Heuristic Optimization Implementation Exercise 1

Alberto Parravicini

1 How to compile and run the code

1.1 *Prerequisites*

- **Cmake**, **make**, a **C++11** compiler.
- **Armadillo**, a linear algebra library. How to install it:
  ```
  sudo apt-get install liblapack-dev
  sudo apt-get install libblas-dev
  sudo apt-get install libboost-dev

  sudo apt-get install libarmadillo-dev
  ```

1.2 *Compilation*

- Move to the *build* folder
  ```
  cd build
  ```
- Run *cmake*
  ```
  cmake ../src
  ```
- Run *make*
  ```
  make
  ```

The executable will be located in the *build* folder, and is called **flowshop**.

1.3 *How to run the program*

The generic syntax to run the program is

```
./flowshop
  --filename ../instances/instance_name
  --algorithm [ii|vnd]
  --random_seed 42
  --neighbourhood_function [t|e|i]
  --initial_state_function [random|rz]
  --use_best_improvement [0|1]
  --neigh_vector [tei|tie]
```

- `--filename`, `-f`: path to the instance file to be used.
- `--algorithm`, `-a`: algorithm to be used, either **ii**, *Iterative improvement* (default), or **vnd**, *Variable neighbour descent*.
- `--random_seed`, `-r`: integer number, used as seed for random number generation by the program. If omitted, the seed is randomized.
- `--neighbourhood_function`, `-n`: how the neighbours of a given candidate solution are computed, either **t**, *Transpose* (default), **e**, *Exchange*, or **i**, *Insert*.
- `--initial_state_function`, `-i`: how the initial candidate solution is computed, either **random**, *randomly*, or **rz**, using the *Simplified RZ heuristic* (default).
- `--use_best_improvement`, `-b`: either **0** or **1** (default), set if the *iterative improvement* algorithm should use *First improvement* or *Best improvement*. Ignored if the algorithm is set to **vnd**.
- `--neigh_vector`, `-v`: set the sequence of neighbourhood functions to be used by the **vnd** algorithm, either **tei**, *Transpose, Exchange, Insert* (default) or **tie**, *Transpose, Insert, Exchange*. Ignored if the algorithm is set to **ii**.

Example:

```
./flowshop -f ../instances/50_20_10 -a ii --use_best_improvement=1 -i rz
```

The algorithm will solve the instance **50_20_10** using *Best improvement iterative search*, with a random seed, *Transpose* neighbour function, and *RZ heuristic*.

2 Introduction to the PFSP

In the *Permutation Flow Shop Scheduling* (**PFSP**) we are given:

- A set of n jobs $J_1, \ldots, J_n$.
- A set of m machines $M_1 \ldots, M_m$.
- Each job J_i is composed of m different steps, $o_{i1}, \ldots, o_{im}$, and each step must be executed on a different machine.
- Each step o_{ij} has a processing time p_{ij} associated to it.
- All jobs pass through the machines in the same order.
- A machine can process at most 1 job at a time.
- Each job has a weight w_i associated to it, which represents its importance.

Assume that the jobs run in the order $j_1, \ldots, j_n$, and go through the machines in order $M_1, \ldots, M_m$. The completion time of the ith operation of job j_k will be given by

$$C_{i,j_k} = \begin{cases} \sum_{l=i}^{i} p_{l,j_1} & if\ j_k = j_1 \text{ (first job)} \\ \sum_{l=1}^{k} p_{1,j_l} & if\ i = 1 \text{ (first machine)} \\ max(C_{i-1,j_k}, C_{i,j_{k-1}}) + p_{i,j_k} & \text{otherwise} \end{cases}$$

In short, the completion time of the first job will be given by the sum of the processing times of each of its steps (as j_1 will always find all the machines available).
Also, a new job can be processed by the first machine as soon as this is available, hence the second condition.

Our goal is to find the optimal scheduling for the jobs, such that the **weighted completion time** is minimized.

$$min\ WCT = \sum_{i=1}^{n} w_i \cdot C_{m,j_i}$$

3 Implementation

This section will focus on the **C++** implementation of algorithms to solve **PFSP**. For each class/file, it is provided a brief description.

• `flowshop.cpp`:
main access point to the implementation. The input arguments are handled using **Cxxopts** (https://github.com/jarro2783/cxxopts), which provides GNU-style syntax for the input arguments.
It will also measure the execution time, and write the output to a file.

The central idea of the optimization algorithms implementation is to create a structure which allows the algorithms to operate transparently to the problem they are solving.
In order to do so, the implementation of the problem instance and of the candidate solutions are wrapped into appropriate classes, which expose to the optimization algorithm a set of functions what work transparently to the underlying implementation of the algorithm.

Even though this architecture can look rather complex, and potentially slow down the optimization process, it is also very versatile and can be extended to any sort of optimization or search problem that can be solved by *local search* algorithms.

• `pfsp_state.h, pfsp_state.cpp`:
The *candidate solutions* (also referred to as *states*) are wrapped into a class which allows the optimization algorithms to process the candidates solutions independently from their actual implementation.
The class can also store the value of the state, given by some evaluation function. This could be useful, for example, if we had to store states in a priority queue sorted according to the state value.

• `pfps_problem.h, pfsp_problem.cpp`:
This class works as a wrapper to the actual problem instance.
A *problem* is modelled as an entity having an initial state, an evaluation function which can be applied to the states, and a function to compute the neighbourhood of a given state.
All these functions are actually function pointers, and their implementations can be set at runtime when the problem is instantiated, or even after, if one wants to dynamically update the evaluation function that is being used (as in *dynamic search*), or update how neighbours are generated (as in *variable neighbourhood descent*).

• `support_function.h, support_function.cpp`:
this file provides a series of functions that are used by the optimization algorithms. These functions are passed as input to the optimization algorithms by using

function pointers.
This provide higher flexibility, and makes the algorithms transparent to how candidates solutions are evaluated or generated.

The evaluation function is decomposed in 2 parts: one is exposed to the optimization algorithm, and has a signature which is independent to the problem which is being solved; the other takes as input some parameters specific to an instance; this was done so that the **RZ heuristic** can evaluate partial solutions by re-using the existing code.

Armadillo provides easy-to-use data-structures for matrices and vectors, with a syntax close to *MATLAB*.
This allows for easy manipulations of vectors, and very efficient vectorized operations.

The 3 functions to generate the neighbours of a given candidate solution will return a vector of candidate solutions, which can be inspected by the optimization algorithm.
The functions that compute the initial candidate solution are also provided here. They take a problem instance as input, and output a candidate solution.
It should be noted that all the neighbours of a given state are generated, and they are evaluated only later.
Evaluating them straight away would improve the performances of *Iterative Improvement*: however it will be shown later that the speed-up are very small, if the algorithm is combined with the *RZ heuristic* (which is what would be done in a real-case usage).

• `ii_engine.h, ii_engine.cpp`:
this class holds the implementation of the **iterative improvement** algorithm.
The constructor allows to choose between using *first* or *best* improvement, and to specify a *problem* to be solved, or optimized. The result of the search will be stored inside the class, and accessible to the outside.
It is important to note that the algorithm doesn't directly know how the problem instances or the candidate solutions are implemented, and as such the implementation is very flexible and easily extendible to other problems.

• `vnd_engine.h, vnd_engine.cpp`: this class holds the implementation of the **variable neighbourhood descent** algorithm (*VND*).
The structure of the class is similar to the one of iterative improvement.
In the case of *VND*, we use multiple neighbourhood functions, passed as a vector of function pointers to the class constructor.
As such, it is easily possible to use any combination of neighbourhood functions; moreover, if a single function is given, the algorithm becomes equivalent to *iterative search*. However, it was preferred to keep separate the 2 implementations, for the sake of clarity.

4 Inferential statistical analysis

4.1 *Introduction*

The implemented algorithms offer a wide choice of parameters to be set, in terms of how the initial state is computed, how new candidate solutions are generated, and so on.
As such, it is important to evaluate which combinations offer the best performances, both in terms of **results** (the *weighted completion time* defined above) and **execution time**.

The algorithms were tested on 60 different problem instances: 30 instances with 50 jobs and 30 instances with 100 jobs. All the instances had 20 machines.

2 types of tests were performed:

- Test the *iterative improvement* algorithm, by trying all different combinations of initial state generation (*random* and *RZ*), neighbourhood generation (*transpose*, *exchange*, *insert*) and first/best improvement, for a total of 12 different combinations of parameters.
- Test the *variable neighbourhood descent* algorithm, by trying different combinations of neighbourhood generator functions. The algorithm was tested with *Transpose, Exchange, Insert* and with *Transpose, Insert, Exchange*. In both cases, the algorithm was set to use *first improvement* and the *RZ* heuristic for the starting state.

To have comparable results across the tests on a single problem instance, the **seed** used by the *random number generator* was kept fixed across a single instance.

The tests were performed on the following machine:

- Computer: Microsoft Surface Pro 4
- CPU: Intel Core i5-6300U at 2.4 GHz (clocked at 2.95 Ghz)
- RAM: 4 GB at 1867Mhz

4.2 *Exploratory analysis*

Before starting any statistical test, it is a good idea to visualize the data, so to see if it's immediately possible to notice any interesting structure in the results.

First of all, it is required to separate the data relative to the instances with 50 jobs from the ones with 100 jobs, as the values from the 2 sets are not comparable with each other.

Then, it is possible to plot **boxplots** that display summary statistics of the **execution times** and of the **result values**. Relatively to the optimization results, it is also visualized the boxplot of the **ideal optimization results**, the values that would be achieved by an exact solver.

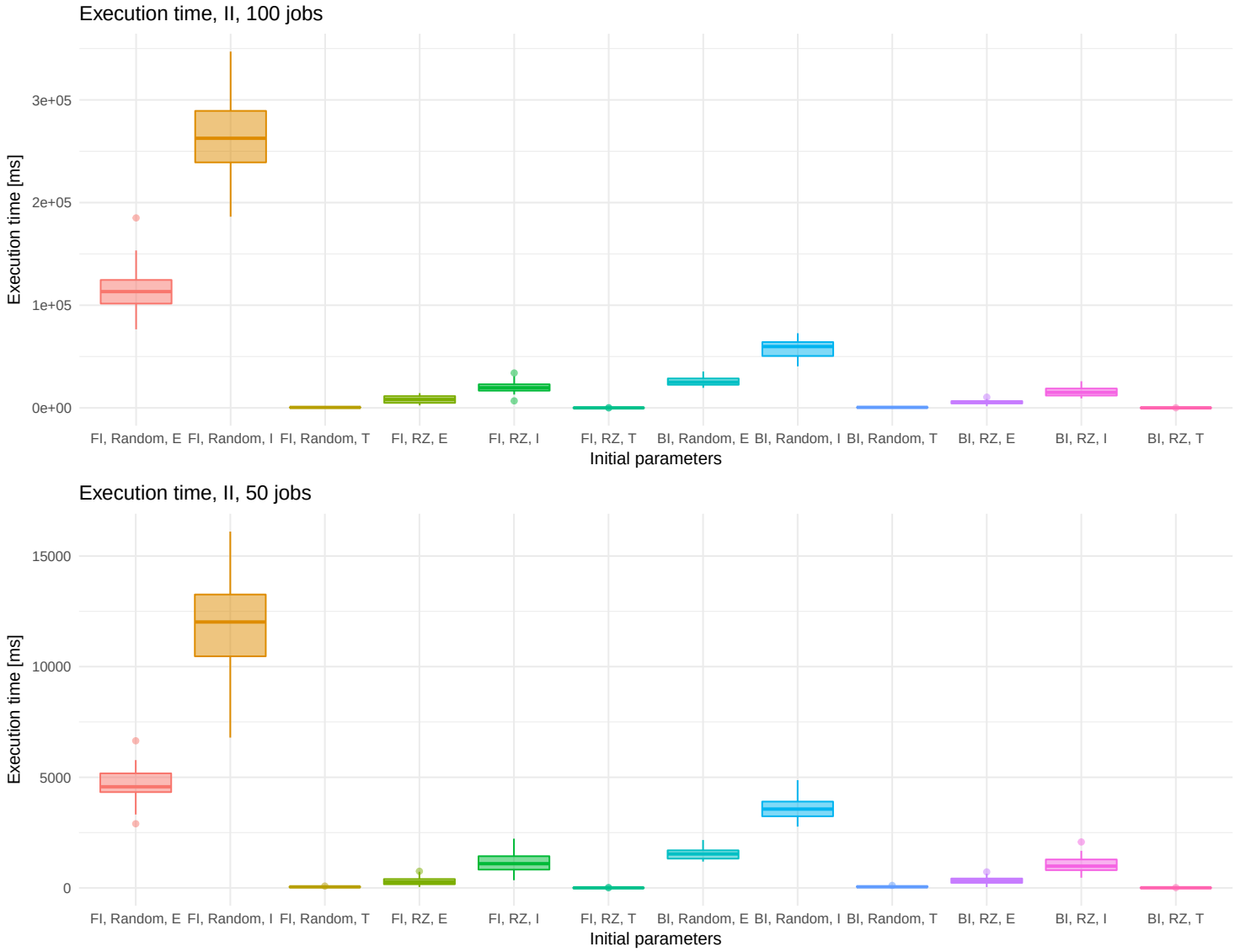

Figure 1: *Summary statistics of the execution time for* ***iterative improvement****, with different parameters.*

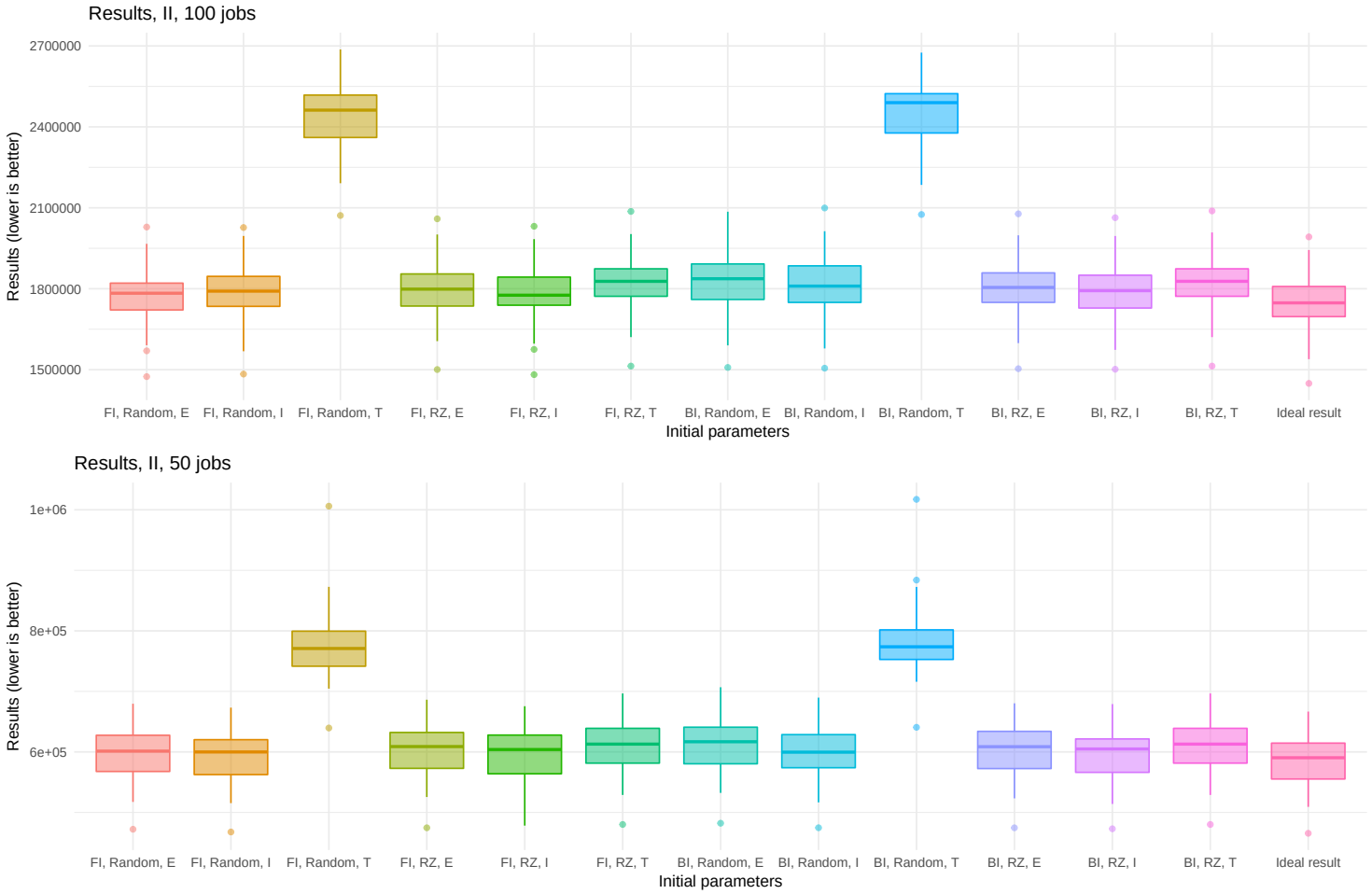

Figure 2: *Summary statistics of the optimization results for* ***iterative improvement****, with different parameters.*

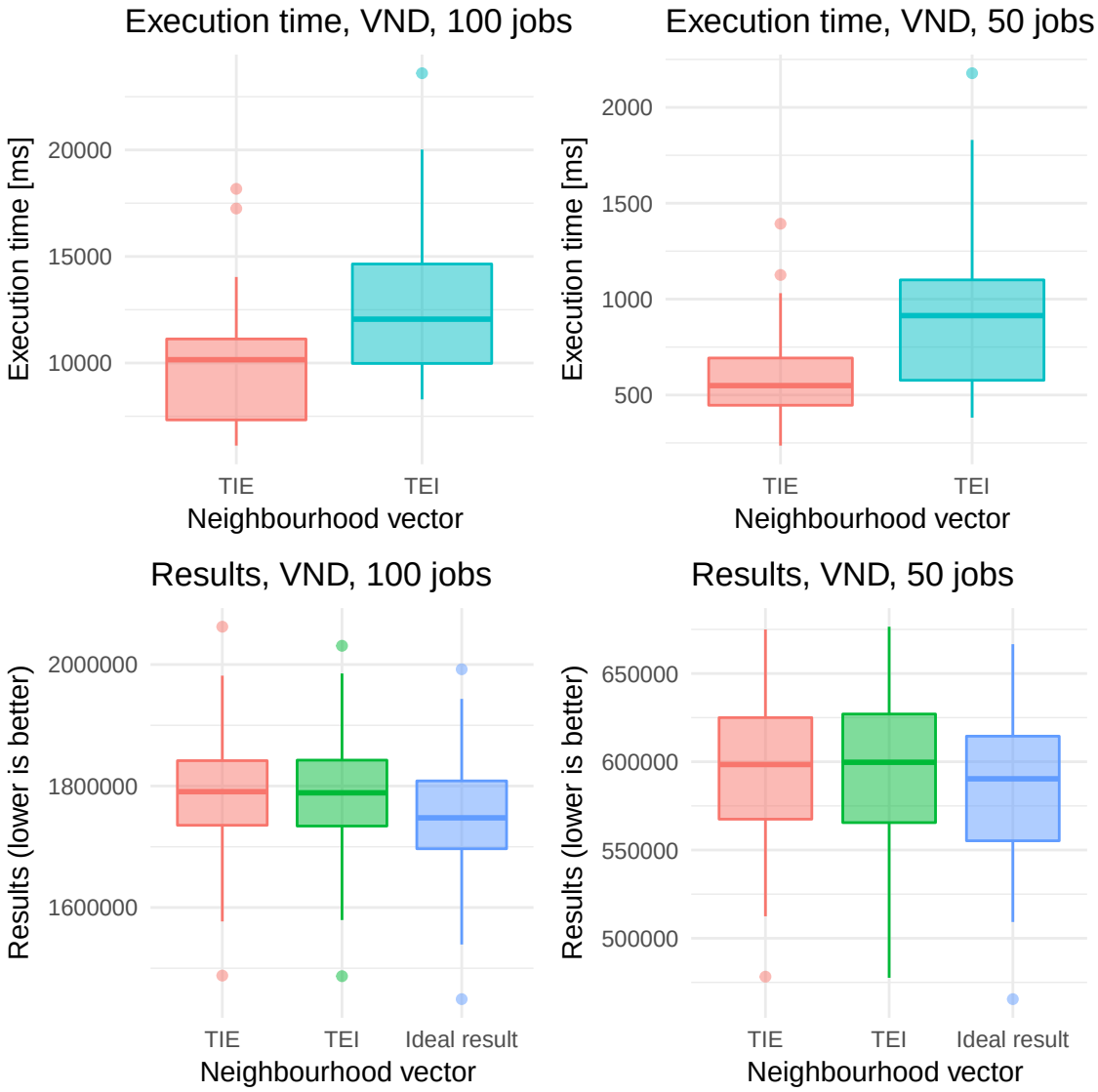

Figure 3: *Summary statistics of execution time and optimization results for* ***variable neighbourhood descent****, with different parameters.*

References