INFO-H-413 - Heuristic Optimization: Implementation Exercise #1 The Traveling Salesman Problem with Time Windows

Prof. T. Stüetzle

Jacopo De Stefani

May 20, 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:

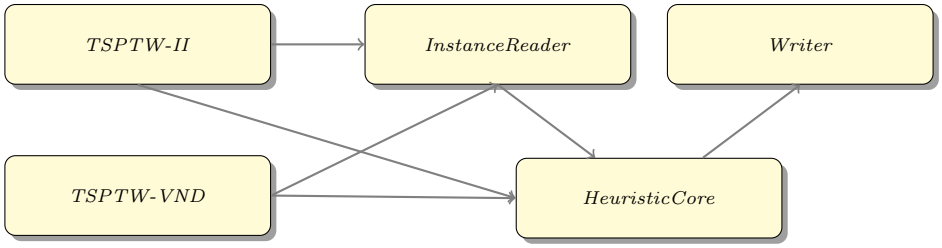

Figure 1: Simulation Code Structure

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

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:

$$pRPD_{kri} = 100 \cdot \frac{(f_{kri} + 10^4 \cdot \Omega_{kri}) - best_i}{best_i} \tag{1}$$

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.

Experiment results

n80w20.001

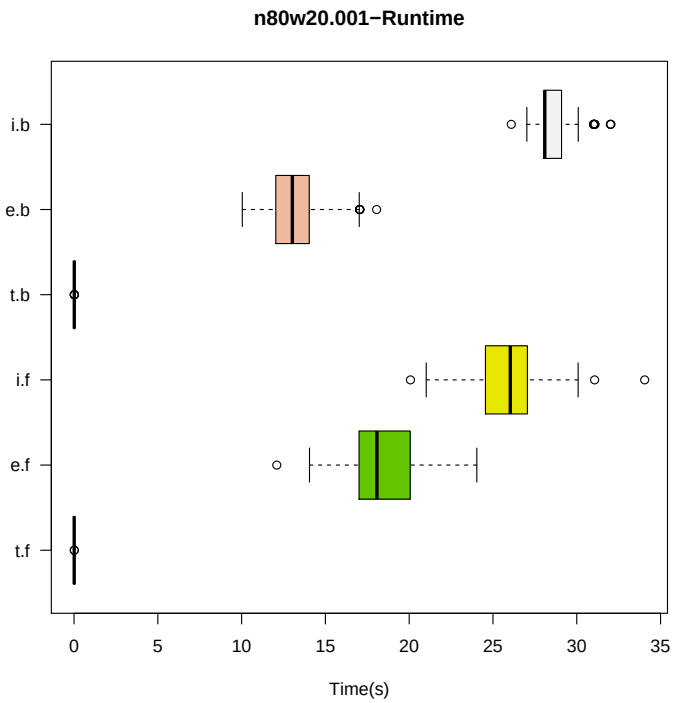

Figure 2: n80w20.001 - Runtime boxplots for the different iterative improvement algorithms

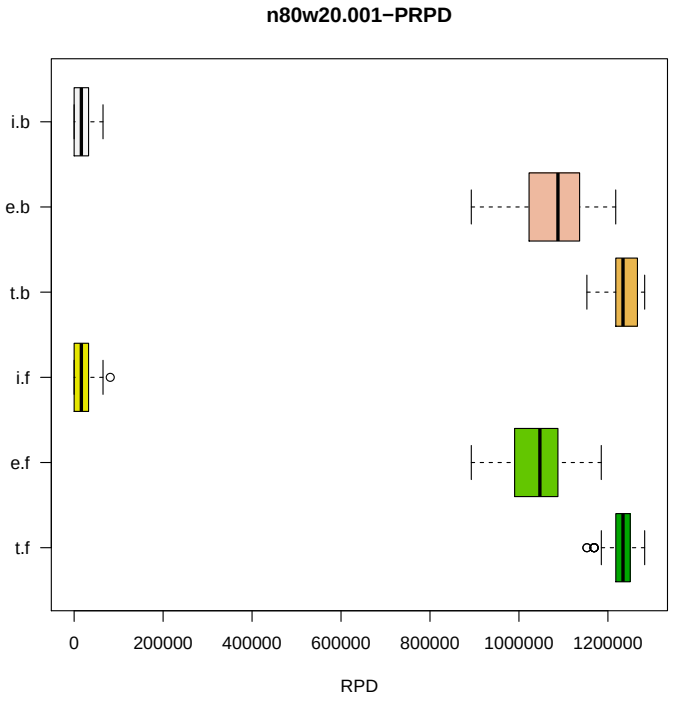

Figure 3: n80w20.001 - PRPD boxplots for the different iterative improvement algorithms

Test	P-Value
First vs best - Transpose	9.74631639820544e-18
First vs best - Exchange	2.04966732989559e-17
First vs best - Insert	1.74838327736385e-15
Exchange vs Insert - First	3.95591160889952e-18
Exchange vs Insert - Best	3.9556885406462e-18

Table 1: n80w20.001 - Results of Wilcoxon paired signed rank test

n80w20.002

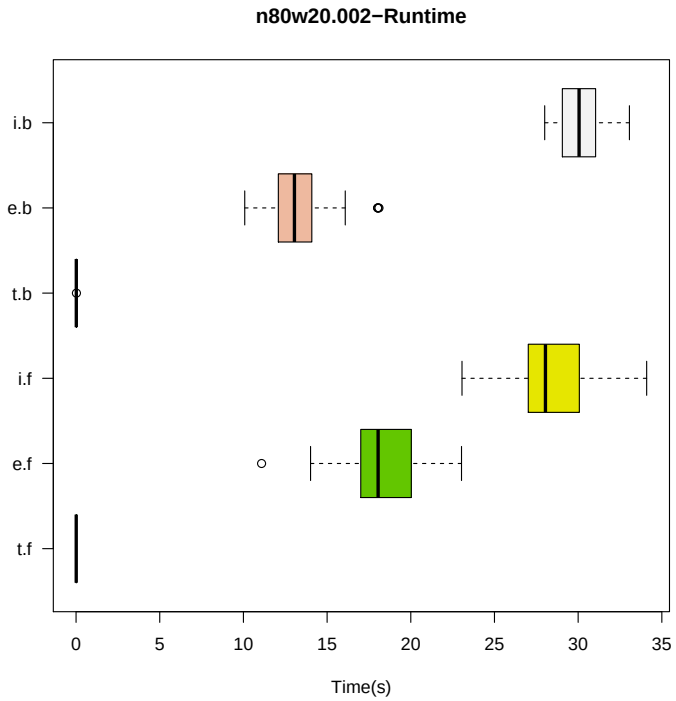

Figure 4: n80w20.002 - Runtime boxplots for the different iterative improvement algorithms

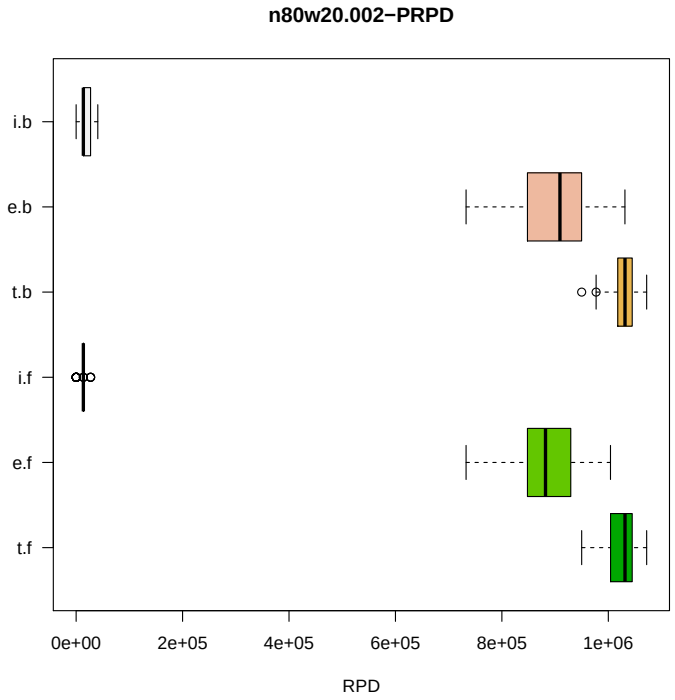

Figure 5: n80w20.002 - PRPD boxplots for the different iterative improvement algorithms

Test	**P-Value**
First vs best - Transpose	3.95591160889952e-18
First vs best - Exchange	1.61703099974578e-17
First vs best - Insert	2.39050570998277e-07
Exchange vs Insert - First	3.95591160889952e-18
Exchange vs Insert - Best	3.9556885406462e-18

Table 2: n80w20.002 - Results of Wilcoxon paired signed rank test

n80w20.003

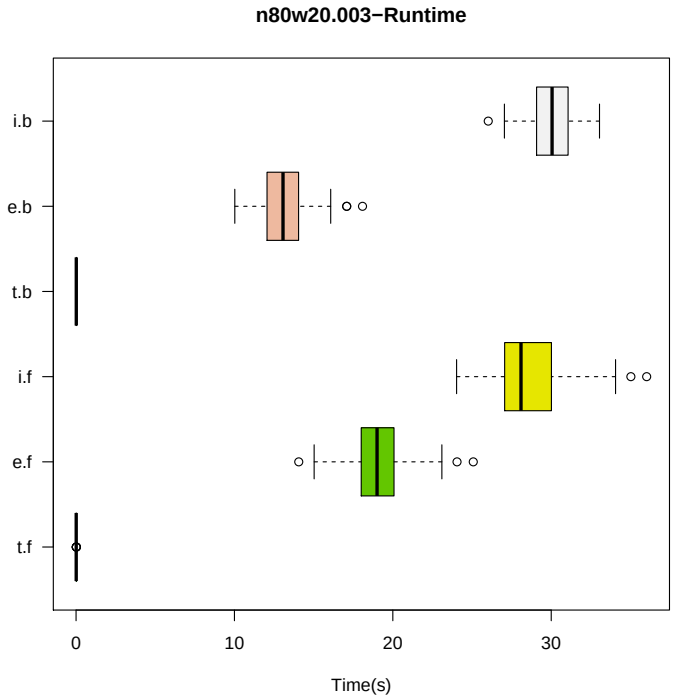

Figure 6: n80w20.003 - Runtime boxplots for the different iterative improvement algorithms

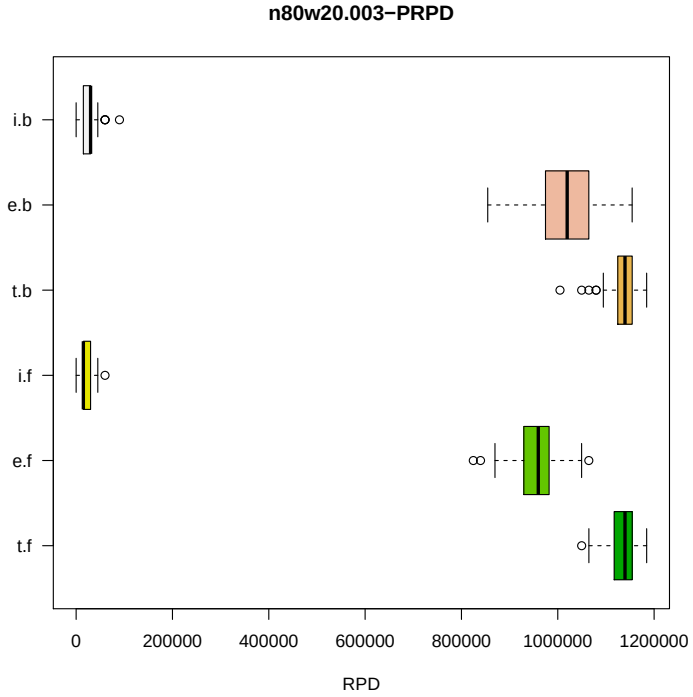

Figure 7: n80w20.003 - PRPD boxplots for the different iterative improvement algorithms

Test	P-Value
First vs best - Transpose	3.95591160889952e-18
First vs best - Exchange	6.21747363653032e-18
First vs best - Insert	6.2952945764779e-08
Exchange vs Insert - First	3.9556885406462e-18
Exchange vs Insert - Best	3.95591160889952e-18

Table 3: n80w20.003 - Results of Wilcoxon paired signed rank test

n80w20.004

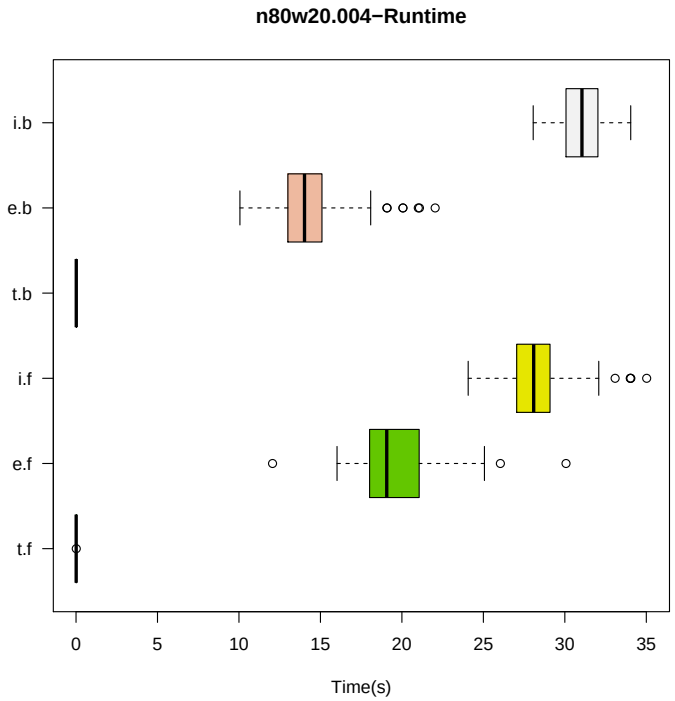

Figure 8: n80w20.004 - Runtime boxplots for the different iterative improvement algorithms

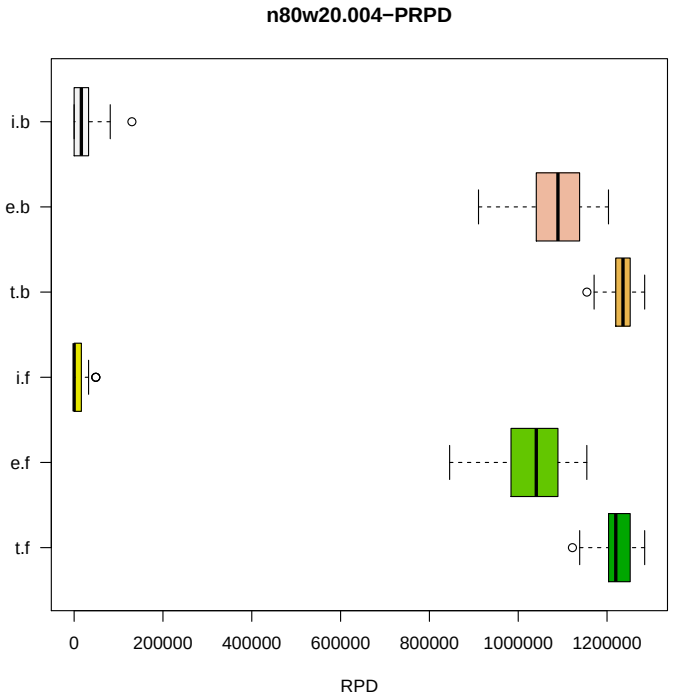

Figure 9: n80w20.001 - PRPD boxplots for the different iterative improvement algorithms

Test	**P-Value**
First vs best - Transpose	4.33123080260219e-18
First vs best - Exchange	1.5356610755813e-16
First vs best - Insert	4.27702026764362e-14
Exchange vs Insert - First	5.59593516960623e-18
Exchange vs Insert - Best	3.95591160889952e-18

Table 4: n80w20.004 - Results of Wilcoxon paired signed rank test

n80w20.005

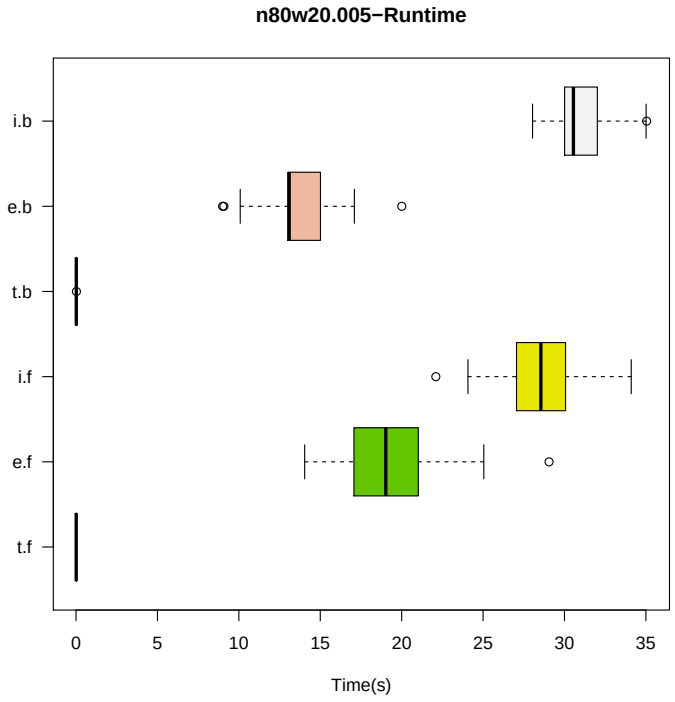

Figure 10: n80w20.005 - Runtime boxplots for the different iterative improvement algorithms

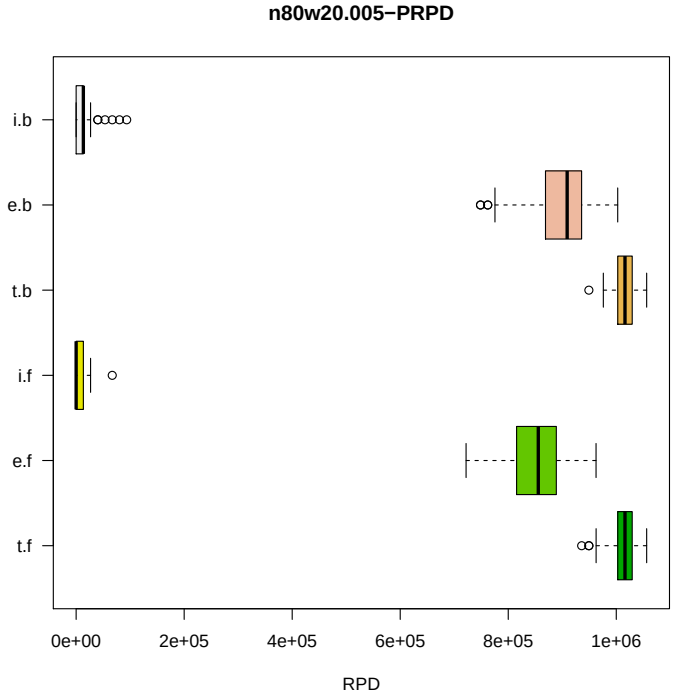

Figure 11: n80w20.005 - PRPD boxplots for the different iterative improvement algorithms

Test	P-Value
First vs best - Transpose	4.46398542390809e-18
First vs best - Exchange	4.74166029806301e-18
First vs best - Insert	4.0369131744045e-10
Exchange vs Insert - First	4.74166029806301e-18
Exchange vs Insert - Best	3.95591160889952e-18

Table 5: n80w20.005 - Results of Wilcoxon paired signed rank test

n80w200.001

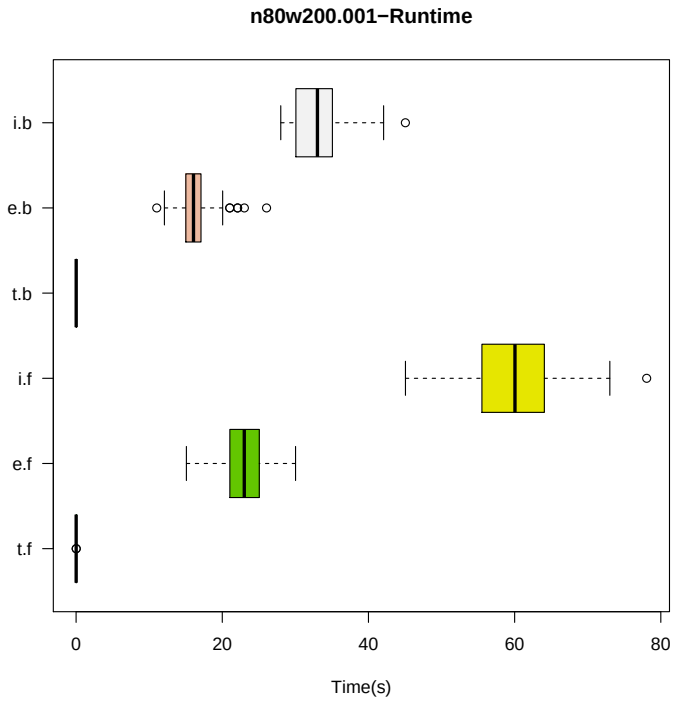

Figure 12: n80w200.001 - Runtime boxplots for the different iterative improvement algorithms

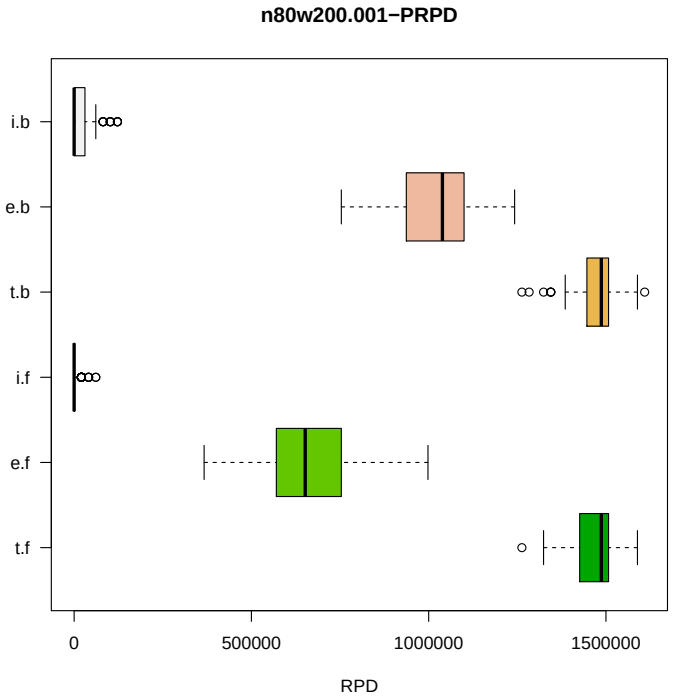

Figure 13: n80w200.001 - PRPD boxplots for the different iterative improvement algorithms

Test	**P-Value**
First vs best - Transpose	4.07730530936212e-18
First vs best - Exchange	2.17457280454137e-17
First vs best - Insert	3.95591160889952e-18
Exchange vs Insert - First	3.95591160889952e-18
Exchange vs Insert - Best	3.95591160889952e-18

Table 6: n80w200.001 - Results of Wilcoxon paired signed rank test

n80w200.002

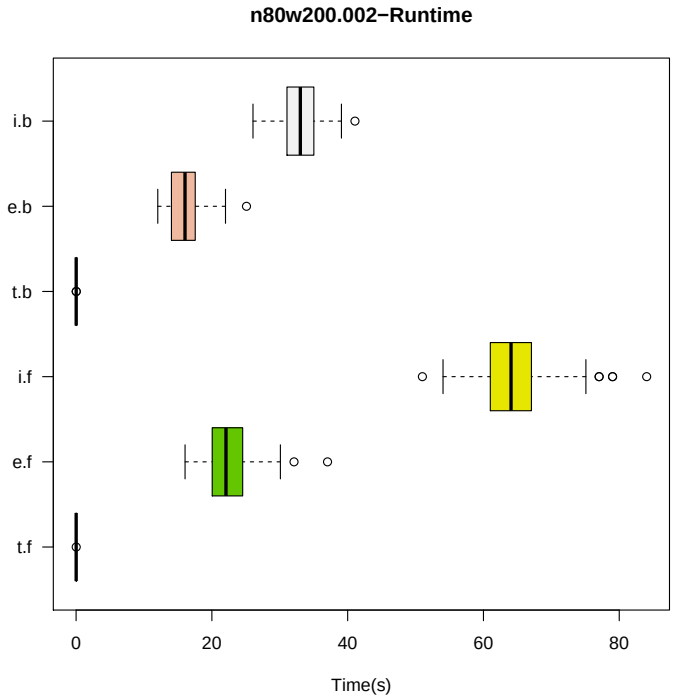

Figure 14: n80w200.002 - Runtime boxplots for the different iterative improvement algorithms

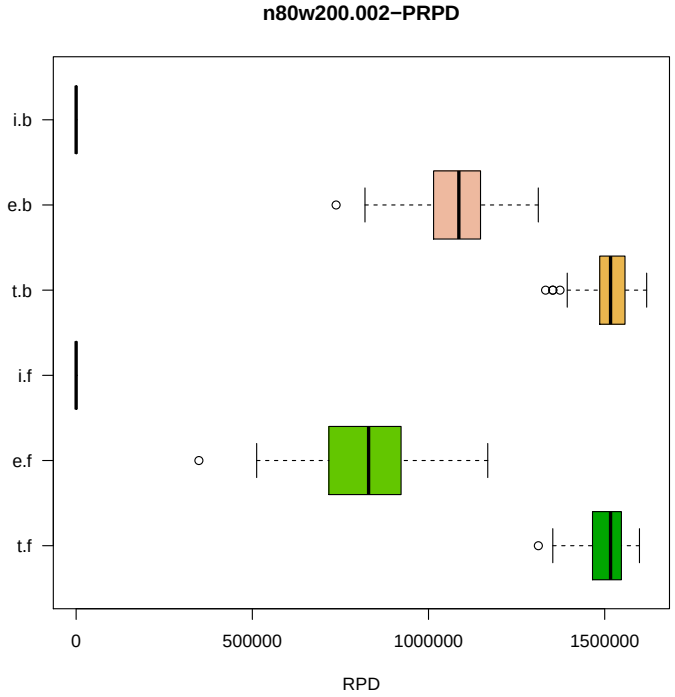

Figure 15: n80w200.002 - PRPD boxplots for the different iterative improvement algorithms

Test	P-Value
First vs best - Transpose	5.19043683699158e-18
First vs best - Exchange	4.6720416035814e-17
First vs best - Insert	3.95591160889952e-18
Exchange vs Insert - First	3.95591160889952e-18
Exchange vs Insert - Best	3.95591160889952e-18

Table 7: n80w200.002 - Results of Wilcoxon paired signed rank test

n80w200.003

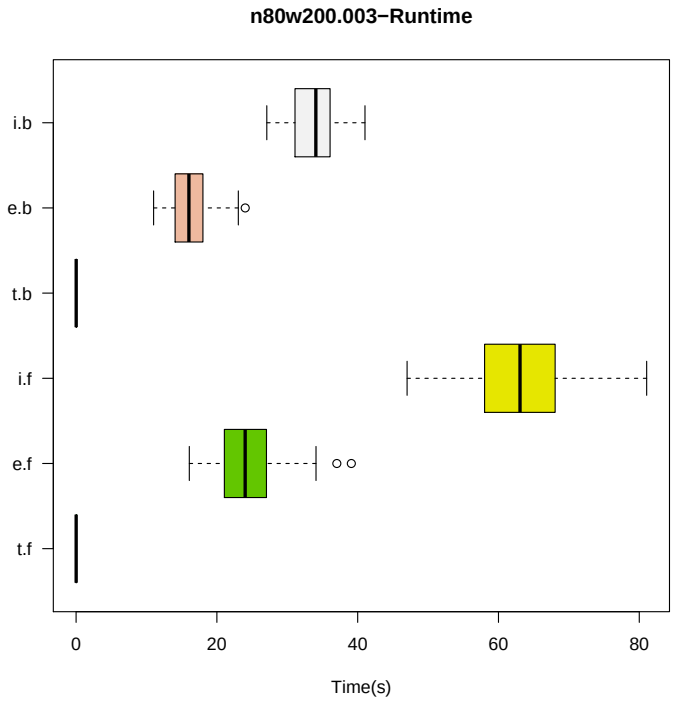

Figure 16: n80w200.003 - Runtime boxplots for the different iterative improvement algorithms

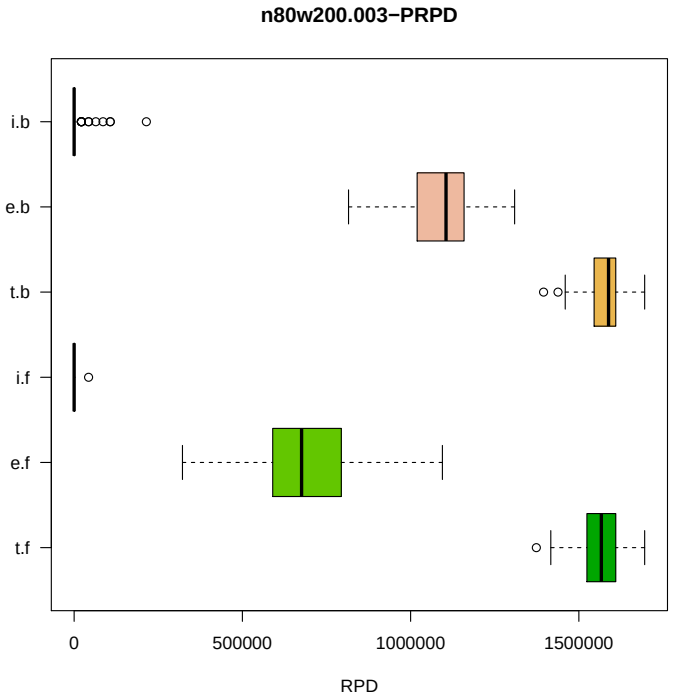

Figure 17: n80w200.003 - PRPD boxplots for the different iterative improvement algorithms

Test	**P-Value**
First vs best - Transpose	4.33123080260219e-18
First vs best - Exchange	7.01070639830382e-18
First vs best - Insert	3.95591160889952e-18
Exchange vs Insert - First	3.95591160889952e-18
Exchange vs Insert - Best	3.95591160889952e-18

Table 8: n80w200.003 - Results of Wilcoxon paired signed rank test

n80w200.004

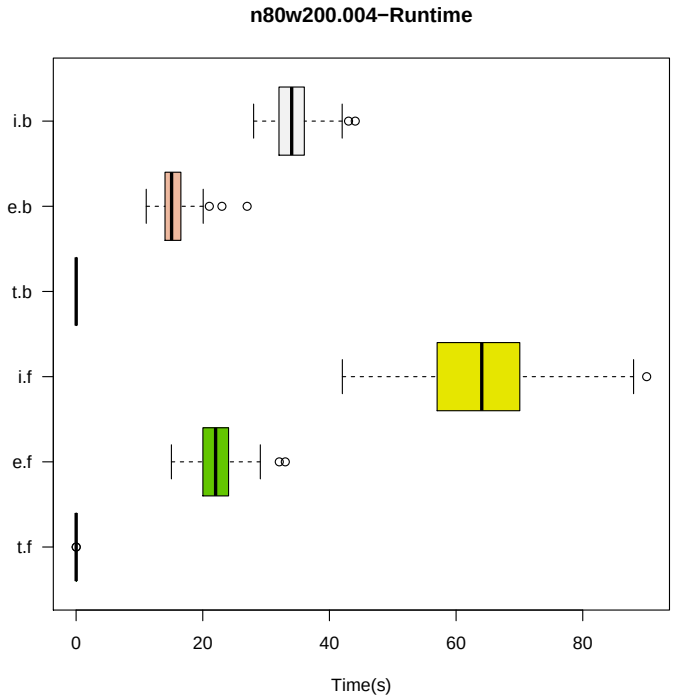

Figure 18: n80w200.004 - Runtime boxplots for the different iterative improvement algorithms

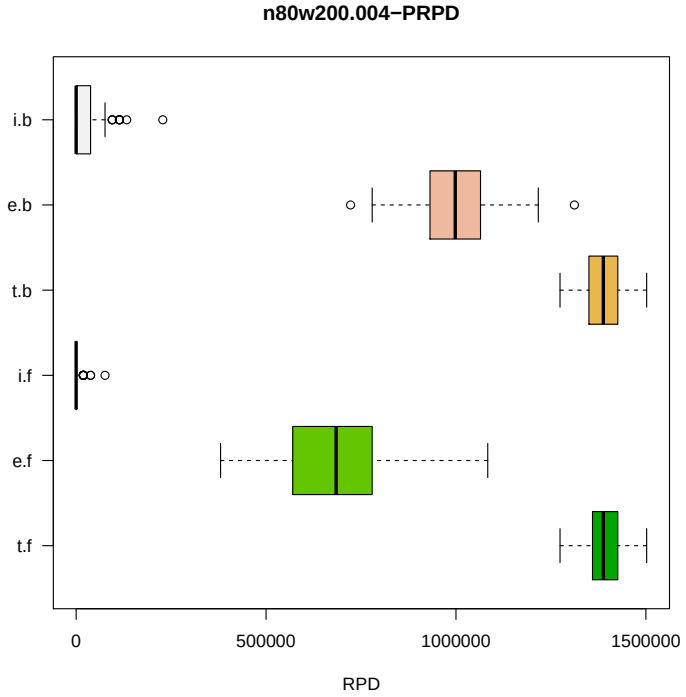

Figure 19: n80w200.001 - PRPD boxplots for the different iterative improvement algorithms

Test	P-Value
First vs best - Transpose	4.33123080260219e-18
First vs best - Exchange	2.4473398426105e-17
First vs best - Insert	3.95591160889952e-18
Exchange vs Insert - First	3.95591160889952e-18
Exchange vs Insert - Best	3.95591160889952e-18

Table 9: n80w200.004 - Results of Wilcoxon paired signed rank test

n80w200.005

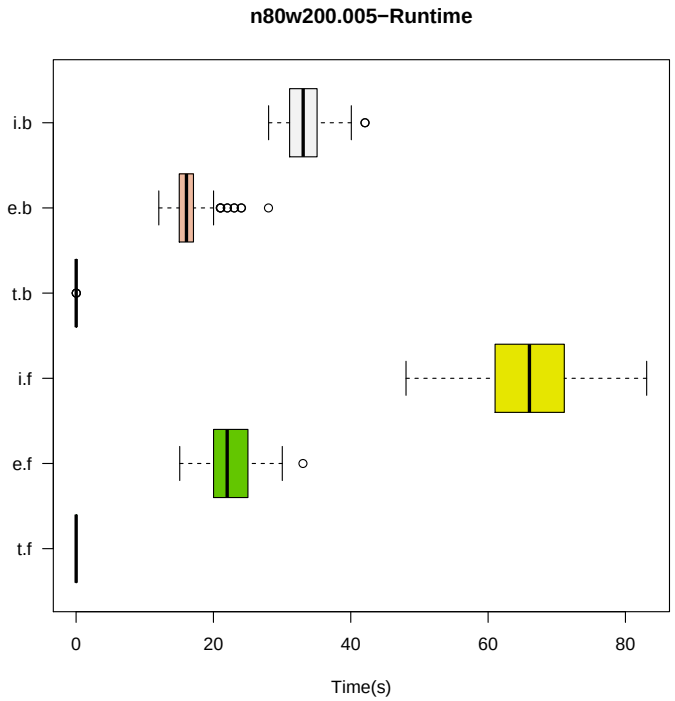

Figure 20: n80w200.005 - Runtime boxplots for the different iterative improvement algorithms

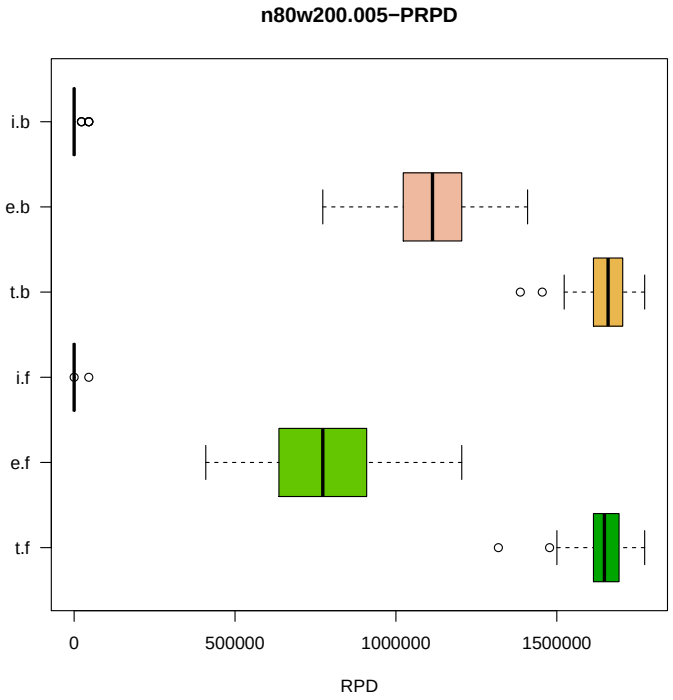

Figure 21: n80w200.005 - PRPD boxplots for the different iterative improvement algorithms

Test	P-Value
First vs best - Transpose	4.74166029806301e-18
First vs best - Exchange	1.40854365025687e-16
First vs best - Insert	3.95591160889952e-18
Exchange vs Insert - First	3.95591160889952e-18
Exchange vs Insert - Best	3.95591160889952e-18

Table 10: n80w200.005 - Results of Wilcoxon paired signed rank test

Statistics

Transpose-First Improvement

Instance	% Infeasible	$\bar{PRDP}$	$\bar{Runtime}$
n80w20.001	1	1229712.6	0.0100035563
n80w20.002	1	1028075.17	0.0095843375
n80w20.003	1	1132968.5	0.0098099452
n80w20.005	1	1010681.79	0.0097989762
n80w20.004	1	1226174.6	0.0094877067
n80w200.001	1	1467634.1	0.0098152878
n80w200.002	1	1504523.7	0.0099101404
n80w200.004	1	1388037.6	0.0098893615
n80w200.003	1	1567210.5	0.0097697727
n80w200.005	1	1644110.9	0.0097550971

Table 11: Statistics summary for iterative improvement algorithm with Transpose neighborhood and First Improvement pivoting rule

Transpose-Best Improvement

Instance	% Infeasible	$\bar{PRDP}$	$\bar{Runtime}$
n80w20.001	1	1236039.3	0.014545497
n80w20.002	1	1033090.48	0.0145450422
n80w20.003	1	1137758.7	0.014536776
n80w20.005	1	1014818.46	0.014947609
n80w20.004	1	1232996	0.015151286
n80w200.001	1	1476994.8	0.0147534638
n80w200.002	1	1511281	0.014884921
n80w200.004	1	1392023.3	0.0146445173
n80w200.003	1	1575355.5	0.0143582468
n80w200.005	1	1653419.6	0.0150098656

Table 12: Statistics summary for iterative improvement algorithm with Transpose neighborhood and Best Improvement pivoting rule

Exchange-First Improvement

Instance	% Infeasible	$\bar{PRDP}$	$\bar{Runtime}$
n80w20.001	1	1035718.78	18.390814
n80w20.002	1	884386.3	18.198632
n80w20.003	1	956518.77	18.913801
n80w20.005	1	849054.52	19.043496
n80w20.004	1	1030411.56	19.660314
n80w200.001	1	661322.18	23.117446
n80w200.002	1	813339.6	22.552976
n80w200.004	1	679489.06	22.068859
n80w200.003	1	697450.03	24.338257
n80w200.005	1	760027.65	22.348975

Table 13: Statistics summary for iterative improvement algorithm with Exchange neighborhood and First Improvement pivoting rule

Exchange-Best Improvement

Instance	% Infeasible	$\bar{PRDP}$	$\bar{Runtime}$
n80w20.001	1	1086217.68	13.091453
n80w20.002	1	901762.05	13.365301
n80w20.003	1	1017243.86	13.267128
n80w20.005	1	895720.53	13.5824245
n80w20.004	1	1084080.4	14.189582
n80w200.001	1	1022433.76	16.418229
n80w200.002	1	1075649.94	16.028712
n80w200.004	1	994703.67	15.518766
n80w200.003	1	1094460.9	16.16696
n80w200.005	1	1110949.86	16.566802

Table 14: Statistics summary for iterative improvement algorithm with Exchange neighborhood and Best Improvement pivoting rule

Insert-First Improvement

Instance	% Infeasible	$\bar{PRDP}$	$\bar{Runtime}$
n80w20.001	0.65	16070.27322082	25.88279
n80w20.002	0.83	11803.71462682	28.509938
n80w20.003	0.83	21587.617	28.579453
n80w20.005	0.32	4945.642107	28.672083
n80w20.004	0.49	10894.01867489	28.474429
n80w200.001	0.22	6119.9927045	59.544212
n80w200.002	0	11.2561521	64.580306
n80w200.004	0.11	3049.80805246	63.942238
n80w200.003	0.01	437.87774695	63.687806
n80w200.005	0.01	464.62990671	66.084369

Table 15: Statistics summary for iterative improvement algorithm with Insert neighborhood and First Improvement pivoting rule

Insert-Best Improvement

Instance	% Infeasible	$\bar{PRDP}$	$\bar{Runtime}$
n80w20.001	0.56	14772.44442862	28.769951
n80w20.002	0.84	16009.51797952	30.159812
n80w20.003	0.89	26984.256	30.197019
n80w20.005	0.52	10159.3062354	30.776906
n80w20.004	0.59	18861.32619536	31.013
n80w200.001	0.37	21195.2243605	33.031965
n80w200.002	0	13.2172159	33.011011
n80w200.004	0.42	23019.4946845	34.123681
n80w200.003	0.12	7740.06242146	34.080592
n80w200.005	0.07	2516.0079072	33.450537

Table 16: Statistics summary for iterative improvement algorithm with Insert neighborhood and Best Improvement pivoting rule

Results discussion

By looking at tables 11, 12, 13, 14 15, 16 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 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 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 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.

Heuristic generation of the initial solution

In addition to the randomly generated initial function, I developed the `GenerateHeuristicInitialSolution()` which construct the initial solution according to an heuristic which aims to minimize the number of constraint violations.
The general outline of the algorithm is as follows:

1. Construct a solution by ordering the solution components in an ascending way according to their time windows closing time
2. For a number of times equal to the solution size divided by 10:
   (a) Select a random solution component in the solution
   (b) Select a neighborhood size randomly.
   (c) Shuffle the elements in the chosen neighborhood of the chosen solution

Due to the lack of time, I was only able to test the heuristic function on a limited number of instances. The same metrics as in will be used to evaluate the algorithms.

n80w200.001

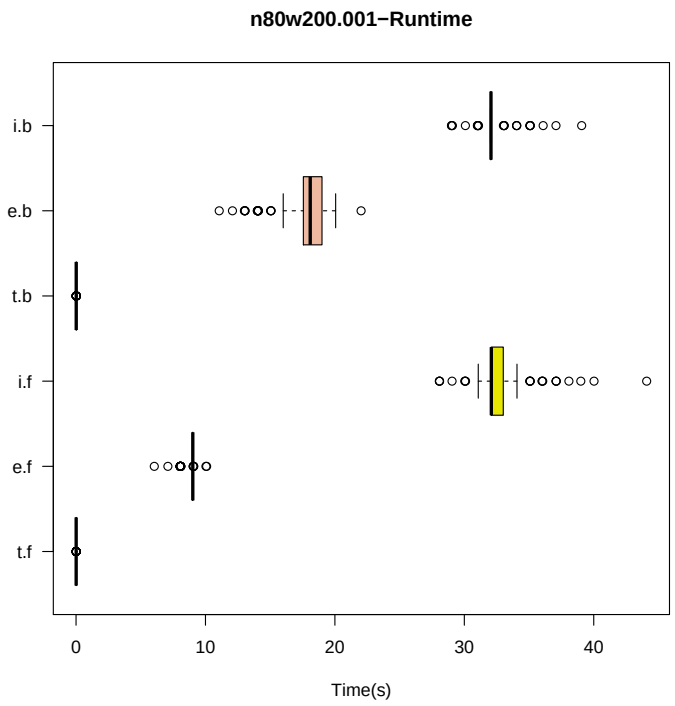

Figure 22: n80w200.001 - Runtime boxplots for the different iterative improvement algorithms with heuristic initialization

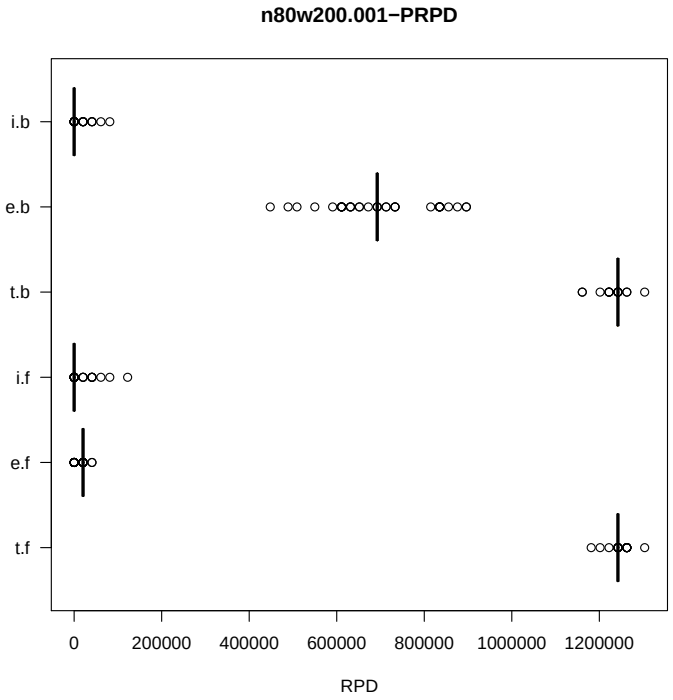

Figure 23: n80w200.001 - PRPD boxplots for the different iterative improvement algorithms with heuristic initialization

Test	**P-Value**
First vs best - Transpose	3.95591160889952e-18
First vs best - Exchange	3.95591160889952e-18
First vs best - Insert	7.16468868599392e-06
Exchange vs Insert - First	3.95591160889952e-18
Exchange vs Insert - Best	3.9550194074242e-18

Table 17: n80w200.001 - Results of Wilcoxon paired signed rank test

n80w200.002

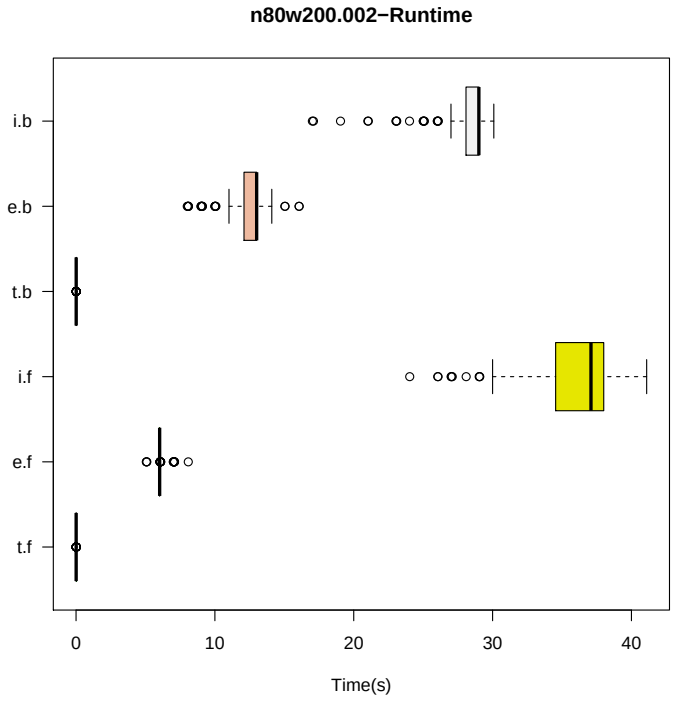

Figure 24: n80w200.002 - Runtime boxplots for the different iterative improvement algorithms with heuristic initialization

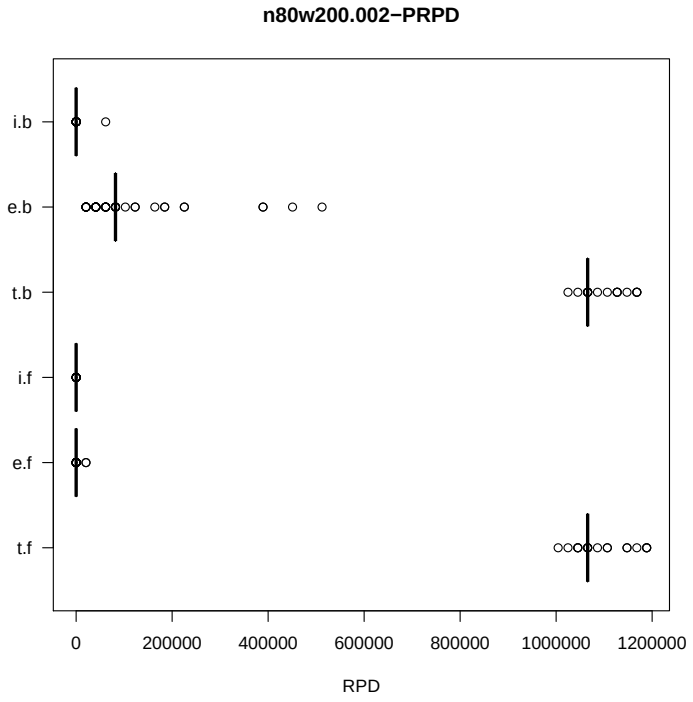

Figure 25: n80w200.002 - PRPD boxplots for the different iterative improvement algorithms with heuristic initialization

Test	**P-Value**
First vs best - Transpose	3.95591160889952e-18
First vs best - Exchange	3.95591160889952e-18
First vs best - Insert	1.5011633635878e-17
Exchange vs Insert - First	3.95591160889952e-18
Exchange vs Insert - Best	3.9552424399092e-18

Table 18: n80w200.002 - Results of Wilcoxon paired signed rank test

n80w200.003

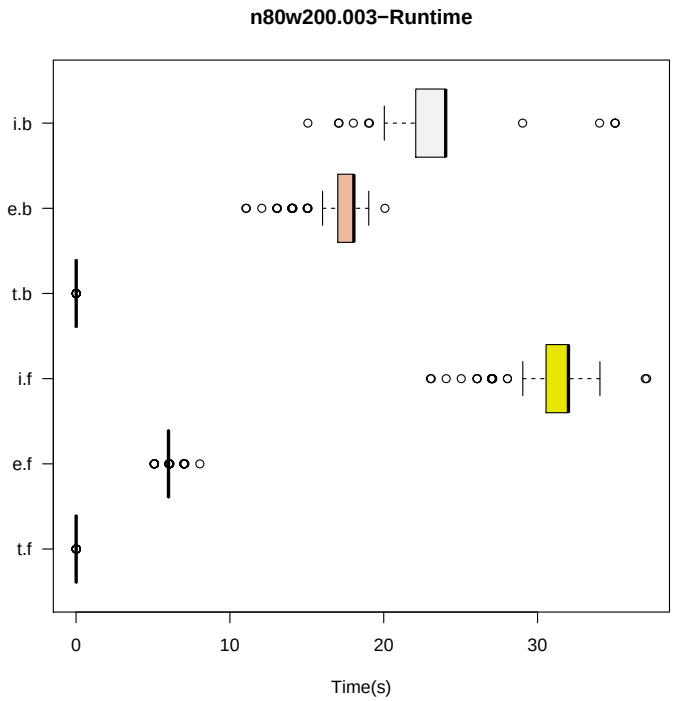

Figure 26: n80w200.003 - Runtime boxplots for the different iterative improvement algorithms with heuristic initialization

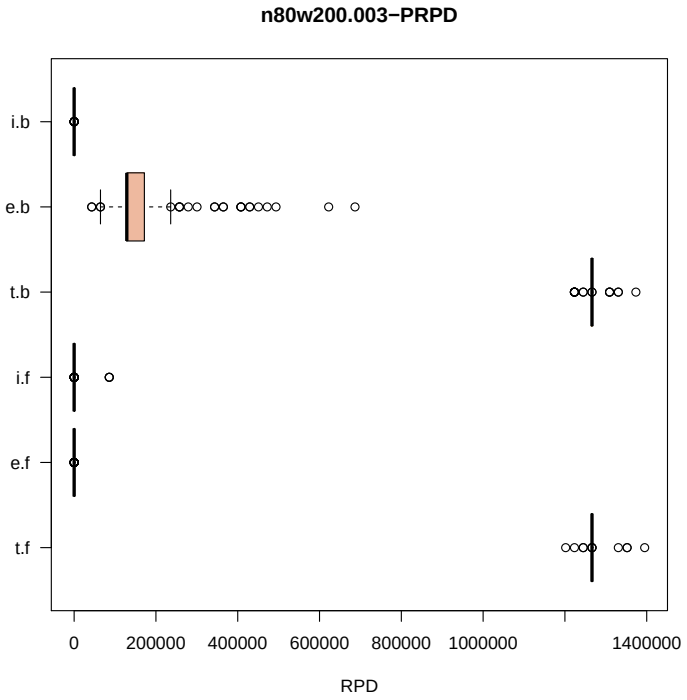

Figure 27: n80w200.003 - PRPD boxplots for the different iterative improvement algorithms with heuristic initialization

Test	**P-Value**
First vs best - Transpose	3.95591160889952e-18
First vs best - Exchange	3.95591160889952e-18
First vs best - Insert	2.88431649979563e-16
Exchange vs Insert - First	3.9556885406462e-18
Exchange vs Insert - Best	4.33074349739998e-18

Table 19: n80w200.003 - Results of Wilcoxon paired signed rank test

Statistics

Transpose-First Improvement

Instance	**% Infeasible**	**$\bar{\text{PRDP}}$**	**$\bar{\text{Runtime}}$**
n80w200.003	1	1268365.4	0.0065871265
n80w200.002	1	1071407.7	0.007598828
n80w200.001	1	1243295.2	0.0093113496

Table 20: Statistics summary for iterative improvement algorithm with Transpose neighborhood and First Improvement pivoting rule

Transpose-Best Improvement

Instance	% Infeasible	$\bar{PRDP}$	$\bar{Runtime}$
n80w200.003	1	1266853.5	0.0106505552
n80w200.002	1	1071417.8	0.011013054
n80w200.001	1	1240638.1	0.013889943

Table 21: Statistics summary for iterative improvement algorithm with Transpose neighborhood and Best Improvement pivoting rule

Exchange-First Improvement

Instance	% Infeasible	$\bar{PRDP}$	$\bar{Runtime}$
n80w200.003	0	27.068636	6.0144574
n80w200.002	0.02	432.979367	6.1109123
n80w200.001	0.87	18146.162472	8.842327

Table 22: Statistics summary for iterative improvement algorithm with Exchange neighborhood and First Improvement pivoting rule

Exchange-Best Improvement

Instance	% Infeasible	$\bar{PRDP}$	$\bar{Runtime}$
n80w200.003	1	180715.273	17.098637
n80w200.002	1	92647.622	12.3893899
n80w200.001	1	694332.77	17.575257

Table 23: Statistics summary for iterative improvement algorithm with Exchange neighborhood and Best Improvement pivoting rule

Insert-First Improvement

Instance	% Infeasible	$\bar{PRDP}$	$\bar{Runtime}$
n80w200.003	0.03	2587.6121429	30.841123
n80w200.002	0	9.9979528	35.672697
n80w200.001	0.1	4897.5213208	32.775532

Table 24: Statistics summary for iterative improvement algorithm with Insert neighborhood and First Improvement pivoting rule

Insert-Best Improvement

n80w200.003	0	4.3454898	23.452399
n80w200.002	0.01	630.9635938	27.922795
n80w200.001	0.1	3680.5235577	32.277408

Table 25: Statistics summary for iterative improvement algorithm with Insert neighborhood and Best Improvement pivoting rule

Results discussion

By looking at tables 20, 21, 22, 23 24, 25 on can see that:

- The only neighborhood type which does not allow to generate feasible solution is the Transpose one.
- By using the heuristic, also the algorithm using the Exchange neighbohood is able to generate feasible solutions that are close to the best known value.
- The use of the heuristic allows for a consistent reduction of the runtime, which becomes on average on half ot the runtime of the algorithm with random initiaalisation on the same instances.
- The solution quality of the generated solutions also benefits from the introduction of an heurstic initialization.
- Tables 17, 18, 19 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.

Problem 2

Variable neighborhood descent algorithms

Problem statement

Implement both a standard and piped variable neighborhood descent (VND) algorithm (concatenation of the underlying iterative improvement algorithms; see lecture slides). Consider the two possible (reasonable) orderings of the iterative improvement algorithms (indicated by the neighborhood relation they use):

- transpose, exchange, insert
- transpose, insert, exchange

Implement the 4 VND algorithms (all combinations of standard and piped and the two neighborhood orders) using first-improvement and random initialization.

Introduction

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.

Experiment results

n80w20.001

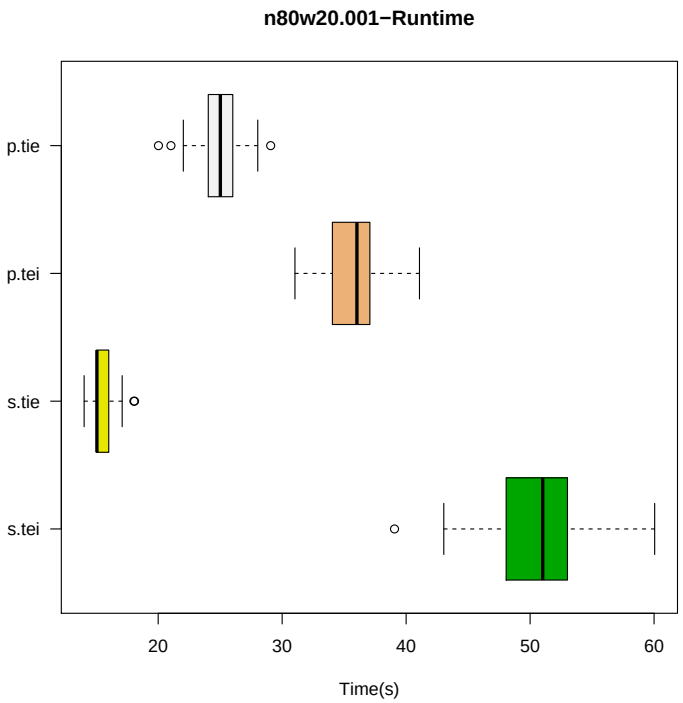

Figure 28: n80w20.001 - Runtime boxplots for the different variable neighborhood descent algorithms

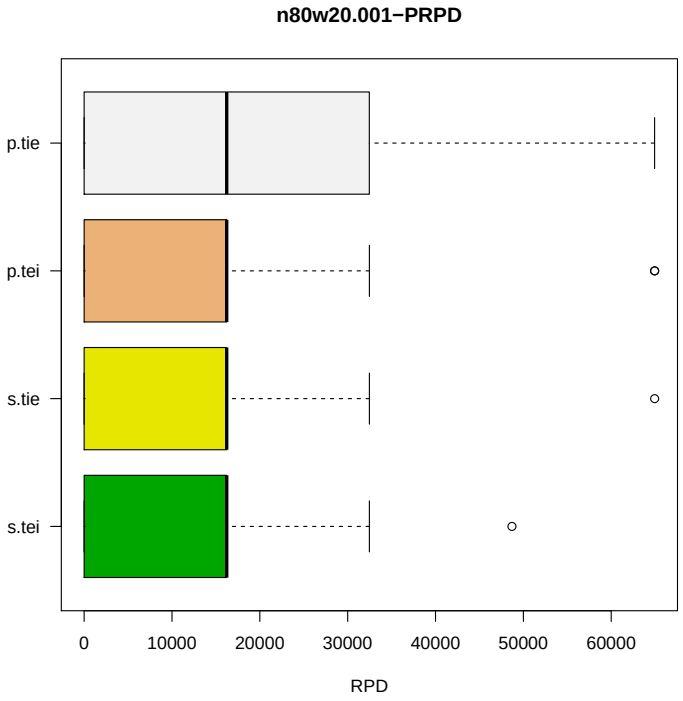

Figure 29: n80w20.001 - PRPD boxplots for the different variable neighborhood descent algorithms

Test	P-Value
Tei vs Tie - Standard	3.95591160889952e-18
Tei vs Tie - Piped	3.9556885406462e-18
Standard vs Piped - Tei	3.95591160889952e-18
Standard vs Piped - Tie	3.95591160889952e-18

Table 26: n80w20.001 - Results of Wilcoxon paired signed rank test

n80w20.002

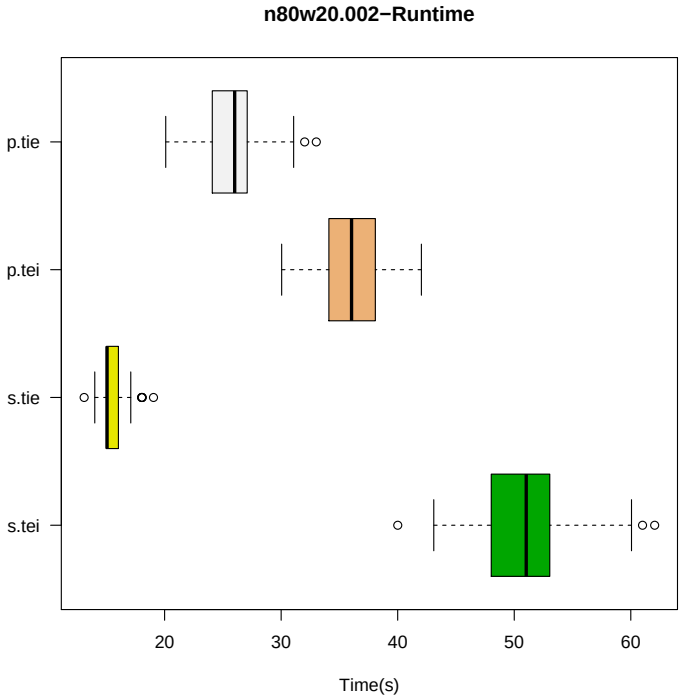

Figure 30: n80w20.002 - Runtime boxplots for the different variable neighborhood descent algorithms

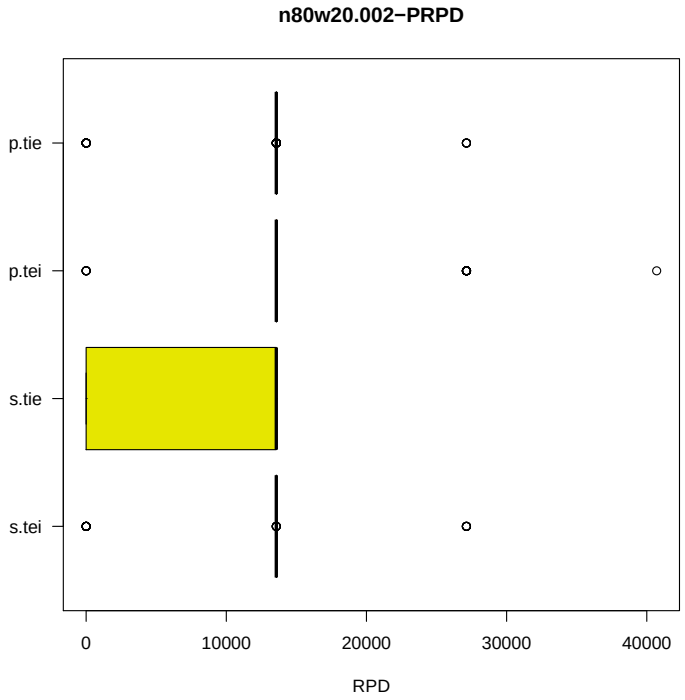

Figure 31: n80w20.002 - PRPD boxplots for the different variable neighborhood descent algorithms

Test	**P-Value**
Tei vs Tie - Standard	3.9556885406462e-18
Tei vs Tie - Piped	3.95591160889952e-18
Standard vs Piped - Tei	3.95591160889952e-18
Standard vs Piped - Tie	3.95591160889952e-18

Table 27: n80w20.002 - Results of Wilcoxon paired signed rank test

n80w20.003

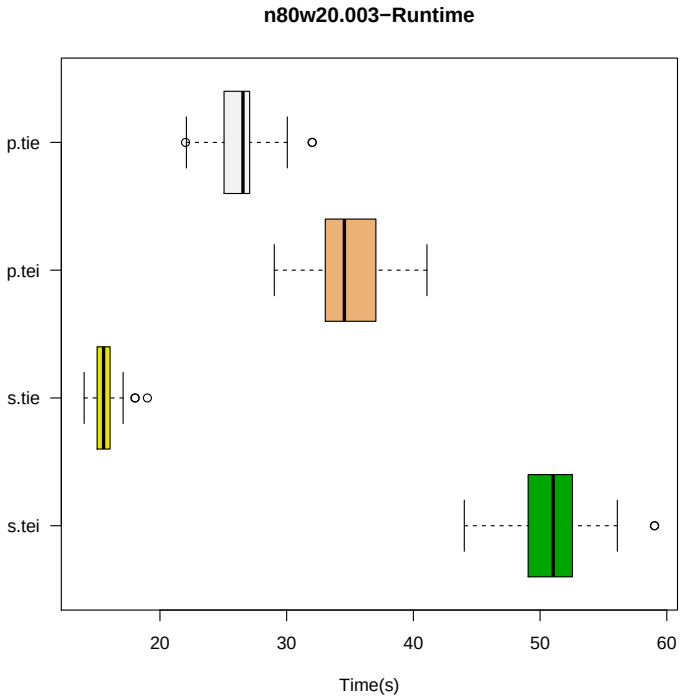

Figure 32: n80w20.003 - Runtime boxplots for the different variable neighborhood descent algorithms

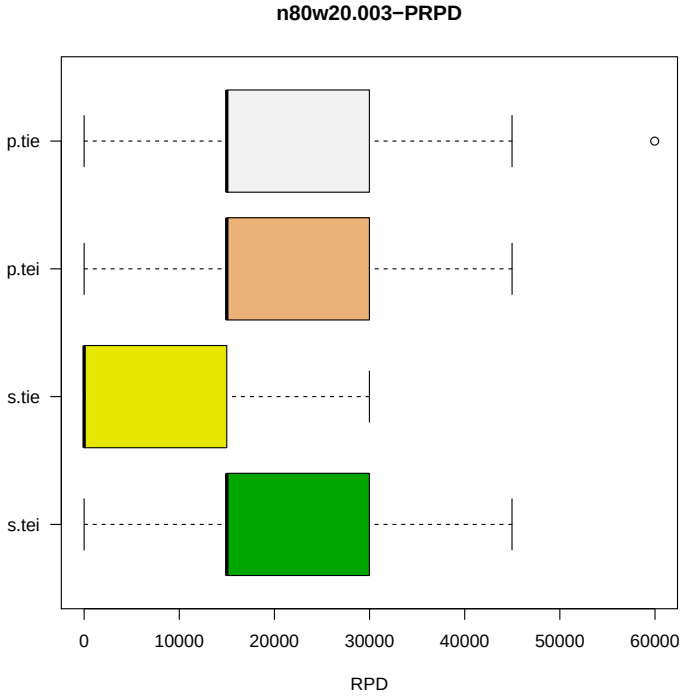

Figure 33: n80w20.003 - PRPD boxplots for the different variable neighborhood descent algorithms

Test	P-Value
Tei vs Tie - Standard	3.9552424399092e-18
Tei vs Tie - Piped	3.95591160889952e-18
Standard vs Piped - Tei	3.95591160889952e-18
Standard vs Piped - Tie	3.95591160889952e-18

Table 28: n80w20.003 - Results of Wilcoxon paired signed rank test

n80w20.004

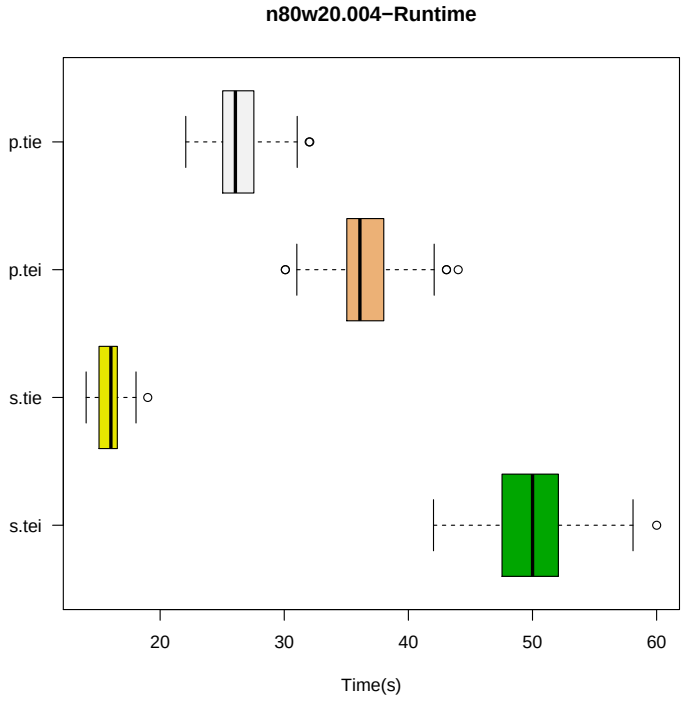

Figure 34: n80w20.004 - Runtime boxplots for the different variable neighborhood descent algorithms

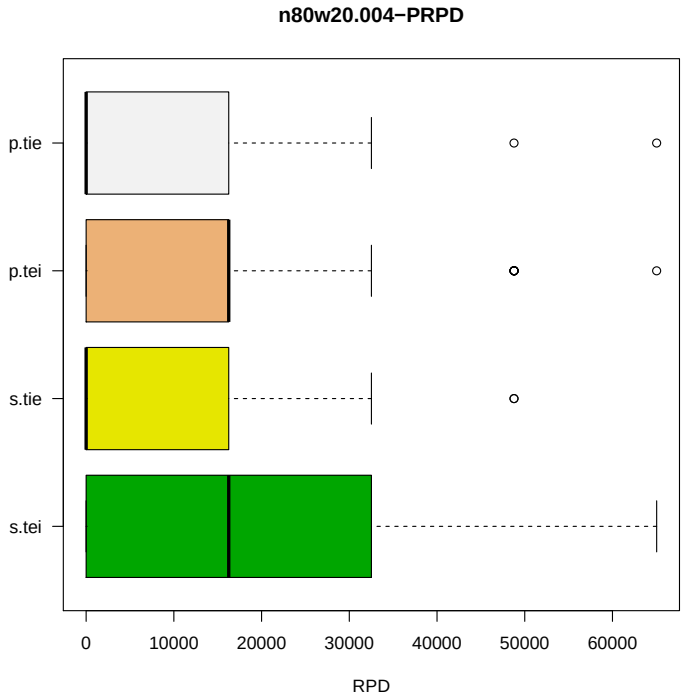

Figure 35: n80w20.004 - PRPD boxplots for the different variable neighborhood descent algorithms

Test	**P-Value**
Tei vs Tie - Standard	3.95591160889952e-18
Tei vs Tie - Piped	3.95591160889952e-18
Standard vs Piped - Tei	3.95591160889952e-18
Standard vs Piped - Tie	3.95591160889952e-18

Table 29: n80w20.004 - Results of Wilcoxon paired signed rank test

n80w20.005

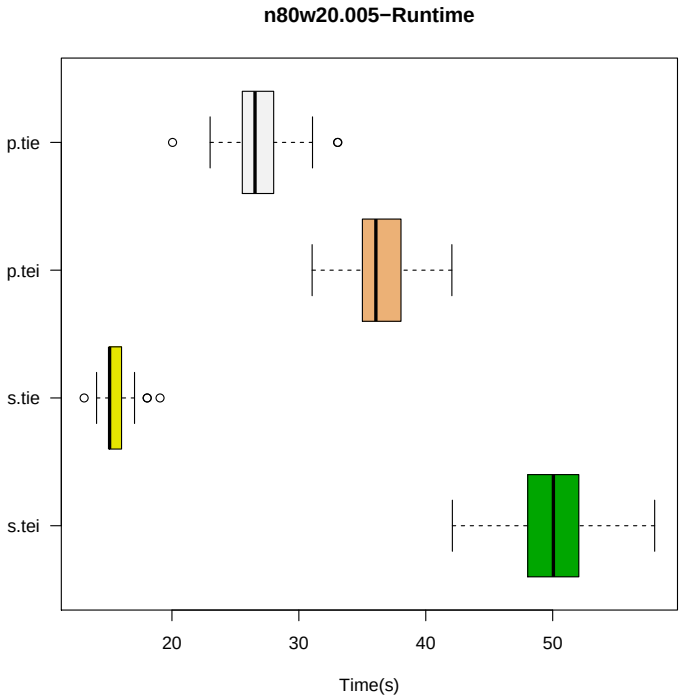

Figure 36: n80w20.005 - Runtime boxplots for the different variable neighborhood descent algorithms

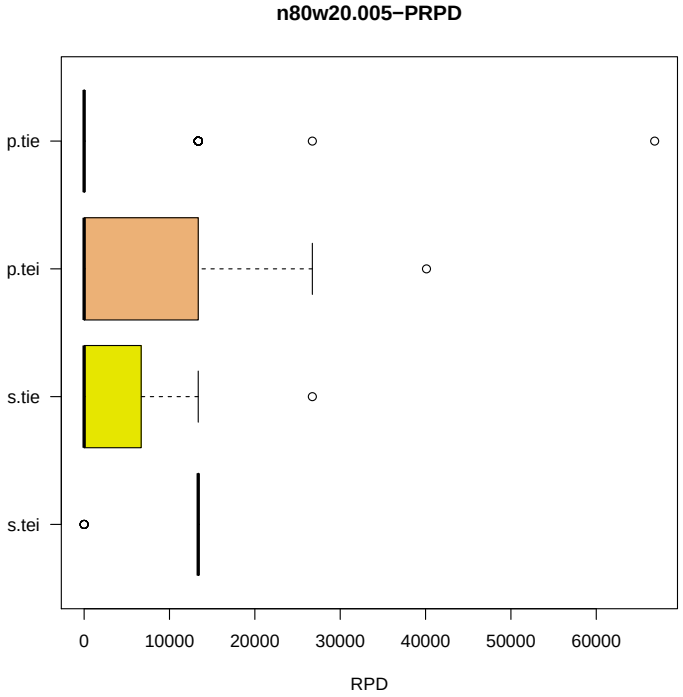

Figure 37: n80w20.001 - PRPD boxplots for the different variable neighborhood descent algorithms

Test	P-Value
Tei vs Tie - Standard	3.95591160889952e-18
Tei vs Tie - Piped	3.95591160889952e-18
Standard vs Piped - Tei	3.95591160889952e-18
Standard vs Piped - Tie	3.95591160889952e-18

Table 30: n80w20.005 - Results of Wilcoxon paired signed rank test

n80w200.001

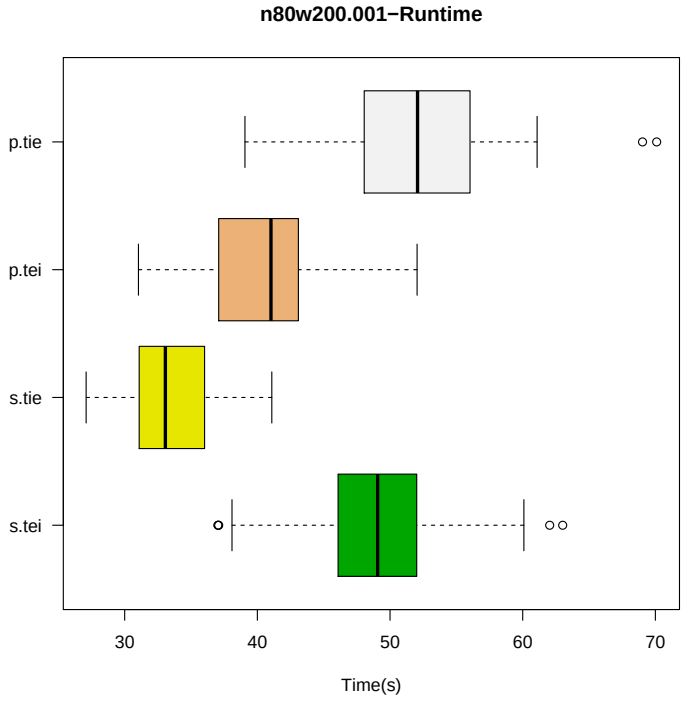

Figure 38: n80w200.001 - Runtime boxplots for the different variable neighborhood descent algorithms

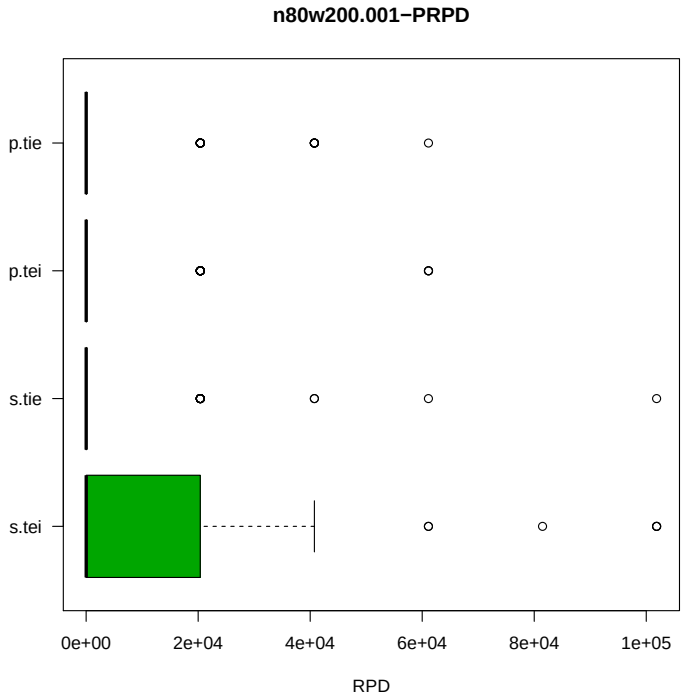

Figure 39: n80w200.001 - PRPD boxplots for the different variable neighborhood descent algorithms

Test	**P-Value**
Tei vs Tie - Standard	4.07730530936212e-18
Tei vs Tie - Piped	2.92094064174088e-17
Standard vs Piped - Tei	2.72456795287507e-16
Standard vs Piped - Tie	3.95591160889952e-18

Table 31: n80w200.001 - Results of Wilcoxon paired signed rank test

n80w200.002

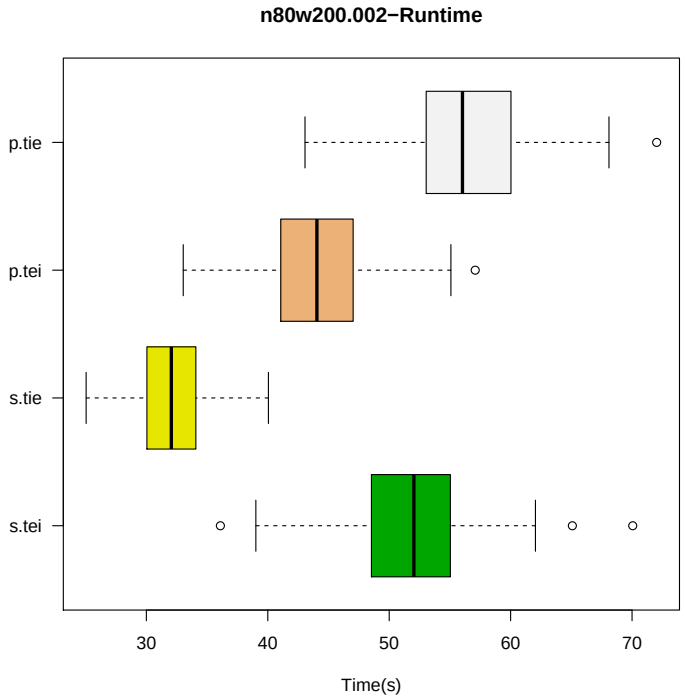

Figure 40: n80w200.002 - Runtime boxplots for the different variable neighborhood descent algorithms

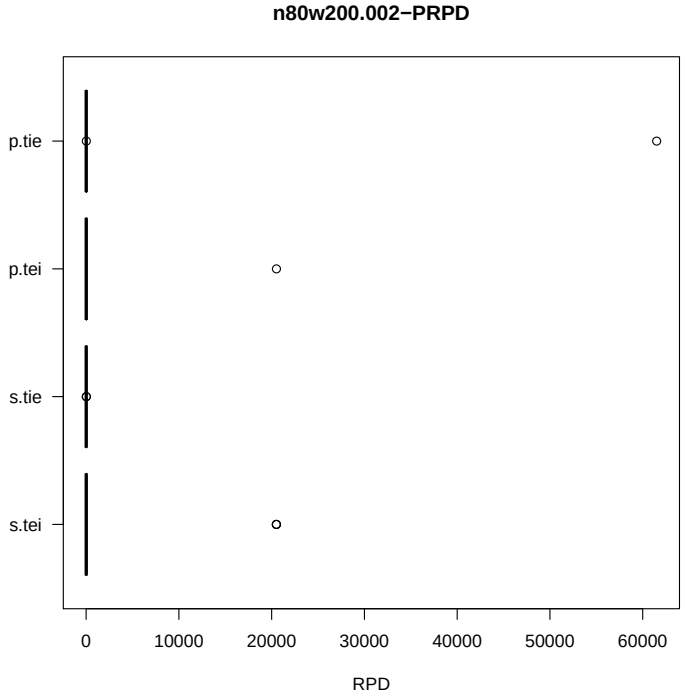

Figure 41: n80w200.002 - PRPD boxplots for the different variable neighborhood descent algorithms

Test	P-Value
Tei vs Tie - Standard	3.95591160889952e-18
Tei vs Tie - Piped	1.52379449675399e-17
Standard vs Piped - Tei	1.74838327736385e-15
Standard vs Piped - Tie	3.95591160889952e-18

Table 32: n80w200.002 - Results of Wilcoxon paired signed rank test

n80w200.003

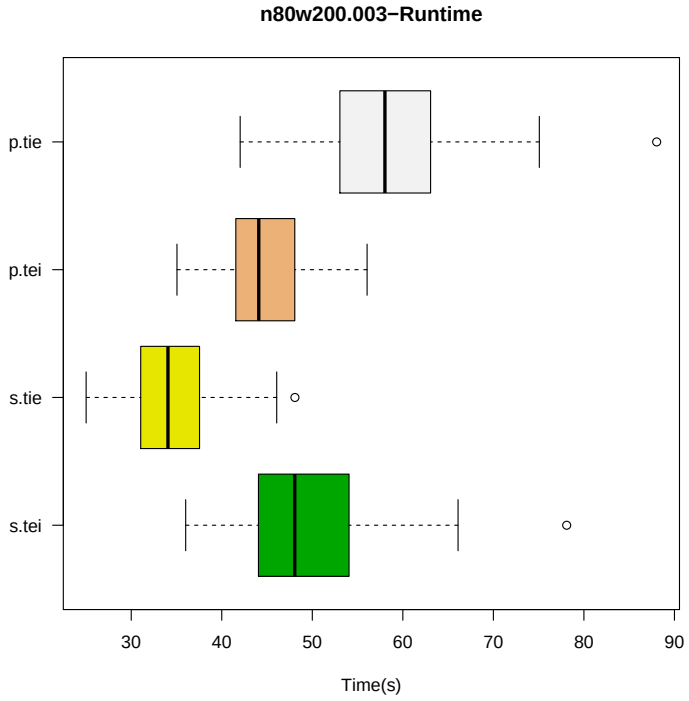

Figure 42: n80w200.003 - Runtime boxplots for the different variable neighborhood descent algorithms

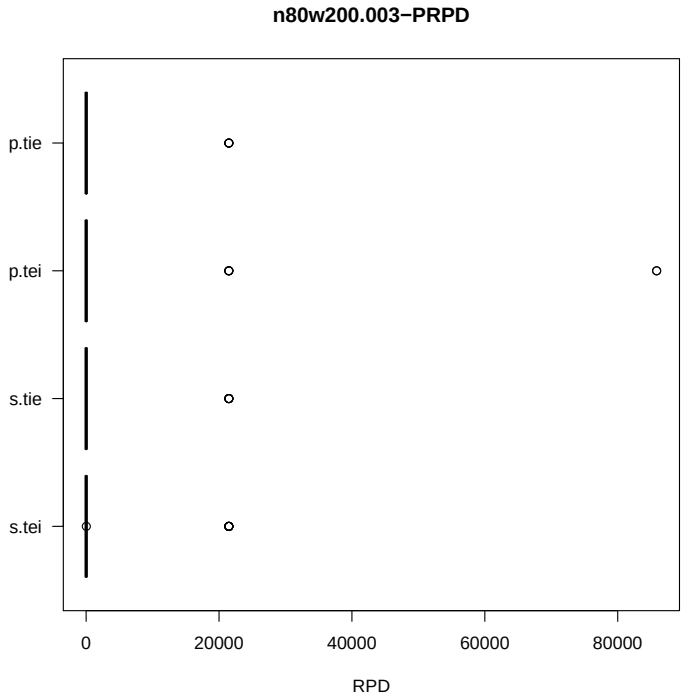

Figure 43: n80w200.003 - PRPD boxplots for the different variable neighborhood descent algorithms

Test	**P-Value**
Tei vs Tie - Standard	2.04955667109233e-17
Tei vs Tie - Piped	2.59611565456869e-17
Standard vs Piped - Tei	1.50422804122146e-07
Standard vs Piped - Tie	3.95591160889952e-18

Table 33: n80w200.003 - Results of Wilcoxon paired signed rank test

n80w200.004

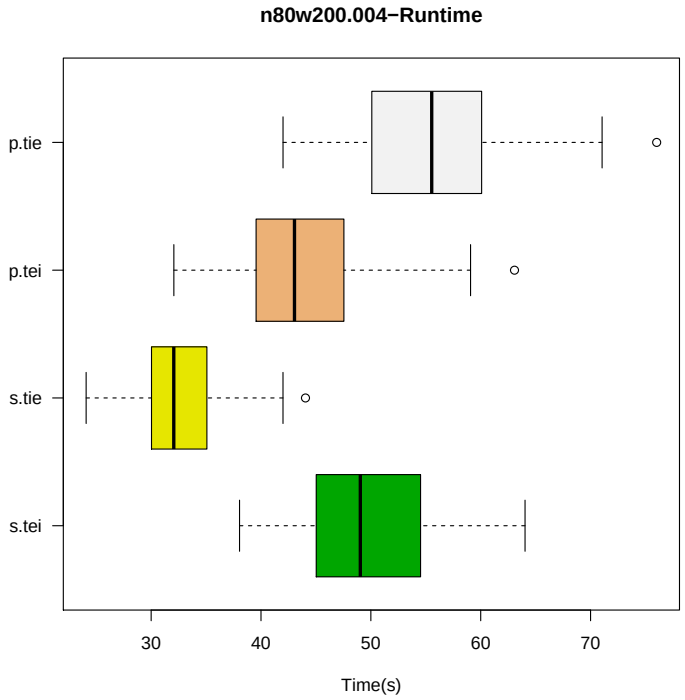

Figure 44: n80w200.004 - Runtime boxplots for the different variable neighborhood descent algorithms

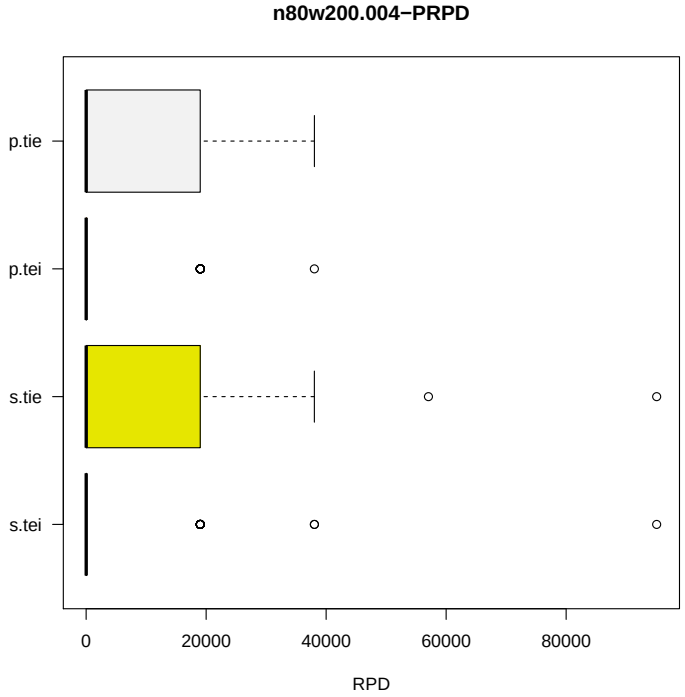

Figure 45: n80w200.004 - PRPD boxplots for the different variable neighborhood descent algorithms

Test	P-Value
Tei vs Tie - Standard	4.07730530936212e-18
Tei vs Tie - Piped	4.29577057320019e-16
Standard vs Piped - Tei	5.3075517052254e-11
Standard vs Piped - Tie	3.95591160889952e-18

Table 34: n80w200.004 - Results of Wilcoxon paired signed rank test

n80w200.005

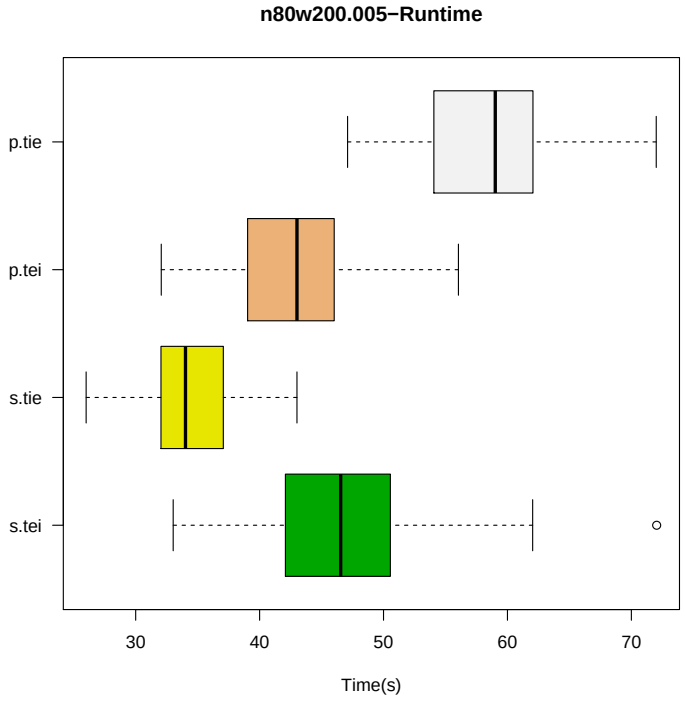

Figure 46: n80w200.005 - Runtime boxplots for the different variable neighborhood descent algorithms

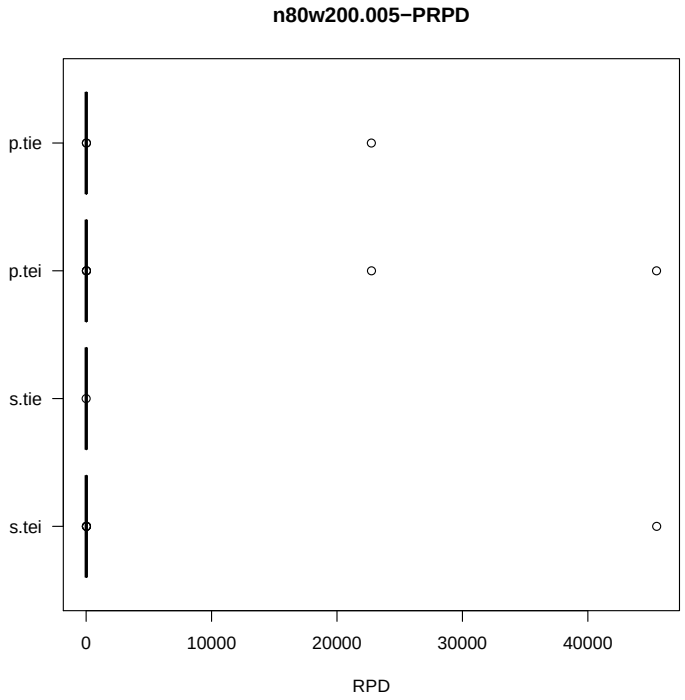

Figure 47: n80w200.001 - PRPD boxplots for the different variable neighborhood descent algorithms

Test	**P-Value**
Tei vs Tie - Standard	1.39380002081336e-17
Tei vs Tie - Piped	4.07730530936212e-18
Standard vs Piped - Tei	3.72316935219101e-06
Standard vs Piped - Tie	3.95591160889952e-18

Table 35: n80w200.001 - Results of Wilcoxon paired signed rank test

Statistics

Standard-Transpose-Exchange-Insert

Instance	**% Infeasible**	**PR$\bar{D}$P**	**Run$\bar{t}$ime**
n80w20.001	0.71	14772.04644164	50.611339
n80w20.002	0.88	12888.542	50.727053
n80w20.003	0.92	19936.872	50.820348
n80w20.004	0.62	17234.94260984	50.049484
n80w20.005	0.94	12564.0560428	50.269182
n80w200.001	0.28	11212.97389136	49.151249
n80w200.002	0.03	629.5853274	51.433949
n80w200.003	0.07	1511.56628539	49.082085
n80w200.004	0.16	4193.4817209	49.662512
n80w200.005	0.01	466.6729061	46.701953

Table 36: Statistics summary for variable neighborhood descent algorithm with Transpose-Exchange-Insert neighborhood chain and Standard VND type

Standard-Transpose-Insert-Exchange

Instance	% Infeasible	$\bar{PRDP}$	$\bar{Runtime}$
n80w20.001	0.54	10874.77632472	15.268454
n80w20.002	0.62	8411.724	15.386641
n80w20.003	0.44	7645.295	15.638153
n80w20.004	0.39	7153.68881324	15.980347
n80w20.005	0.25	3475.2731712	15.55767
n80w200.001	0.16	4898.3227617	33.424555
n80w200.002	0	11.0430351	32.198479
n80w200.003	0.05	1082.1460308	34.345522
n80w200.004	0.28	7804.19186258	32.583152
n80w200.005	0	10.20227353	34.501294

Table 37: Statistics summary for variable neighborhood descent algorithm with Transpose-Insert-Exchange neighborhood chain and Standard VND type

Piped-Transpose-Exchange-Insert

Instance	% Infeasible	$\bar{PRDP}$	$\bar{Runtime}$
n80w20.001	0.59	12336.84228578	35.694416
n80w20.002	0.94	15603.0142035	36.212393
n80w20.003	0.83	19338.924	34.821217
n80w20.004	0.55	13170.33921962	36.438959
n80w20.005	0.45	6683.336214	36.202891
n80w200.001	0.19	5104.3621015	40.772642
n80w200.002	0.01	218.8179842	44.241593
n80w200.003	0.06	2584.90674231	44.725066
n80w200.004	0.17	3430.64506042	43.760992
n80w200.005	0.02	693.0136326	42.646023

Table 38: Statistics summary for variable neighborhood descent algorithm with Transpose-Exchange-Insert neighborhood chain and Piped VND type

Piped-Transpose-Insert-Exchange

Instance	% Infeasible	$\bar{PRDP}$	$\bar{Runtime}$
n80w20.001	0.68	16393.81210394	24.788225
n80w20.002	0.81	11667.654	25.902581
n80w20.003	0.84	20537.669	26.442309
n80w20.004	0.46	8779.79314651	26.424231
n80w20.005	0.24	3876.3661498	26.511156
n80w200.001	0.21	5917.0312803	52.302366
n80w200.002	0.01	626.0983757	56.238843
n80w200.003	0.04	867.92563269	58.498874
n80w200.004	0.28	6281.58944822	55.867038
n80w200.005	0.01	236.8657243	58.331595

Table 39: Statistics summary for variable neighborhood descent algorithm with Transpose-Insert-Exchange neighborhood chain and Piped VND type

Results discussion

By looking at tables 36, 37, 38, 39 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 26, 27, 28, 29, 30, 31, 32, 33, 34, 35 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).