INFO-H-413 - Learning Dynamics: Implementation Exercise #2 The Traveling Salesman Problem with Time Windows - Stochastic Local Search Algorithms

Prof. T. Stüetzle

Jacopo De Stefani

April 24, 2013

Contents

Implementation

The heuristic solver had been written using the C++ programming language, in order to be able to take adventage of the functionalities offered by the Standard Template Library (STL).
By combining an object-oriented approach with those functionalities, I built a modular application core (*HeuristicCore*) that has been used for the two exercises required for this implementation.
The core is directly linked with the reader (*InstanceReader*) which reads the files containing all the informations to run the simulation (distance matrix, time windows vector and seeds list) and the writer (*Writer*) which outputs the results of the simulation in a CSV file.
Two different command line front-ends, sharing the same underlying structure, but having a different set of parameters, have been developed to distinguish the iterative-improvement interface from the variable neighborhood descent one.
The global program structure is depicted here:

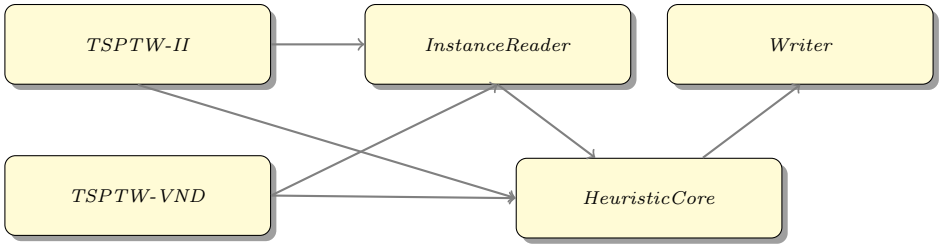

Figure 1: Simulation Code Structure

The names of the nodes corresponds to those of the implementation (.cpp) and header files for the corresponding classes. In addition to these, a class to represent the time windows (TimeWindow.h) and another one to represent the candidate solution as a user defined data type (CandidateSolution.{h,cpp}) have been implemented.

How to compile the program?

The software was developed in C++98 under Linux (Debian Wheezy), using the GNU g++ (Debian 4.7.2-5) 4.7.2 compiler and tested in this environment. The software is distributed as a compressed tar file, containing both the source code (in the `src` folder) and the scripts and instances (in the `bin` and `bin\instances` respectively). To install the program, first obtain the file TSPTW.V1.0.tar.gz.
Unzip the file by typing:

```
gunzip TSPTW.V1.0.tar.gz
```

and then unpack it by typing:

```
tar -xvf TSPTW.V1.0.tar
```

The software will be extracted in a new folder TSPTW.V1.0
Finally, by launching

```
make all
```

the Makefile will trigger the compilation of the files, producing the executables 'TSPTW-II' and 'TSPTW-VND" in the `bin` folder.
Note: The code is written in C++98. Hence, the code should be reasonable portable to other Operating Systems than Linux or Unix.

How to run the program?

Once the program has been compiled, two separate executable files, corresponding to the different kind of metaheuristic (i.e. Iterative Improvement and Variable Neighborhood Descent) can be either launched directly via the command line interface or using the bash script to launch.
The design choice to separate the two different kind of metaheuristic is made in order to limit the number of command line parameters to be given as input to the program.

Command-line execution

By launching[1]:

```
./TSPTW-II [PARAMETERS] -i [INPUT_FILE] -s [SEEDS_FILE]
```

```
./TSPTW-VND [PARAMETERS] -i [INPUT_FILE] -s [SEEDS_FILE]
```

one can display the information about the meaning of the different options that can be given as input to the program.
Additional information concerning the usage of the program can be found in the README file.
The only mandatory options are "-i, –input" and "-s,-seeds", since without these components will not be possible to execute the simulation.
The argument to the input option must contain the full path to the instance file.
The seeds file must contain a number of seeds at least equal to the number of runs required, one seed per line without any additional information.
The options controlling the same parameters are mutually exclusive, with that meaning that only one option must be selected in order to run the program. If multiple options for the same parameter are chosen, the program will not run.

TSPTW-II

Parameter	Mutually exclusive options
Initial solution	-d,–random , -h,–heuristic
Neighborhood type	-t,–transpose , -e,–exchange , -n,–insert
Pivoting rule	-f,–first-imp , -b,–best-imp

TSPTW-VND

Parameter	Mutually exclusive options
VND algorithm	-t,–standard , -p,–piped
Neighborhood chain	-a,–TEI , -b,–TIE

Output Every experiment produces a single file:

- **(II)** $[pivoting\_rule].[neighborhood\_type].[INSTANCE\_NAME]$ with $[pivoting\_rule] \in \{first, best\}$ and $[neighborhood\_type] \in \{transpose, exchange, insert\}$
- **(VND)** $[vnd\_type].[neighborhood\_chain].[INSTANCE\_NAME]$ with $[vnd\_type] \in \{standard, piped\}$ and $[neighborhood\_type] \in \{tei, tie\}$

Examples:

```
exchange.best.n80w200.004.txt
standard.tei.n80w20.003.txt
```

The internal structure of the file is the following:

Seed	CV	CpuTime	PRPD

For each run of the algorithm, the program writes in the file, using a tabulation as separator, the used seed, the number of constraints violations, the cpu runtime and the penalised relative percentage deviation.

[1] The squared-bracket notation implies that the formal parameters to the script have to be substituted by their actual value.

Script execution

By launching:

```
./launchTSPTW-II.sh [INSTANCE_NAME] [RUNS] [SEEDS_FILE]
```

```
./launchTSPTW-VND.sh [INSTANCE_NAME] [RUNS] [SEEDS_FILE]
```

The script will:

1. Create the files required by the data processing script to write statistics.
2. Generate a seed file, named [SEEDS_FILE] containing [RUNS] randomly generated seeds.
3. Launch the corresponding TSPTW program for [RUNS] using all the possible combinations of input options.
4. Wait for the termination of all the previously launched experiment and call the R data processing script

The script assumes that all the instances are located into the instances folder, hence it is only necessary to indicate the instance name, instead of the complete path.

Output Each execution of the script will then generate the files:

- $[INSTANCE\_NAME] - CpuTime.pdf$ containing the boxplots of the runtime distribution for each algorithm.
- $[INSTANCE\_NAME] - PRPD.pdf$ containing the boxplots of the PRPD distribution for each algorithm.
-
  - **(II)** transpose.first, exchange.first, insert.first, transpose.best, exchange.best, insert.best
  - **(VND)** standard.tei, standard.tie, piped.tei, piped.tie

The internal structure of the file is the following:

Instance	**Infeasible**	**mean(PRDP)**	**mean(CpuTime)**

Each line contains the instance name, the percentage of infeasible runs, the mean PRDP and the mean runtime across [RUNS] runs.

Problem 1

Ant Colony Optimization

Problem statement

Implement two stochastic local search (SLS) algorithms for the traveling salesman problem with time windows (TPSTW), building on top of the perturbative local search methods from the first implementation exercise.

1. Run each algorithm 25 times with different random seed on each instance. Instances will be available from http://iridia.ulb.ac.be/~stuetzle/Teaching/HO/. As termination criterion, for each instance, use the maximum computation time it takes to run a full VND (implemented in the previous exercise) on the same instance and then multiply this time by 1000 (to allow for long enough runs of the SLS algorithms).
2. Compute the following statistics for each of the two SLS algorithms and each instance:
   - Percentage of runs with constraint violations
   - Mean penalised relative percentage deviation
3. Produce boxplots of penalised relative percentage deviation.
4. Determine, using statistical tests (in this case, the Wilcoxon test), whether there is a statistically significant difference between the quality of the solutions generated by the two algorithms.
5. Measure, for each of the implemented algorithms on 5 instances, the run-time distributions to reach sufficiently high quality solutions (e.g. best-known solutions available at http://iridia.ulb.ac.be/~manuel/tsptw-instances#instances). Measure the run-time distributions across 25 repetitions using a cut-off time of 10 times the termination criterion above.
6. Produce a written report on the implementation exercise:
   - Please make sure that each implemented SLS algorithm is appropriately described and that the computational results are carefully interpreted. Justify also the choice of the parameter settings and the choice of the iterative improvement algorithm for the hybrid SLS algorithm.
   - Present the results as in the previous implementation exercise (tables, boxplots, statistical tests).
   - Present graphically the results of the analysis of the run-time distributions.
   - Interpret appropriately the results and make conclusions on the relative performance of the algorithms across all the benchmark instances studied.

Introduction

Ant Colony Optimization is an example of population-based metaheuristic (i.e a set of algorithmic concepts that can be used to define heuristic methods) inspired by the behavior of the ant species *Iridomyrmex humilis*. To be more precise, these insects are able, by means of stigmergic communication, to choose the shortest path between their nest and a food source, when given the choice ([1]).
The communication process occurs by deposing a certain quantity of pheromone in the environment that can be sensed by the other ants and that will be used by them as an heuristic (i.e an information to guide their choice) for selecting the shortest path.
Furthermore, the pheromone quantity on a certain location decreases over time because of evaporation, thus requiring a continuous deposit process to be effective.

The convergence to one of the paths will occur as a consequence of the self-reinforcing pheromone deposit mechanism. In fact the more pheromone is deposited on a path, the more ants will follow the pheromone trail on that path deposing even more pheromone.
The first application of Ant Colony Optimization method, the Ant System, has been made on the optimization version of the Travelling Salesman Problem (TSP) ([3]).
In this implemetation a population of virtual agents (an ant colony) is used to explore the search space (the virtual environment).
In the same fashion as the real insects, the ants are able to deposit virtual pheromone in the environment, to signal to the other ants the presence of promising solutions.
The general outline of the implemented algorithm is the following (as in [2]):

Algorithm 1 Ant Colony Optimization - Outline

1: *InitalizePheromoneTrail*
2: **while** !(TerminationCondition) **do**
3: *ConstructAntsSolutions*
4: *LocalSearch* (Optional)
5: *UpdatePheromoneTrails*
6: **end while**

Problem formulation

An instance of the Travelling Salesman Problem with Time Windows (TSPTW) can be expressed as a tuple $< N, E, c, t >$ where:

- $N : \{0, \cdots, n\}$ - Node set
- $E : N \times N$ - Edge set
- $c : E \mapsto \mathbb{R}$ - Edge cost function mapping a cost c to every edge $e \in E$.
- $t : N \mapsto \mathbb{R}^2$ - Time window function mapping a couple of values a_i, b_i representing respectively, the opening and closing time of the time window, such that $a_i < b_i$, to every node $i \in N$.
- A candidate solution for the problem, in this case, is represented as a permutation p of the nodes in N, where p_i represents the i^{th} solution component (node) in the sequence, such that $p_0 = 0 \forall p$.

As in [4] the formal definition of the problem is:

$$\begin{aligned} \textbf{min} \quad & f(p) = \sum_{i=0}^{n} c(e_{p_i p_{k+i}}) \\ \textbf{subject to} \quad & \Omega(p) = \sum_{i=0}^{n+1} \omega(p_i) = 0 \end{aligned} \tag{1}$$

where

$$\begin{aligned} \omega(p_i) &\begin{cases} 1 & A_{p_i} < b_{p_i} \\ 0 & \text{otherwise} \end{cases} \\ A_{p_{i+1}} &= \max(A_{p_i}, a_i) + c(e_{p_i p_{k+i}}) \end{aligned} \tag{2}$$

As one may notice in 1 and 2, a feasible solution p for the problem is a permutation where the constraints related to the time windows are met for every node i in the permutation.

Algorithm structure

The proposed algorithm is an implementation of one of the extensions to the Ant Colony Optimization metaheuristic framework, the $\mathcal{MAX} - \mathcal{MIN}$ Ant System (cf. [5]). The main features of the algorithm are the following:

- Only iteration best or best-so-far ants update pheromone
- $\forall t\ \tau_{\min} < \tau_{i,j}(t) < \tau_{\max}$ - Pheromone trails have explicit upper and lower limits
- $\tau_{i,j}(0) = \tau_0 = \tau_{\max}$ - Pheromone trails initialized to upper limit
- Pheromone trails are re-initialized when stagnated

Algorithm 2 $\mathcal{MAX} - \mathcal{MIN}$ Ant System for TSPTW - Outline

procedure ACO($\alpha, \beta, \rho, \tau_0, q_0, n_{ants}, n_{cities}, t_{max}$) ▷ The main procedure
Require: N - Node set
Require: E - Edge set
Require: c - Edge cost function
Require: t - Time window function
    InitalizePheromoneTrail(τ_0, n_{cities})
    while *!TerminationCondition*(t_{max}) **do**
        for all Ants **do**
            ConstructSolution(α, β)
            IterativeImprovement(INSERT,BEST-IMPROVEMENT)
            UpdatePheromoneTrails(ρ, q_0)
        end for
    end while
    return solution

end procedure

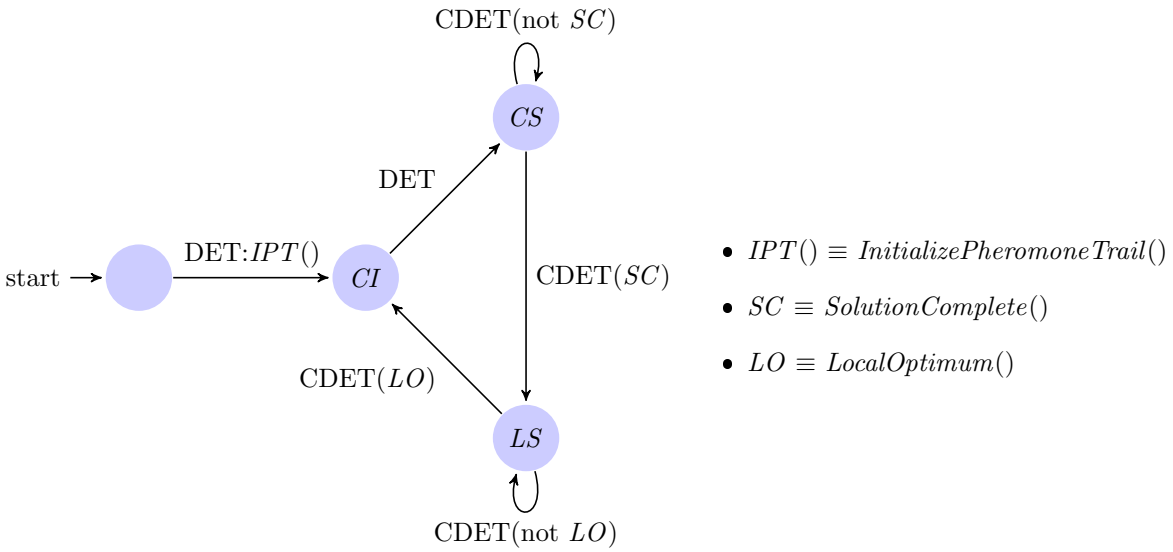

Figure 2: $\mathcal{MAX} - \mathcal{MIN}$ Ant System GLSM

In the implementation, an instance is completely defined by:

- **Cost matrix** - Encapsulating information on the node set N, edge set E, and weighting of each edge c.
- **Time window vector** - Describing the time window mapping function t for each node.

Pheromone Initialization

As discussed in , the ACO methods are based on stigmergic communication among the agents by means of virtual pheromone.
While the real ants can deposit pheromone anywhere in the environment, the virtual ants may only exchange informations concerning solutions components.
For this reason, every admissible edge e_{ij} of E has an associated pheromone value τ_{ij}, that have to be initialized at the beginning of the execution of the algorithm.
The inizialization value τ_0 is a parameter of the algorithm.

Algorithm 3 Pheromone Initialization

```
1: procedure InitalizePheromoneTrail(τ_0, n_cities)
2:     τ_max ←
3:     τ_min ←
4:     i ← 0
5:     j ← 0
6:     for i < n_cities do
7:         for j < i do
8:             τ_ij ← τ_0
9:             τ_ji ← τ_ij
10:            j ← j + 1
11:        end for
12:        i ← i + 1
13:    end for
14: end procedure
```

Solution construction

Algorithm 4 Solution Construction

```
1: procedure ConstructSolution(α, β)                    ▷ The main procedure
2:                                                      ▷ s_i reprents the i^th component of the solution
3:     s_0 ← 0                                          ▷ Every solution starts at the depot
4:     s_1 ← RandomCitySelection()                      ▷ Random choice of the starting city
5:     i ← 1
6:     while !SolutionComplete() do
7:         k ← RouletteWheelSelection()                 ▷ k stochastically chosen according to the probability distribution defined by p_k
8:         s_i ← k
9:         i ← i + 1
10:    end while
11: end procedure
```

$$p_k = \begin{cases} \frac{[\tau_{i-1,k}]^\alpha \cdot [\eta_{i-1,k}]^\beta}{\sum_{j \in N(s_{i-1})} [\tau_{i-1,j}]^\alpha \cdot [\eta_{i-1,j}]^\beta} & k \in N(s_{i-1}) \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Possible heuristics

Cheng and Mao, 2007

The local heuristics used in the current algorithm are similar to that proposed by Gambardella et al. [24] in their multiple ant colony system (MACS) designed to solve the vehicle routing problem with time windows (VRPTW).
[24] L.M. Gambardella, E.D. Taillard, G. Agazzi, MACS-VRPTW: A multiple ant colony system for vehicle routing problems with time windows, in: D. Corne, M. Dorigo, F. Glover (Eds.), New Ideas in Optimization, McGraw Hill, London, UK, 1999, pp. 63–76.

$$[\eta_{i-1,k}]^\beta = [g_{i-1,k}]^\beta \cdot [h_{i-1,k}]^\gamma \tag{4}$$

$$g_{i-1,k} = \begin{cases} \frac{1}{1+e^{\delta \cdot (G_{i-1,k} - \mu)}} & G_{i-1,k} = b_k - t_k \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

where

- $G_{i,j} = b_j - t_j$ - Slack corresponding to the time window j while being in node i
- t_i - Arrival time at node i
- b_i - Closing time of time window i
- $G(i) = \{k \mid G_{i,k} \geq 0\}$ - Set of feasible neighbors of node i (i.e. such that node k is reached earlier than its closing time)
- $\mu = \frac{1}{|G(i)|} \sum_{j \in G(i)} G_{i,j}$ - Average slack
- δ - Parameter to control the slope of the sigmoidal function

$$h_{i-1,k} = \begin{cases} \frac{1}{1+e^{\lambda \cdot (H_{i-1,k} - \upsilon)}} & H_{i-1,k} = t_k - a_k \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

where

- $H_{i,j} = t_j - a_j$ - Waiting time corresponding to the time window j while being in node i
- t_i - Arrival time at node i
- a_i - Opening time of time window i
- $H(i) = \{k \mid H_{i,k} \geq 0\}$ - Set of non-waiting neighbors of node i (i.e such that node k is reached within the time window)
- $\upsilon = \frac{1}{|H(i)|} \sum_{j \in H(i)} H_{i,j}$ - Average waiting time λ - Parameter to control the slope of the sigmoidal function

Lopez-Ibanez and Blum, 2009

$$\eta_{i-1,k} = \lambda_a \cdot \frac{a_{\max} - a_k}{a_{\max} - a_{\min}} + \lambda_b \cdot \frac{b_{\max} - b_k}{b_{\max} - b_{\min}} + \lambda_c \cdot \frac{c_{\max} - c_{i-1,k}}{c_{\max} - c_{\min}} \tag{7}$$

where

- a_i - Opening time of time window i
- $a_{\max} = \max_{j \in N(i)} a_j$ - Maximum time window opening time in the neighborhood of node i
- $a_{\min} = \min_{j \in N(i)} a_j$ - Minimum time window opening time in the neighborhood of node i
- b_i - Closing time of time window i
- $b_{\max} = \max_{j \in N(i)} b_j$ - Maximum time window closing time in the neighborhood of node i
- $b_{\min} = \min_{j \in N(i)} b_j$ - Minimum time window closing time in the neighborhood of node i
- $c_{i,j}$ - Travelling cost from node i - j
- $c_{\max} = \max_j c_{i,j}$ - Maximum travelling cost from node i
- $c_{\min} = \min_j c_{i,j}$ - Minimum travelling cost from node i
- $\lambda_a, \lambda_b, \lambda_c$ such that $\lambda_a + \lambda_b + \lambda_c = 1$ - Randomly picked weights

Algorithm 5 Pheromone Trails Update

```
1:  procedure UpdatePheromoneTrails(ρ, q_0)                ▷ The main procedure
2:      i ← 0
3:      j ← 0
4:      for i < n_cities do
5:          for j < i do
6:              if Random() < q_0 then
7:                  τ_ij ← (1 − ρ) · τ_ij + Δτ^Bi_{i,j}
8:              else
9:                  τ_ij ← (1 − ρ) · τ_ij + Δτ^Bo_{i,j}
10:             end if
11:             if τ_ij < τ_min then
12:                 τ_ij ← τ_min
13:             end if
14:             if τ_ij > τ_max then
15:                 τ_ij ← τ_max
16:             end if
17:             τ_ji ← τ_ij
18:             j ← j + 1
19:         end for
20:         i ← i + 1
21:     end for
22: end procedure
```

$$\Delta\tau^{Bi}_{i,j} = \begin{cases} \frac{1}{T^{\mathrm{Bi}}_d} & e_{i,j} \in T^{\mathrm{Bi}} \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

$$\Delta\tau_{i,j}^{Bo} = \begin{cases} \frac{1}{T_d^{\mathrm{Bo}}} & e_{i,j} \in T^{\mathrm{Bo}} \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

where

- $e_{i,j}$ - Edge connecting node i and j
- T_d^i - Complete tour duration of tour i
- T^{Bi} - Best tour of the current iteration
- T^{Bo} - Best tour overall

An Iterative Improvement algorithm generally starts from a candidate solution, which can be either generated randomly or using an heuristic, and improves the evaluation of the solution at each step by modifying the solution structure , until a local optimum is reached.
In the previous problem, I considered different kinds of 2-opt neighborhood, and different pivoting rules. This means that, at each step, a new solution is constructed from the current best by modifying only two solution components (with Transpose,Exchange or Insert operations) and only the first/best improving solution will become the new best soluion.
The main limitation of such kind of algorithms is that they tend to get stuck in solutions that are locally optimum but not globally.
Provided that:

- A global optimum is optimal with respect to any kind of neighborhood.
- A solution that is locally optimal with respect to a neighborhood may not be optimal with respect to other kinds of neighborhood

by dynamically changing the neighborhood type an algorithm is able to escape local optima.
This section will analyse the results of the execution of two variable neighborhood descent algorithm, based on the previously analyzed iterative improvement algorithms :

- **Standard Variable Neighborhood Descent** (i.e. Changing neighborhood when a local optimum is encoutered, until the neighborhood chain is terminated and going back to the smallest neighborhood every time the local optimum is escaped.)
- **Piped Variable Neighborhood Descent** (i.e. Using the locally optimum solution found using one neighborhood type in the chain as the initial solution for the following type.)

The same metrics as in will be used to evaluate the algorithms.

Results discussion

By looking at tables **??**, **??**, **??**, **??** one can see that:

- For some instances (e.g. $n80w20.002$,$n80w20.003$) the algorithm are not able to converge to a feasible solution, as shown in the corresponding boxplots, since the PRPD distribution is centered around 12000-15000, thus indicating the presence of at least 1 constraint violations in most of the cases.
- For some other instances (e.g. $n80w20.004$,$n80w20.005$) the algorithms are able to converge to feasible solutions and to the best-known one, but having a right-skewed distribution towards higher values of PRPD.

- For the remaining instances, except for some outlier values, the algorithms are able to converge to the best-known solution in most of the runs , even though the average PRPD is not closer to 0. This is due to the fact that the mean of a distribution is sensible to outliers and the penalisation for a constraint violations is extremely high when compared to the mean value.
- The algorithm ordering in terms of runtimes is $s.tie < p.tie < p.tei < s.tei$ for the $n80w20.X$ instances while $s.tie < p.tei < s.tei < p.tie$ for $n80w200.X$ ones. The choice to explore the Insert Neighborhood before the Exchange one allows to reduce the computation time for the $n80w20.X$ instances, with a similar solution quality.
- The algorithms are more effective on the $n80w200.X$ instances then the $n80w20.X$ once, since they have a lower percentage of infeasible runs and a lower PRPD.
- The standard variable neighborhood descent with Transpose-Insert-Exchange neighborhood chain (s.tie) outperforms all the other algorithms in terms of solution quality and runtime.
- Tables **??**, **??**, **??**, **??**, **??**, **??**, **??**, **??**, **??**, **??** contain, in any case, p-values considerably smaller than the significance level ($\alpha = 0.05$).

  This implies that the null hypothesis corresponding to the equality of the median values of the differences of the two distributions can be rejected, hence assessing the existence of a statistically significant difference among the solution quality generated by analyzed algorithms.
- By looking at the Cpu time, one can see that the instances *n80w20.X* have generally lower runtimes than the *n80w200.X* ones. They can then be considered, with respect to the variable neighborhood descent algorithms, simpler (quickier to solve) instances with respect to the latter.

Problem 2

Iterative improvement algorithms

Problem statement

Implement iterative improvement algorithms with:

- first-improvement
- best-improvement

pivoting rule for each of the three neighborhoods: transpose, exchange, and insert.
As a starting solution for iterative improvement, consider a random permutation, that is, use the method "Uninformed Random Picking" (see slides of lectures). Bonus points will be awarded for also considering an insertion heuristic as an alternative to random initialization.

1. Run the 6 resulting iterative improvement algorithms (all combinations of the two pivoting rules and the three neighborhoods) on each of the instances. Repeat each run 100 times with different seed for the random number generator.
2. Compute the following statistics for each of the 6 iterative improvement algorithms and each instance:
   - Percentage of runs with constraint violations
   - Mean penalised relative percentage deviation
   - Mean computation time
3. Produce boxplots of penalised relative percentage deviation and computation time per instance.
4. Determine using statistical tests (in this case, the Wilcoxon test), whether there is a statistically significant difference between the quality of the solutions generated by the different algorithms. In particular, compare best vs. first-improvement for each neighborhood, and exchange vs. insertion for each pivoting rule.

Metric definitions

For each algorithm k, applied on instance i, using different randomly generated seeds one have to compute:

- Number of constraint violations
- Penalised relative percentage deviation (PRPD)
- Computation time (CPU time)

In order to perform a statistical analysis of the results, each algorithm k is launched 100 times on the same instance, computing the following statistics:

- Percentage of infeasible solutions (0.x has to be interpreted as x%)
- Average Penalised relative percentage deviation.
- Average Computation time (CPU time).

For each instance i, the distributions of PRPD and Cpu Times are displayed using box plots and the Wilcoxon signed rank test is performed, in order to assess the existence of a statistically significant difference among the results obtained by the different algorithms on the same instance.

Constraint Violations In the standard formulation of the TSP problem, a solution to the problem is represented by a permutation of the different entities (solution components), that the hypothetical travelling salesman has to visit.
The best solution for the problem is the permutation that minimizes the total travelling time (distance) among the cities. The presence of time windows introduce an additional constraint on the feasibility of the solution.
In fact, each solution component has an associated time window within which it has to be visited in order to guarantee the feasiblity of the tour.
Arriving in a (city) before the opening of the corresponding time window involves a delay in the total travelling time (to wait for the time window to open) whereas the arrival after the closure of the time windows will generate a constraint violation.
Thus, a solution is feasible if and only if all the time windows constraints are met, or in other words, if there are no constraint violations.
In this case, the best solution is the feasible solution which minimizes the total travel time.

Penalised Relative Percentage Deviation The penalised relative percentage deviation (PRDP from now on) is a measure of the solution quality, with respect to the best known solution for the instance, taking into account a strong penalisation for the violation of constraints. The PRPD is computed as follows:

$$pRPD_{kri} = 100 \cdot \frac{(f_{kri} + 10^4 \cdot \Omega_{kri}) - best_i}{best_i} \tag{10}$$

Runtime The runtime is a measure of both the quality and the time complexity of the algorithm.
It is measured using the function `int clock_gettime(clockid_t clk_id, struct timespect *tp)` from the `time.h` library.
The parameter `clk_id=CLOCK_PROCESS_CPUTIME_ID`, is used to read the values from an high-resolution timer provided by the CPU for each process.
The runtime is computed (using the user defined function `ComputeRunTime`) as the difference, with a resolution of $10^{-}9$ s, from the time obtained using `clock_gettime` at the beginning and the one obtained at the end of the simulation.

Simulated Annealing

Algorithm 6 Simulated Annealing TSPTW

procedure SIMULATEDANNEALING(τ_0, n_{cities})
Require: Annealing schedule
  $s \leftarrow$ *InitialSolution*() ▷ Heuristic or random permutation
  $T \leftarrow T_0$
  while !*TerminationCondition*() **do**
    $s' \leftarrow$ *ProposalMechanism()* ▷ (Often) Uniform random choice in $N(s)$
    if *AcceptanceCriterion*(T) **then** ▷ (Often) Metropolis condition
      $s \leftarrow s'$
    end if
    $T \leftarrow$ *Update*(T) ▷ $T_{i+1} = \alpha \cdot T_i$
  end while
end procedure

Algorithm 7 Initial solution

procedure INITIAL SOLUTION
end procedure

Algorithm 8 Termination condition

procedure TERMINATION CONDITION
end procedure

$$P(s, s', T) = \begin{cases} 1 & f(s) < f(s') \\ 1 & f(s) = f(s') \wedge \Omega(s) < \Omega(s') \\ e^{\frac{f(s)-f(s')}{T}} & \text{otherwise} \end{cases} \tag{11}$$

Results discussion

By looking at tables **??**, **??**, **??**, **?? ??**, **??** on can see that:

- The neighborhood type has a strong influence on the both the time complexity of the algorithm and the generated solution quality. This is due to the size of the different neighborhoods ($n = 80$):
  - Transpose - $(n-1)$
  - Exchange - $\frac{n \cdot (n-1)}{2}$
  - Insert - $(n-1)^2$

  The different size of the neighborhoods corresponds to different degrees of exploration (diversification).
- Transpose and Exchange neighborhoods have smaller runtimes but a percentage of infeasible runs equal to 1. Both the algorithm do not allow to find a feasible solution but the Exchange algorithm constructs solutions with a better quality (reduced, but not yet null, constraint violations and total travel time).
- Insert is the only neighborhood type that allows to generate solutions that are both feasible and closer to the global optima.
- The first-improvement pivoting rule is generally slower than the best-improvement one, when considering the same neighborhood type. This is due to the fact that, with the first-improvement pivoting rule, smaller improvement are made to the solution at each iteration, thus requiring and higher number of iteration to converge to a local optima, with respect to the case where the best improvement is chosen at each time step.
- The quality of the solutions generated using the first-improvement pivoting rule is slightly better thant those generated using the best-improvement one.
- Tables **??**, **??**, **??**, **??**, **??**, **??**, **??**, **??**, **??**, **??** contain, in any case, p-values considerably smaller than the significance level ($\alpha = 0.05$).

Algorithm 9 Proposal mechanism

procedure PROPOSAL MECHANISM
end procedure

Algorithm 10 Acceptance Criterion

procedure ACCEPTANCE CRITERION
end procedure

Algorithm 11 Update according to annealing schedule

procedure UPDATE(T)
$T \leftarrow \alpha \cdot T$
end procedure

This implies that the null hypothesis corresponding to the equality of the median values of the differences of the two distributions can be rejected, hence assessing the existence of a statistically significant difference among the solution quality generated by analyzed algorithms.

- By looking at the Cpu time, one can easily see that the instances *n80w20.X* have lower runtimes than the *n80w200.X* ones. They can then be considered, with respect to the iterative improvement algorithms, simpler instances with respect to the latter.

Conclusions

By combining the results from the previous analysis:

- The Iterative Improvement algorithms based on Transpose and Exchange neighborhoods do not allow to find feasible solutions hence they should not be considered for a practical application.
- On the other hand, the solution quality generated by the Iterative Improvement algorithm with the Insert neighborhood is similar to those generated by the VND algorithms, regardless of the instances.
- On this set of instances, the VND algorithms have a lower runtime than the Iterative Improvement one using Insert neighborhood, hence being similar the resulting solution quality, they should be preferred to the Iterative Improvement ones.
- The algorithm that showed the best performances in terms of solution quality and runtime is the Standard Variable Neighborhood Descent with Transpose-Insert-Exchange neighborhood chain.
- The usage of average statistics as metrics to measure the quality of the algorithms is strongly biased by the presence of outliers (penalisation, in this case).

References

[1] J-L Deneubourg, Serge Aron, Simon Goss, and Jacques Marie Pasteels. The self-organizing exploratory pattern of the argentine ant. *Journal of insect behavior*, 3(2):159–168, 1990.

[2] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Artificial ants as a computational intelligence technique. *IEEE computational intelligence magazine*, pages 265–276, 2006.

[3] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant system: optimization by a colony of cooperating agents. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 26(1):29–41, 1996.

[4] Manuel López-Ibáñez and Christian Blum. Beam-aco for the travelling salesman problem with time windows. *Computers & operations research*, 37(9):1570–1583, 2010.

[5] Thomas Stützle and Holger H Hoos. Max–min ant system. *Future Generation Computer Systems*, 16(8):889–914, 2000.