwoOptSwap(C, init_tour, mode=1)
518
519 global eps
520 sitesN = size(C)[1]
521
522 opt_tour = copy(init_tour)
523 opt_cost = CalculateCost(C, opt_tour, sitesN)
524
525 flag = true
526 if mode==1
527     while flag
528         flag = false
529         for sInd in 1:1:(sitesN-1), ssInd in (sInd+1):1:sitesN
530             dl = CalculateCostDiff(C, opt_tour, sitesN, sInd, ssInd)
531             if dl < -eps
532                 opt_tour = Swap(opt_tour, sInd, ssInd)
533                 opt_cost += dl
534                 flag = true
535             end
536         end
537     end
538 elseif mode==2
539     while flag
540         flag = false
541         for sInd in 1:1:(sitesN-1)
542             dl_list = [CalculateCostDiff(C, opt_tour, sitesN, sInd, ssInd) for ssInd in (sInd+1):1:sitesN]
543             ssInd_min = argmin(dl_list)
```

```
            dl_min = dl_list[ssInd_min]
            if dl_min < -eps
                opt_tour = Swap(opt_tour, sInd, sInd+ssInd_min)
                opt_cost += dl_min
                flag = true
            end
        end
    end
else
    println("provide a proper mode value")
    return nothing
end

return opt_tour, opt_cost

end
```

The inputs, C and init_tour, along with the outputs, opt_tour and opt_cost, remain the same as the previous function, but it also introduces an additional variable, 'mode.' This variable controls the switching between the two strategies outlined in the assignment guide. If the mode value is 1, the function executes a swap operation whenever an improvement is identified. Conversely, if the mode value is 2, it performs the swap operation for the best improvement identified.

The final function, 'MyTwoOptSwap,' maintains the input-output structure consistent with the preceding functions, defined as follows:

```
function MyTwoOptSwap(C, init_tour)

global eps
sitesN = size(C)[1]

opt_tour = copy(init_tour)
opt_cost = CalculateCost(C, opt_tour, sitesN)

T0 = 100.0
Tf = 1e-3
alpha = exp(-1/sitesN^2)
T = T0
while T > Tf
    sInd, ssInd = sort!(sample(1:1:sitesN, 2, replace=false))
    dl = CalculateCostDiff(C, opt_tour, sitesN, sInd, ssInd)
    if (dl < -eps) || (rand() < exp(-dl/T*sitesN))
        opt_tour = Swap(opt_tour, sInd, ssInd)
        opt_cost += dl
    end

    T *= alpha
end

return opt_tour, opt_cost

end
```

This algorithm provides probabilistic outcomes, borrowing an idea from the simulated annealing algorithm. This

approach involves gradually decreasing the value of ‘temperature’ until it reaches the preset threshold. At each iteration, a pair of sites is randomly sampled. If swapping the sampled pair improves the solution, then it is adopted. If the swap does not improve, the change is accepted with a probability which decreases as the algorithm proceeds.

As the algorithm progresses, the likelihood of accepting solutions that worsen the performance index diminishes. To put it another way, initially, this strategy encourages ’exploration,’ permitting a wide-ranging search across the solution space that might lead to superior outcomes in later iterations. As the process continues, the emphasis transitions to ’exploitation,’ progressively favoring solutions that enhance the performance index. Balancing between exploration and exploitation is the key of this algorithm, enabling it to move beyond local optima and achieve the global optimal solution.

The number of iterations until the algorithm termination is controlled by several parameters: the initial temperature (‘T0’ in line 573), the final temperature (‘Tf’ in line 574), and the cooling rate (‘alpha’ in line 575). The total number of iterations until the algorithm concludes is designed to be proportional to the square of the number of sites, reflecting the quadratic increase in the number of potential edges.

3 Experiments design and results

For the purpose of testing, 100 instances were generated for each specified problem size, with $N = 20, 50, 100, 1000$, to ensure a comprehensive evaluation. All the problems are symmetric, and the true optimal solutions were obtained through the Concorde solver.

Table 1: Experimental Environment

Resource	Specification
CPU	AMD Ryzen 5 5600X
RAM	80GB
OS	Ubuntu 22.04.4
Language	Julia 1.10.2
Used packages	Random, Base, LinearAlgebra Concorde (v0.1.3) DelimitedFiles (v1.9.1) JuMP (v1.20.0) Plots (v1.40.2) ProgressBars (v1.5.1) Statistics (v1.10.0) StatsBase (v0.33.21)

The experiment is performed on my personal computer located in my laboratory, N7-2 3332. The specifications of the computer, language, and used packages are in Table 1

3.1 Two ways for tour length calculation - Problem 1

Two ways of calculating tour lengths are compared. The first strategy is to calculate the tour length after creating a new tour, and the second strategy is to calculate the tour length by calculating the improvement. To compare the computational burden between these approaches, instances of problem sizes $N = 20, 50, 100$ are used. The results are given in Table 2

Table 2: Mean Time Consumption of Two Approaches (Problem 1 in the Assignment Guide)

# of Sites	Strategy #1 (sec)	Strategy #2 (sec)
20	4.11e-4	3.08e-5
50	1.65e-3	2.10e-4
100	9.90e-3	9.44e-4

The results show that the second approach is faster (about 10 times!) than the first.

3.2 Two ways for tour length improvement - Problem 2

Two distinct strategies for enhancing tour lengths are compared. The first strategy involves updating the tour as soon as any improvement is detected, while the second strategy updates the tour with the best improvement. The optimality gaps ($= (c_{heuristic} - c^*)/c^*$) and time consumption are compared for this purpose provided in Table 3.

Table 3: Statistics of Performance Indices of Two Strategies (Problem 2 in the Assignment Guide)

	Time Consumption (sec)				Optimality Gap (%)			
	Stg. #1		Stg. #2		Stg. #1		Stg. #2	
# of Sites	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.
20	3.08e-5	8.11e-6	2.43e-4	1.98e-3	3.55	4.29	2.90	4.08
50	2.10e-4	4.95e-5	2.56e-4	5.26e-5	7.20	3.82	6.26	3.45
100	9.44e-4	4.07e-4	1.02e-3	3.03e-4	10.10	3.02	9.12	2.71
1000	2.02e-1	2.47e-2	2.22e-1	2.26e-2	12.88	1.02	11.85	1.13

The findings presented in Table 3 indicate that despite the computational load being relatively similar, the second strategy surpasses the first regarding solution quality.

3.3 Probabilistic method - Problem 3

The experimental results of the probabilistic algorithm introduced in the previous section are summarized in Table 4. Two key observations emerge by comparing those with the results in Table 3. First, the probabilistic algorithm takes more time (about 10 times) than the other algorithms. However, this approach demonstrates a smaller variance

in time consumption relative to its mean. It is because the total number of iterations remains the same if the problem size is the same, but the other algorithms vary depending on the problem instance. To summarize, although the probabilistic algorithm demands greater computational time, its predictability is markedly enhanced, facilitating more reliable planning and resource allocation.

Regarding the solution quality, the probabilistic algorithm outperforms other strategies. This advantage becomes increasingly evident with larger problem sizes. The algorithm not only consistently secures lower mean optimality gaps compared to the others but also exhibits a trend of decreasing variances despite the escalation in problem scale. This pattern underscores the algorithm's robustness and ability to deliver better solutions.

Table 4: Statistics of Performance Indices of the Developed Algorithm (Problem 3 in the Assignment Guide)

	Time Consumption (sec)		Optimality Gap (%)	
# of Sites	Mean	Std.	Mean	Std.
20	7.19e-4	1.13e-3	2.06	3.05
50	3.39e-3	7.77e-4	4.75	2.77
100	1.30e-2	7.43e-4	7.09	2.64
1000	1.22	3.56e-2	9.78	0.922

4 Conclusions

To summarize the findings:

- Problem 1: Calculating improvement is better even though it needs more effort for implementation.
- Problem 2: Choosing the best one seems to be better.
- Problem 3: A probabilistic algorithm was developed, and the trade-off between time consumption & the solution quality was discussed.

References

Cook, W. J. (2015). In pursuit of the traveling salesman: mathematics at the limits of computation. Princeton University Press.

Animations

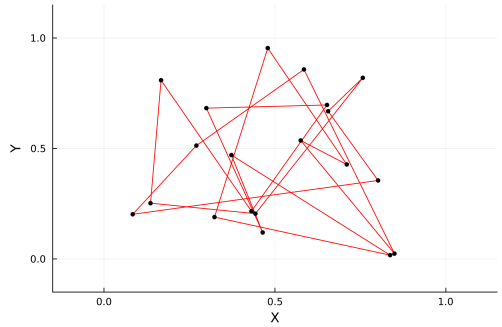

Figure 1: Animation 1: Adopting the Strategy 2-(ii) in the Assignment Guide

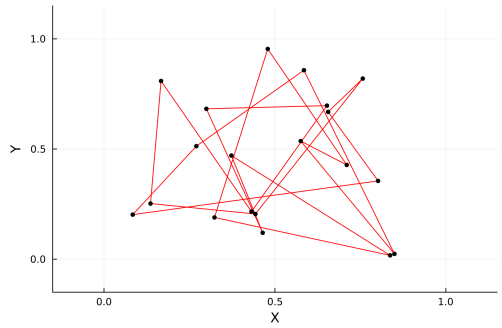

Figure 2: Animation 2: Adopting the Strategy 2-(i) in the Assignment Guide

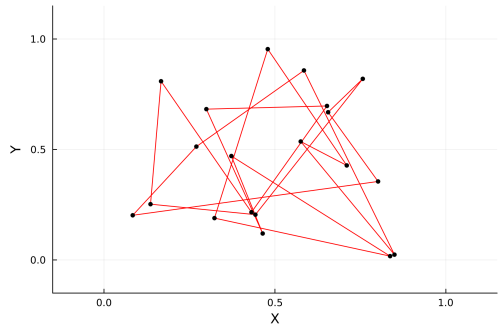

Figure 3: Animation 3: Adopting the Probabilistic Strategy