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 22, 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:

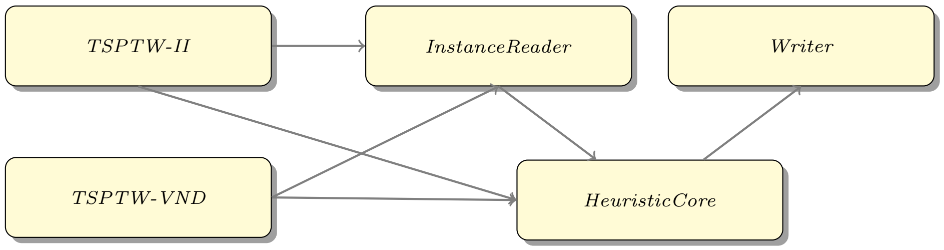


Figure 1: Simulation Code Structure

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 stochastic local search method.
The algorithm is 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 (CITE Deneubourg).
The ants are able to communicate indirectly 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 to choose the right path.
The convergence to one of the paths will occur as a consequence of the self-reinforcing pheromone deposit mechanism (i.e the more pheromone is deposited, the more ants will follow the pheromone trail deposing even more pheromone).
The Ant Colony Optimization method arise from the implementation of this mechanism to optimization problems (CITE Dorigo), with the following analogies:

- Ants ≡ Virtual ants
- Pheromone ≡ Virtual pheromone
- Environment ≡ Search space

Algorithm 1 Ant Colony Optimization for TSPTW - Outline

```
1: procedure ACO($\alpha, \beta, \rho, \tau_0, n_{ants}, n_{cities}, t_{max}$)    ▷ The main procedure
2:     InitalizePheromoneTrail($\tau_0, n_{cities}$) ▷ Set the pheromone values for all the solution components to $\tau_0$
3:     while !TerminationCondition($t_{max}$) do ▷ The termination condition can be either based on time or solution quality
4:         for all ants do
5:             ConstructSolution($\alpha, \beta$)
6:             LocalSearch(TO DEFINE)
7:             UpdatePheromoneTrails($\rho$)
8:         end for
9:     end while
10:    return solution    ▷ The gcd is b
11:
12: end procedure
```

Algorithm 2 Pheromone Initialization

```
1: procedure InitalizePheromoneTrail($\tau_0, n_{cities}$)    ▷ The main procedure
2:     $i \leftarrow 0$
3:     $j \leftarrow 0$
4:     for $i < n_{cities}$ do
5:         for $j < i$ do
6:             $\tau_{ij} \leftarrow \tau_0$
7:             $\tau_{ji} \leftarrow \tau_{ij}$
8:             $j \leftarrow j + 1$
9:         end for
10:        $i \leftarrow i + 1$
11:    end for
12: end procedure
```

$$p_k = \frac{[\tau_{i-1,k}]^\alpha \cdot [\eta_{i-1,k}]^\beta}{\sum_{j \in N(s_{i-1})} [\tau_{i-1,j}]^\alpha \cdot [\eta_{i-1,j}]^\beta} \tag{1}$$

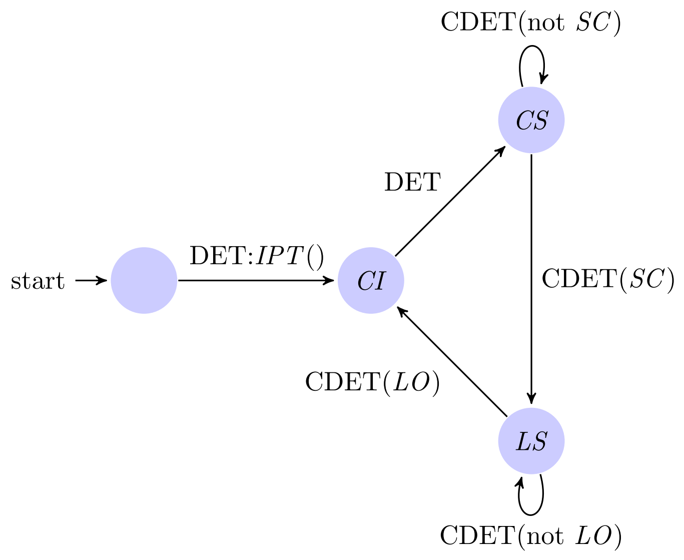
$$\Delta_\tau = DEFINE \tag{2}$$

Algorithm 3 Solution Construction

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

Algorithm 4 Pheromone Trails Update

```
1: procedure UpdatePheromoneTrails($\rho$)                      ▷ The main procedure
2:     $i \leftarrow 0$
3:     $j \leftarrow 0$
4:     for $i < n_{cities}$ do
5:         for $j < i$ do
6:             $\tau_{ij} \leftarrow (1 - \rho) \cdot \tau_{ij} + \Delta_\tau$
7:             $\tau_{ji} \leftarrow \tau_{ij}$
8:             $j \leftarrow j + 1$
9:         end for
10:        $i \leftarrow i + 1$
11:    end for
12: end procedure
```



- $IPT() \equiv \mathit{InitializePheromoneTrail}()$
- $SC \equiv \mathit{SolutionComplete}()$
- $LO \equiv \mathit{LocalOptimum}()$

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.

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).