
Utrecht University

Department of Information and Computing Sciences


Master Thesis Computing Science

---

# Incorporating Domain Knowledge in Permutation Gene-pool Optimal Mixing Evolutionary Algorithms

---


*Author:*
Gerben Aalvanger, BSc (3987051)

*Supervisor:*
dr. ir. D. Thierens


July 14, 2017

# Abstract

The Gene-pool Optimal Mixing Evolutionary Algorithm for permutation problems (pGOMEA), is a model based evolutionary algorithm (MBEA) capable of learning problem structure by itself. pGOMEA does so by creating a linkage tree from dependencies found in the population. This problem structure is subsequently used to steer the recombination in pGOMEA. Though pGOMEA models the problem structure, problem specific knowledge can still be added to pGOMEA in order to improve its quality. In this thesis report, different forms of incorporating domain knowledge are applied for the permutation flowshop scheduling problem (PFSP), in order to get insight in their ability to improve pGOMEA. In the first part of this thesis, a literature study discusses pGOMEA, the PFSP and its heuristics and the various ways of incorporating domain knowledge in evolutionary algorithms in general. The second part describes the experiments that are performed to test the effectiveness of using domain knowledge in pGOMEA. The experiments show how pGOMEA cannot be improved using simple improvement heuristics like the swap and insertion heuristic. More informed strategies using domain-specific improvement heuristics however are able to boost pGOMEAs performance significantly. Unfortunately, the improvement heuristics do not further benefit from knowledge in the linkage tree. Neither does substructural search on its own achieve a significant improvement over standard improvement heuristics. Other experiments show how pGOMEA can be seeded in two ways. Firstly, good solutions can be added to the initial population, leading to a boost in performance. Secondly, the linkage tree can be build using seeded dependency values from domain knowledge, which can improve the quality of the linkage tree in early generations. As a final experiment, we have tested pGOMEA with domain knowledge (hybridized and using population seeding) against state-of-the-art algorithms solving the PFSP. The experiments show that pGOMEA is able to give good results and that pGOMEA can outperform a state-of-the-art algorithm for the PFSP with the total flowtime criterion. The effect of structure in problems does not give pGOMEA a bigger advantage over other algorithms, though we empirically show that pGOMEAs dependency functions perform better than random dependency functions.
Within the scope of this thesis, we propose a better way of generating structured PFSP instances.

# Contents

# Chapter 1

# Introduction

## 1.1 Research goal

In order to find near-optimal solutions to complex problems, Evolutionary Algorithms (EAs) combine *individuals*, representing problem solutions, in such a way that their offspring will have a combination of the good features of its parent solutions. Due to competition between individuals, only the best individuals reproduce and over time the individuals in the population get fitter. In the end, the population is supposed to converge to a good (near-optimal) solution.

Combining solutions using a random approach, can result in the problem that the algorithm won't find a good solution in reasonable time. Therefore, exploiting structure is an important area of research in EAs. Structure can be exploited by creating *combination* or *mutation* operators that fit the problem well. However, one can also use a Black-Box Optimization (BBO) approach, where little is know and assumed about the domain. This optimization approach is able to create problem-independent algorithms that are more general applicable. A BBO approach is used in so called Model Based Evolutionary Algorithms (MBEAs). MBEAs build a model for each generation, which is then used to steer the process of combining solutions into new, better solutions.

A lot of important real-world, hard optimization problems are permutation-based (e.g. vehicle routing and scheduling). However, little MBEAs are designed to solve these problems. Recently, the Gene-pool Optimal Mixing Evolutionary Algorithm (GOMEA) MBEA has been adapted to work on permutation spaces [39]. Without much knowledge of the problem, permutation GOMEA (pGOMEA) already performs surprisingly well on PFSP instances as given in the well known Taillard-benchmark. GOMEA has found numerous best known solutions for these extensively used benchmark instances. Also, pGOMEA does not often perform significantly worse than GM-EDA, a recently proposed permutation-based Estimation of Distribution Algorithm (EDA) learning problem structure.

As pGOMEA is able to perform well without having any explicit domain knowledge in it, the question arises whether pGOMEA can be further improved by adding domain knowledge to it. For instance, EAs are known to be good at searching globally through a complex search space, but they perform worse with respect to local optima. Therefore EAs have been hybridized using local searchers. Also, pGOMEA uses its linkage tree in order to model domain knowledge, which allows one to add domain knowledge in the process of building this tree.

In order to investigate the effectiveness of the various ways to incorporate domain knowledge in pGOMEA, we apply different experiments. Most of these experiments investigate a combination of heuristics for the PFSP with pGOMEA. Therefore, this research focuses on the following question:

*How can domain-knowledge of permutation problems be exploited (using heuristics) in the permutation-based Gene-pool Optimal Mixing Evolutionary Algorithm?*

## 1.2 Research approach

In order to answer the research question, a literature study has first been performed. This study first discusses GOMEA and its permutation variant pGOMEA in detail in Chapter 2. Then, the Permutation Flowshop Scheduling Problem (PFSP) is introduced in Chapter 3. This chapter also discusses domain knowledge of the PFSP by explaining constructive heuristics, improvement heuristics and meta-heuristic solvers. Finally, Chapter 4 identifies the use of domain knowledge in EAs. Here, special attention is given to local search and MBEAs. The information from literature will be used in the second part of this thesis, where pGOMEA is tried to improve using domain knowledge.

In the second part of this thesis, various experiments are conducted to identify the behavior of pGOMEA when domain knowledge is incorporated. Here, we try to answer our questions as much as possible in the context of black-box optimization. Using this approach, we can make more general statements about the effectivity of incorporating domain knowledge in pGOMEA. Most of our experiments will compare algorithms based on fitness evaluations, allowing us to make statements that do not depend too much on the domain we are working in (grey-box optimization). Where this is not possible, we experiment using computation time thus going to a white-box optimization area. First, Chapter 5 discusses the local optimality of pGOMEA solutions with respect to improvement heuristics. Experiments are conducted to get insight in the improvability of pGOMEA with local search strategies. Chapter 6 evaluates various ways of seeding pGOMEA with solutions generated by constructive heuristics. Chapter 7 experiments with hybridizing pGOMEA with LS. Both domain-independent and domain-specific local search are tried to use in pGOMEA. After these rather well-known forms of adding domain knowledge, dependency-seeding is introduced in Chapter 8. Dependency seeding is a new concept, where the model of pGOMEA is build using domain knowledge as well as population knowledge. As a final experiment, we shortly investigate the applicability of pGOMEA models to substructural neighborhood searchers in Chapter 9. The best form of pGOMEA with domain knowledge, is then compared with state-of-the-art algorithms solving the PFSP in Chapter 10. In this chapter we will compare pGOMEA both on random as well as on structured instances. A technical description of a new structured benchmark set is given in Appendix A. We will conclude this thesis with a summary of the results and some practical recommendations in 11.

# Chapter 2

# Gene-pool Optimal Mixing Evolutionary Algorithm

This chapter introduces (p)GOMEA, as we will examine different possibilities of exploiting domain-knowledge in pGOMEA in Chapters 5 to 9. (p)GOMEA is an MBEA that uses sets of dependent variables as crossover masks. GOMEA has its origins in the Linkage Tree Genetic Algorithm (LTGA) as proposed by Thierens [38]. In later research the notion of Optimal Mixing (OM) is introduced by Thierens and Bosman [39]. This leads to the introduction of GOMEA and the closely related Recombinative Optimal Mixing Evolutionary Algorithm (ROMEA). In this chapter, the need for MBEAs like GOMEA is first explained in Section 2.1. Then GOMEA and its Family-Of-Subsets (FOS) models are discussed in Sections 2.2 and 2.3. Thereafter a review of GOMEA with respect to BBO is given in Section 2.4. Since GOMEA traditionally works on problems with a Cartesian search space, a permutation-based GOMEA (pGOMEA) is recently introduced by Bosman and Thierens [2]. The main differences between pGOMEA and GOMEA are explained in Sections 2.5 and 2.6. Finally Section 2.7 treats the effectiveness of pGOMEA on the PFSP as introduced in Chapter 3.

## 2.1 Background

EAs contain a wide range of population-based optimization algorithms inspired by nature. A well known type of population-based optimization algorithms is the Genetic Algorithm (GA). GAs aim to find a good or even optimal solution with respect to a given (black box) fitness function. The program flow of a GA is based on a repetition of *selection*, *crossover* and optionally *mutation*. In the selection phase, the best individuals of the population are selected according to a given selection protocol (e.g. tournament or proportionate selection). In the crossover or recombination phase, the selected individuals are (pairwise) combined using a crossover operator forming a child population. These new solutions possibly undergo a mutation, which introduces diversity in the offspring. After each iteration, the population is (partially) replaced by its offspring.

The idea behind a GA is that the recombination of good solutions forms other good solutions. Due to the selection pressure in the selection phase, the population will converge to a population with high-fitness individuals. A big challenge in designing GAs is the choice of a good recombination operator and the determination of the optimal population size. The recombination operator should combine the building-blocks of two solutions and merge them together. With a good recombination operator, the good properties are preserved for the next generation. A bad recombination operator can lead to the creation of 'random' children, since it can break important building blocks. In that case the GA will randomly search the problem space and it will not exploit problem structure. When the population size in a GA is chosen too small, the population will converge before exploring the search space enough. This leads to a final population with mediocre fitness and little diversity. When a population is too large, too much computing resources are spend for finding a good solution.

Unfortunately, the population size and the optimal choice of a recombination operator differs

per problem and even per instance of a problem. Theoretical analysis shows that in worst-case, exponential population sizes are needed to solve polynomial-time solvable problems [37]. Therefore, exploiting problem structure in GAs is very important. Problem structure can be exploited by using foreknowledge of the domain (in the form of LS or specific operators) or by using model-learning. Especially in BBO, where no domain knowledge is given, model-learning can be hugely beneficial. For this purpose MBEAs have been designed. The goal of MBEAs is to exploit self-learned dependencies between variables. One of the main categories in MBEAs consist of the EDAs. EDAs differ from GAs in that they do not use crossover and mutation. Instead, EDAs learn a (probabilistic) model of the population and sample the offspring from this model.

GOMEA is like EDAs an MBEA; instead of using a probabilistic model, GOMEA uses linkage learning in order to learn and exploit domain knowledge. Linkage learning learns which variables form important building-blocks of the solution and should therefore not be disrupted by a crossover (or mutation) operator. In GOMEA, linkage learning identifies subsets of variables that together form building-blocks. These sets are then used as crossover masks in GOMEAs optimal mixing phase.

## 2.2 GOMEA

GOMEA originates from the LTGA as introduced by Thierens [38]. Currently, the linkage tree is only one of the possible FOS models that can be used in GOMEA, these models will be further explained in Section 2.3. In short, the FOS contains sets of variables which can be used during crossover. GOMEA is closely related to ROMEA, but since GOMEA outperforms ROMEA (with a factor $\geq 1.7$), we will only discuss GOMEA here.

GOMEA can be seen as a combination of an EDA and a GA. On the one hand it uses a model that is learned, but on the other hand it uses crossover in a similar way as GAs do. Like GAs, GOMEA starts with a (random) initial population of size $n$. Then, the iterative process is started, which has two successive steps: model-building and Gene-pool Optimal Mixing (GOM). In the first step, a FOS-model is build (Section 2.3). The FOS contains sets of variables (e.g. $\{x_1, x_4\}$) that are somehow linked, they share information. After building this FOS-model, each solution in the population is greedily improved using the sets in the FOS-model. For each solution (*receiver*), every FOS-set is tried as a crossover mask in a random order. For every mask/FOS-set, a random solution (*donor*) is then selected from the population. The masked variables in the receiver will be substituted with the masked variables of the donor. An example of such a donation would be $Donate(11111, \{x_1, x_4\}, 00000) = 01101$. If a donation leads to a decrease in fitness of the receiver, the donation is rejected. The process of donating using a FOS-set is called *Optimal Mixing* (OM). Since for each FOS-set a random donor is chosen, this is called Gene-pool optimal mixing. Model-building and Optimal mixing are repeated until a given termination criterion is met (e.g. running time or population convergence). A pseudocode overview of GOMEA for maximizing fitness function $f$ is given in Algorithm 1.

```
Result: A good/optimal solution with respect to fitness function f
Pop ← rand_Pop(n) ;
while ¬termination_criterion do
    FOS ← build_FOS(Pop);
    foreach receiver ∈ Pop do
        receiver* ← receiver;
        foreach set ∈ FOS do
            donor ← Random(Pop);
            child ← Donate(receiver*, set, donor);
            if f(child) ≥ f(receiver*) then
                receiver* ← child
return best solution from Pop
```

**Algorithm 1:** GOMEA outline

The GOMEA algorithm as outlined above is only a basic implementation of GOMEA. In reality it is worthwhile to consider two extensions called Forced Improvement (FI) and Population sizing.

### 2.2.1 Forced Improvement

FI is a second phase in the GOM step of GOMEA, which is only entered when the population does not improve fast enough. FI is designed in order to force improvements on the solutions in the population, and thereby forcing convergence. The FI phase does exactly what the first phase does, except it does not use a random donor, but the overall best solution as a donor. It also only accepts strict improvements after which the phase is stopped, this ensures that diversity is not lost in the population. (Note the difference between a change and an improvement of a solution). The FI phase can be entered based on two criteria: Firstly, solutions that have not changed in the first GOM phase will enter the FI phase. Secondly, if the overall best solution has not improved for a number of generations (called the *no-improvement-stretch* (NIS)), the FI phase is entered. Experiments show that $NIS = \lfloor 1 + \log_{10}(n) \rfloor$ is a good value for the NIS. Experiments show that FI establishes good convergence without reducing diversity too much [5].

### 2.2.2 Population sizing

The ability of a GA to solve a problem is very dependent on the population size $n$. A small population size converges too fast to a non-optimal solution, while a big population size gives a lot of computational overhead. Therefore, GOMEA uses a population sizing scheme which does not use a fixed population size, instead it finds the right population size during the optimization process without much overhead. In the population sizing scheme, GOMEA starts with a small population size $n_{base}$. Every 4 evaluations of this population a population with twice the size is evaluated once. This procedure recurses, meaning that a population of size $4 \cdot n_{base}$ is once evaluated when the population of size $n_{base}$ is evaluated 16 times. When a population has a higher average fitness than a smaller population, the smaller population is thrown away. It is assumed to improve too slow; for the same reason, populations without any diversity are terminated.

## 2.3 FOS-Models

For now, we have not mentioned how the FOS-Model is build in GOMEA. The way the FOS is constructed is of big importance for GOMEA. The sets in the FOS should resemble the building-blocks of the solutions, the FOS then helps the crossover operator to perform good recombination. Unfortunately, the optimal FOS-model can only be approximated from the knowledge in the current population. The optimal FOS-model will always be a trade-off between size and information. A FOS-model containing all subsets of variables leads to exponential time in the GOM step. A FOS-model without any dependencies makes a big assumption about the problem domain. In this section, we discuss three possible FOS-models, which are used in or are related to known MBEAs.

### 2.3.1 Univariate model

The most simple FOS-model is the Univariate model, which assumes that each variable is independent from the other variables. This assumption leads to a FOS with only singleton sets whose union forms the set of all variables i.e. if there are $l$ variables then $FOS = \{\{x_1\}, \{x_2\}, \cdots, \{x_l\}\}$.

### 2.3.2 Marginal product model

The Marginal Product Model (MPM) assumes that variables can be grouped together according to dependencies. The groups of variables are mutually independent. This results in a FOS with non-overlapping sets whose union forms the set of all variables. The MPM can be seen as a univariate model where multiple sets are merged together. (e.g. $FOS = \{\{x_1, x_3\}, \{x_2, x_4, x_5\} \cdots\}$)

### 2.3.3 Linkage tree model

The linkage tree model is often incorporated in GOMEA, forming LT-GOMEA or LTGA. A linkage tree is a binary tree with the set of all problem variables as root and all singleton sets of the problem variables as leaves. The children of a set are non-overlapping sets whose union forms the set in their parent. While the univariate structure has $l$ sets and the MPM structure has less than $l$ sets, a linkage tree has $2l - 1$ sets. A linkage tree is learned/built bottom up, starting from the univariate structure. Nodes/sets are merged into their parents by selecting the two sets with the

highest Mutual Information (MI). For sets with more than one item, this is approximated using the Unweighted Pairwise Group Method with Arithmetic mean (UPGMA), given by:

$$MI_{UPGMA}(S_i, S_j) = \frac{1}{|S_i| \cdot |S_j|} \sum_{x \in S_i} \sum_{y \in S_j} MI(x, y). \tag{2.1}$$

This notion of UPGMA is an approximation that theoretically reduces the quality of the linkage tree, but improves the time in which it can be generated. The computational complexity of generating a linkage tree is with $\mathcal{O}(l^2 n)$ relatively low, while in practice the quality of the tree does not significantly decrease. As the linkage tree is the most powerful FOS of the three, we will use the linkage tree in our experiments.

## 2.4 GOMEA from a Black-Box perspective

Since GOMEA builds its own model, based on domain knowledge in the population, it is a good candidate to be used in BBO. GOMEA does not need a problem-specific crossover operator, instead it uses crossover masks that are based on the FOS structure. Bosman and Thierens [4] review the quality of GOMEA with respect to a BBO perspective. GOMEA is compared with EDAs and combinations of them with a simple (bit-flip) LS in a model-based neighborhood. First, the algorithms are tested on a GA-hard deceptive trap function defined by

$$f_{Trap5} = \sum_{i=0}^{(l/5)-1} f^{sub}_{Trap5}\Big( \sum_{j=5i}^{5(i+1)-1} x_j \Big), \tag{2.2}$$

where

$$f^{sub}_{Trap5}(u) = \left\{ \begin{array}{ll} 1, & \text{if } u = 5 \\ \frac{4-u}{5}, & \text{otherwise} \end{array} \right\}. \tag{2.3}$$

This function is well suited for the MPM structure as also used in EDAs. Secondly, the algorithms are tested on the NK-landscape which is a hard problem for GAs. The formal definition of an NK fitness function is given by

$$f_{NK}(x) = \sum_{0}^{l-5} f^{sub}_{NK}(x_{(i,i+1 \cdots i+4)}). \tag{2.4}$$

Here, the sub-function is a random predetermined value in $[0, 1]$. Experiments show that LT-GOMEA performs very well on these problems. In terms of function evaluations, LT-GOMEA outperforms the other model-based approaches. However, LT-GOMEA could not be improved using a substructural neighborhood (as further explained in Chapter 4), which might indicate that LT-GOMEA already uses a LS strategy. The authors conclude that GOMEA is a very effective mix between an EA and LS in a BBO context. From a non-BBO perspective, LS operators might still be very useful for GOMEA, since fitness functions might be partially evaluated. We will further examine the combination of LS and GOMEA in Chapter 7. There we use the notions in Chapter 4, where we use literature to study the combination of LS and GAs.

## 2.5 GOMEA for permutation problems: pGOMEA

GOMEA is like most MBEAs designed to solve problems in Cartesian space, meaning that every variable can always take any value from a given domain. A lot of hard, real-world problems however are in permutation space, meaning that each variable gets its unique value assigned. Examples of permutation problems are the Traveling Salesman Problem (TSP) where the optimal city order should be found, and the PFSP where the optimal processing order of jobs has to be found. Normal crossover operators cannot be used in permutation space, since this can lead to infeasible solutions. To overcome this problem, advanced operators or different solution encodings have to be used. In order to extend (LT-)GOMEA for permutation problems, the latter approach has been deployed to construct pGOMEA [2].

### 2.5.1 Solution ecoding

For encoding a permutation of the variables $(x_1, x_2 \cdots x_l)$, pGOMEA uses a *random keys* encoding. A random keys encoding assigns a real value $r_{x_i}$ within a certain interval (here: $[0, 1]$) to each variable $x_i$ (e.g. $(0.2, 0.5, 0.3, 0.9, 0.1)$ for a 5-variable problem). The position of variable $x_i$ in the permutation is equal to the position of $r_{x_i}$ when sorting all random keys in ascending order. In the example above, $x_0 = 1$, since it is the second lowest random key. Switching between the random keys encoding and the permutation encoding takes $\mathcal{O}(l \log l)$ time using a sort operation. A simple crossover between two random keys encodings always results in a valid random keys encoding, so the GOM operator can easily be used. Note that different random keys encodings can encode the same permutation e.g. $(0.1, 0.2, 0.3) = (0.2, 0.3, 0.4)$.

### 2.5.2 Building the Linkage Tree

Though the GOM operator can work easily with the random keys encoding, the notion of dependency between (sets of) variables should be redefined. In permutation problems, two factors play a role in the notion of dependency.
The first factor contributing to the dependency of variables is *relative-ordering information*. The reason for incorporating relative-ordering is intuitive: if one variable is always before (or after) another variable in the permutation, this indicates a strong dependency between them. If two variables are randomly before or after each other, this indicates a weak dependency. Therefore, the probability $p_{ij}$ that variable $x_i$ is before variable $x_j$ in a permutation is used (i.e. $p_{ij} = Pr(r_{x_i} < r_{x_j})$). This probability can be incorporated in the notion of entropy:

$$H_{ij} = -[p_{ij} \cdot \log_2(p_{ij}) + (1 - p_{ij}) \cdot \log_2(1 - p_i j)]. \tag{2.5}$$

Since the entropy is 1 when each outcome is as probable as the others, we have to subtract the entropy from 1, resulting in the measure for relative ordering: $\delta_{rel-ord}(i, j) = 1 - H_{ij}$.

The second factor of dependency between two variables in a permutation is *proximity*. Intuitively this can be explained using a TSP example. When two cities are close to each other, they will often occur together in good solutions. They form a building block in the solutions. In order to capture this dependency, we have to look at the distance between two variables in the permutation or to the distance in random-key value between two variables. Thierens and Bosman use the average squared distance between two random keys to define a notion of distance:

$$d_{ij}^2 = \frac{1}{n} \sum_{k=0}^{n} (r_{x_i}^k - r_{x_j}^k)^2. \tag{2.6}$$

A bigger distance corresponds to a lower dependency between the two variables, so we subtract the value from one to invert its meaning: $\delta_{prox}(i, j) = 1 - d_{ij}^2$. Both dependency measures can be combined into a symmetric dependency measure in the interval $[0, 1]$ by multiplying them:

$$\delta(j, i) = \delta(i, j) = \delta_{rel-ord}(i, j) \cdot \delta_{prox}(i, j). \tag{2.7}$$

Like GOMEA, pGOMEA uses the UPGMA measure calculation to calculate the dependency between two sets of variables.

### 2.5.3 Operators

When the linkage tree is incorporated in pGOMEA, LT-pGOMEA or pLTGA is formed. In this paper we will only examine pGOMEA with a linkage tree, therefore pGOMEA will refer to pGOMEA with a linkage tree. At the start of pGOMEA, a random population is generated with random values for the random keys. During the crossovers, diversity in the values of random keys decreases quickly. Therefore, mixing a set of variables in another solution will not always go well. Suppose variables $x_1 \cdots x_3$ and $x_4 \cdots x_6$ are dependent. The probability that mixing is performed resulting in donor (*don.*) and receiver (*rec.*) random keys $r_{x_1}^{don.}, r_{x_2}^{don.}, r_{x_3}^{don.} < r_{x_4}^{rec.}, r_{x_5}^{rec.}, r_{x_6}^{rec.}$ is very low and might decrease over time. However, $x_1 \cdots x_6$ can be a worthwhile building block when this property holds. In order to overcome this, the *random rescaling* operator is introduced. This operator scales the random keys of a variable subset from the donor to another interval before

inserting the random keys in the receiver.

For example, when the random keys $(r_{x_1}, r_{x_2}, r_{x_3}) = (0.2, 0.3, 0.4)$ are scaled to the interval $[0.1, 0.2]$, the random keys will become $(0.1, 0.15, 0.2)$ and will be substituted in the receiver. Note that when the size of the interval is equal to the size of the random keys interval, this rescaling operator does not actually rescale, it only shifts the random keys.

In pGOMEA, random rescaling is applied with a probability of 0.1. The scaling interval is randomly selected from a predetermined set of $l$ equidistant sized sub-intervals of $[0, 1]$. These settings are obtained in experiments that are conducted on the random rescaling using ICE algorithm [3].

Since random rescaling is applied less than 10% of the time, random keys diversity can still drop in the population. A second operator is introduced in order to introduce random keys diversity: *re-encoding*. Re-encoding is applied to every individual at the end of a generation. The re-encoding changes the random keys encoding of each individual without changing the corresponding permutation. Re-encoding can be easily implemented by generating a sorted vector $v$ of random values in $[0, 1]$; the $i$th value in the random keys order gets the $i$th value from $v$. Note that the re-encoding operator changes distances between random keys, which has influence on the proximity dependency.

## 2.6 pGOMEA configurations

When pGOMEA is run with the same parameter settings as GOMEA (population sizing and FI), population sizes of up to $2^{19}$ were created, indicating that smaller configurations are easily overtaken by larger populations. In order to solve this problem, the population sizing scheme and the FI phases are slightly changed.

### 2.6.1 Population sizing

In GOMEA, when a smaller population is overtaken, it is assumed to improve too slow. In pGOMEA, finding improvements takes more time, since permutation space is generally larger than Cartesian space (factorial vs. exponential). An overtaken population can still give good solutions, therefore populations are not prematurely stopped when overtaken. Populations are only stopped when they don't have any diversity.

### 2.6.2 Forced Improvement

We have seen that permutation space is bigger than Carthesian space and thus improvements need more time. This does not only affect population stopping; FI can also be executed too often. This leads to a drop in diversity, because the best solution in the population will hugely determine the contents of the population. This results in a fast convergence towards the best solution in the population. In other words, pGOMEA looses its ability to explore the search space, instead it exploits the best solution in the population too much. In order to overcome this problem, the NIS is multiplied by 10, resulting in $NIS = 10 + 10 \cdot \lfloor \log_{10}(n) \rfloor$, which gives better performance.

## 2.7 Results

pGOMEA has been tested on the PFSP with the total flowtime (TFT) criterion using the Taillard Benchmark [35]. More information about this problem and its benchmark instances is given in the next chapter.
First, the different configurations of pGOMEA have been compared, it turns out that the changed population termination criterion and the new FI threshold significantly improve the quality of the found solutions.
Then, pGOMEA with its four variants (with and without rescaling and re-encoding) is compared with a recently proposed permutation EDA: GM-EDA [7]. Both algorithms get the same computational budget in terms of fitness evaluations. In most cases, pGOMEA outperforms GM-EDA, except when considering instances with 200 jobs. For most other instances, pGOMEA has at least

one significantly better performing variant (with and without rescaling and re-encoding). When no rescaling and re-encoding is used, pGOMEA is able to give good solutions in instances with up to 50 jobs. After that, the use of either of these operators is necessary. Though both operators improve the performance of pGOMEA, rescaling seems to work better on bigger instances. Using both rescaling and re-encoding does not make pGOMEA significantly better than using only rescaling. In short, pGOMEA is a promising algorithm for solving permutation problems; even without explicit domain knowledge, pGOMEA is able to find good or even best solutions for the PFSP.

# Chapter 3

# Permutation Flowshop Scheduling Problem

The PFSP has been used in order to test the performance of pGOMEA. Since this research tries to improve on pGOMEA by incorporating domain knowledge, it is self-evident that we should first take a look at the domain of PFSP. The PFSP is a problem in the field of machine scheduling. In machine scheduling, a batch of $J$ *jobs* should be scheduled in such a way that the processing plan is optimal with respect to a given criterion. Many machine scheduling problems exist and while some are solvable in polynomial time, others are known to be NP-hard. Since the PFSP is also known to be NP-hard [14], this is an ideal optimization problem for pGOMEA. In this chapter, the PFSP is first formally defined in Section 3.1. Then benchmark problems and statistical tests for comparing PFSP algorithms are introduced in Section 3.2. The main constructive and improvement heuristics for the PFSP are discussed in Sections 3.3 and 3.4, as they are a great source of domain knowledge. This chapter concludes with an overview of state-of-the-art algorithms/metaheuristics that are used to find near-optimal solutions for the PFSP. The main goal of this chapter is to give insight in PFSP and approaches that use PFSP domain knowledge in order to 'solve' this problem. The results of this chapter will be used in different experiments on combining pGOMEA with heuristics.

## 3.1 Problem description

The PFSP is concerned with finding the optimal solution for scheduling $J$ jobs on $M$ *machines*. Each job requires $M$ operations, which should be performed sequentially, starting on machine 1 and finishing on machine $M$ (the *Flowshop* property). Operations cannot be interrupted, but a job can be delayed when its operations are not performed immediately after each other. Any solution can be seen as a permutation of jobs, since each machine has to process the jobs in the same order (the *Permutation* property). In three field notation, the PFSP is denoted by $F|prmu|\gamma$, where $\gamma$ refers to the objective function that is used for optimizing the schedule.

For each operation $(i, j)$ of job $i$ that should be operated on machine $j$, we define the processing time $p(i, j)$. We can define any schedule $\pi$ as a permutation of the jobs: $\pi = \{\pi_1, \pi_2, \cdots, \pi_J\}$. For the first machine we can simply compute the completion time $c(\pi_i, 1)$ for each job $\pi_i$ by

$$c(\pi_i, 1) = \sum_{k=0}^{i} p(\pi_k, 1). \tag{3.1}$$

We can also easily compute the completion times of all operations of the first scheduled job ($\pi$) by

$$c(\pi_1, j) = \sum_{k=0}^{j} p(\pi_1, k). \tag{3.2}$$

All other completion times depend on the completion times of the previous operation of the same job and the previous operation on the same machine:

$$c(\pi_i, j) = \max\{c(\pi_{i-1}, j), c(\pi_i, j-1)\} + p(\pi_i, j). \tag{3.3}$$

For each operation $\pi_i$ we can also define the starting time $s(\pi_i, j)$; computed by $s(\pi_i, j) = c(\pi_i, j) - p(\pi_i, j)$.

### 3.1.1 Objective functions

The PFSP has multiple criteria which can be optimized for a schedule. These criteria are an important factor when solving the problem. For some criteria, the problem can be easier solved than for others. Since we will limit ourselves to the relevant PFSP information for pGOMEA, we only discuss the two main optimization criteria for the PFSP.

$C_{max}$: The $C_{max}$ criterion is an intuitive measure for the quality of a solution. In industry, producers want to process the batch of jobs as soon as possible. The time that the last job is finished should therefore be minimized. This is called the $C_{max}$ criterion, also known as the flowtime or makespan criterion. When all completion times are known, $C_{max}$ can be extracted by $C_{max} = c(\pi_J, M)$. The $C_{max}$ criterion makes PFSP NP-hard when $M > 2$. [14]

TFT: The TFT criterion is based on the minimization of the sum of all completion times of the jobs i.e. $TFT = \sum_{i=0}^{J} c(i, M)$. The idea behind this criterion is that each job needs space in the inventory, but each job can be removed from the inventory once it is completed. TFT can also be used when multiple clients are waiting on their jobs, the TFT then minimizes the total dissatisfaction. The Average Flow Time criterion also exists and is almost equal to the TFT criterion, since it can be computed as $\frac{TFT}{J}$. When using the TFT criterion, PFSP is NP-hard when $M > 1$ [15].

Both the $C_{max}$ and the TFT value can be calculated in $\mathcal{O}(J \cdot M)$ time, by calculating all completion times using dynamic programming.

## 3.2 Comparing solutions

Since the introduction of the PFSP, a lot of heuristics, metaheuristics and other algorithms have been proposed in order to find good schedules. These algorithms have to be compared using test instances. The most used set of test instances is created by Taillard [35], he published a set of 120 test instances. The test instances are grouped in sets of 10 instances having the same size. The instance sizes are shown in Table 3.1 an correspond to sizes of real industrial problems. The processing times of the jobs are uniformly sampled from the interval $[1, 99]$. For each instance size, Taillard has chosen 10 tests that are considered to be the hardest from a larger set of generated instances[1].

Whereas the Taillard-instances are hard randomly generated problems, Watson et al. [42] remark that most real-world problems contain structure. Therefore, they propose a new benchmark that contains correlation between jobs and/or machines. Since algorithms that are superior on random instances do not necessarily perform well on structured instances, this is a useful benchmark for comparing the performance of MBEAs. For the Watson-benchmark, a lot of instances with different correlation coefficients ($\alpha$) have been created. The underlined sizes in Table 3.1 correspond to sizes for which both Taillard and Watson instances are available.

| | $J = 20$ | $J = 50$ | $J = 100$ | $J = 200$ | $J = 500$ |
|---|---|---|---|---|---|
| $M = 5$ | $20 \times 5$ | $50 \times 5$ | $100 \times 5$ | | |
| $M = 10$ | $20 \times 10$ | $50 \times 10$ | $100 \times 10$ | $200 \times 10$ | |
| $M = 20$ | $\underline{20 \times 20}$ | $\underline{50 \times 20}$ | $\underline{100 \times 20}$ | $\underline{200 \times 20}$ | $500 \times 20$ |

Table 3.1: Sizes of the Taillard PFSP instances, underlined sizes are also used in the Watson-instances

When algorithms are compared using these benchmark sets, two performance measures are often used: the Average Relative Percentage Deviation (ARPD) and the Median Relative Percentage

[1] A problem was considered interesting if the best found makespan was far from a lower bound of the makespans and if many attempts to solve the problem did not provide the same solution.

Deviation (MRPD). Both measures use the distance of $N$ algorithm results ($RES_i$: $i \in \{1 \cdots N\}$) to the best known upperbound ($UB$) of that instance as the main ingredient. Since the result is always higher or equal to the best known upper bound, this distance is defined by $RES_i - UB$. The relative percentage deviation of result $i$ is the distance divided by the upper bound and scaled by a factor 100:

$$RPD_i = \frac{100 \cdot (RES_i - UB)}{UB}. \tag{3.4}$$

Using this definition, we can take the average over $N$ runs of an algorithm as

$$ARPD = \frac{1}{N} \sum_{i=1}^{N} RPD_i = \frac{1}{N} \sum_{i=1}^{N} \frac{100 \cdot (RES_i - UB)}{UB} \tag{3.5}$$

and similarly, the median over $N$ runs can be defined as

$$MRPD = \text{median} \bigcup_{i=1}^{N} RPD_i = \text{median} \bigcup_{i=1}^{N} \frac{100 \cdot (RES_i - UB)}{UB}. \tag{3.6}$$

Since the outcomes of the algorithm are generally not normally distributed, the Mann-Whitney-Wilcoxin test can be used in order to decide whether one algorithm performs significantly better than another. This test only assumes that the underlying distributions of the algorithm results have the same shape.

## 3.3 Constructive Heuristics

Heuristics can be divided into two main categories: *constructive heuristics* and *improvement heuristics*. Improvement heuristics for the PFSP will be further discussed in the next section, where they will be treated as a part of LSs. Constructive heuristics can be further divided into simple and composite heuristics. Constructive heuristics consist of one or more of the following three phases: *Index development*, *Solution construction* and *Solution improvement*. A heuristic is considered to be composite if it uses a simple heuristic in one or more of the three phases.
Since improvement heuristics are often used in the last phase of a construction heuristic, there will be some overlap between this and the next section. In this section, we limit ourselves to a few heuristics. These heuristics are selected based on speed, schedule quality, ability to be used in GAs and variety in use of domain knowledge. This section is primarily based on papers by Ruiz and Maroto [30] and Pan and Ruiz [26], where we refer to for more details and comparisons with other heuristics. For each heuristic, we give a brief explanation and we report its (time) performance for both the $C_{max}$ and TFT criterion.

### 3.3.1 CDS: Campbell, Dudek and Smith

The two-machine PFSP with the $C_{max}$ criterion can be solved in polynomial time. The algorithm solving this PFSP is known as Johnson's rule. Johnson's rule makes one set of jobs having $p(i,1) < p(i,2)$ and one set of jobs having $p(i,1) > p(i,2)$. The first set is sorted on shortest processing time of $p(i,1)$; the second set is sorted on longest processing time of $p(i,2)$. The sets and their jobs are then scheduled in sequence. As sorting is the most expensive operation, this algorithm uses $\mathcal{O}(J \log J)$ time.

Campbell et al. [6] use Johnson's rule as the basis of their simple CDS heuristic. CDS considers the $m-1$ ways in which the machines can be split and grouped into two machines. When the machines up to machine $k$ are merged into the first new machine, the processing times for both new machines are given by

$$p_1(i) = \sum_{j=0}^{k} p(i,j) \text{ and } p_2(i) = \sum_{j=M-k}^{m} p(i,j). \tag{3.7}$$

This new 2-machine problem can be solved using Johnson's rule. The $m-1$ resulting schedules can be used for population-seeding, as we will show in Chapter 6. The CDS heuristic uses $M-1$ evaluations of Johnson's rule and $M-1$ objective function calculations. This gives a computational

complexity of $\mathcal{O}(M^2J+MJ\log J)$. The CDS heuristic does not give very good results, with respect to the best heuristics, it is however able to generate more than one good solution. Though Johnson's rule and thus CDS is designed for the $C_{max}$ criterion, CDS has also been used as benchmark for comparisons of TFT algorithms, performing better than random creation of solutions.

### 3.3.2 Palmer's Slope-Index

Palmer [25] is the first to use indexing in a constructive PFSP heuristic. For every job a *slope index* value is calculated as

$$S_i = -\frac{M-1}{2}p(i,1) - \frac{M-3}{2}p(i,2) - \cdots + \frac{M-3}{2}p(i,M-1) + \frac{M-1}{2}p(i,M). \tag{3.8}$$

The slope index is high, when a job uses little time on the first machines and much time on the last machines. A low slope index is assigned when a job uses little time on the last machines and much time on the first machines. The jobs are scheduled based on the decreasing slope index. Note that Johnson's rule also uses a notion of 'slope': it creates sets with a positive and negative slope. The performance of Palmer's heuristic is slightly worse than that of CDS, though its computational complexity is only $\mathcal{O}(JM + J\log J)$. Palmer's heuristic is designed for the PFSP with the $C_{max}$ criterion and no data is known about its performance on the TFT criterion. In Chapter 8 we will introduce dependency seeding for pGOMEA, which can use Palmer's heuristic for finding dependency values.

### 3.3.3 Rapid Access: Dannenbring

The Rapid Access (RA) heuristic is a combination of the CDS heuristic and Palmer's slope index [10]. The RA heuristics reduces an m-machine problem to a 2-machine problem which can be solved using Johnson's rule. In contrast to the CDS heuristic, machine merging is not performed. Instead, one schedule is generated from a new problem with processing times based on a form of slope index. The processing times used in the new problem are computed by

$$p_1(i) = \sum_{j=0}^{M}(M-j+1)\cdot p(i,j) \text{ and } p_2(i) = \sum_{j=0}^{M} j \cdot p(i,j). \tag{3.9}$$

The processing times on the new first machine is high for jobs with much processing time on the first machines. For the new second machine, jobs with a high processing time on the last machines get a high processing time. RA finds good solutions very quickly: its computational complexity is $\mathcal{O}(JM+J\log J)$. Dannenbring observed that the RA heuristic often results in schedules that were next to optimal, i.e. only two neighboring jobs should be swapped. In order to further increase the efficiency, RA has been combined with an improvement phase with Closed order Search (RACS) and Extensive Search (RAES). RA is designed to solve the PFSP problem with the $C_{max}$ criterion and no data is known about its performance on the TFT criterion.

### 3.3.4 NEH: Nawaz, Enscore and Ham

The NEH heuristic is considered to be the best currently known heuristic for optimizing the PFSP schedules for the $C_{max}$ criterion. However, because of its speed it has also been used as inspiration optimize PFSP schedules with the TFT criterion [26].
NEH calculates its schedule in two steps. First, the jobs are sorted based on decreasing total processing time: $p_{tot}(i) = \sum_{j=0}^{M} p(i,j)$. Then, the best schedule consisting of only the two longest jobs is selected. In the second phase, the other jobs $(3\cdots J)$ are sequentially inserted in the schedule-position where it minimizes the objective. Since job $k$ can be inserted in $k$ positions, a naive implementation needs $\mathcal{O}(J^2)$ function evaluations, resulting in a computational complexity of $\mathcal{O}(J^3M)$.

The best position to insert job $k$, when using the $C_{max}$ criterion, can be found in $\mathcal{O}(k\cdot m)$ time [36] instead of $\mathcal{O}(k^2\cdot m)$ time. This is done using the completion times $c(i,j)$ and the tails $t(i,j)$ (time between the start of an operation and the end of the schedule) of each operation. An overview of this algorithm is given in Algorithm 2

**Result:** The optimal position to insert job $k$
Compute all $c(\pi_i, j)$
Compute all $t(\pi_i, j) = c(\pi_{k-1}, M) - s(\pi_i, j)$
Compute all $f(\pi_i, j) = max\{f(\pi_i, j-1), c(\pi_{i-1}, j)\} + p(\pi_k, j)$
**return** $\arg\min_i\{\max_j[f(i,j) + t(i,j)]\}$.

**Algorithm 2:** NEH outline

Each of the steps in the algorithm can be computed in $\mathcal{O}(kM)$ time. Since NEH uses $n$ insertions, NEH runs in $\mathcal{O}(J^2M)$ time. Note again that this is only true when considering the $C_{max}$ objective. The NEH heuristic is one of the best heuristics for the $C_{max}$ criterion though it has a (relatively) high time complexity. For the TFT problem, it performs worse and also needs more time.

### 3.3.5 LR(x): Liu and Reeves

Liu and Reeves [24] propose a constructive heuristic that creates a variable amount of schedules ($x \leq J$). The LR heuristic is designed for the TFT problem and uses less time than the NEH heuristic, while giving better results for this criterion.
LR(x) consists of the following three steps:

1. Sort all jobs according to the index function.

2. Create $x$ partial schedules with the top-x jobs scheduled first. Extend the partial schedules by iteratively adding the best job according to the re-evaluated index function.

3. Select the best schedule generated in step 2).

The index function for adding job $i$ after the last job $k$ in the partial schedule consists of two components:

1. A *weighted total machine idle time*, punishing the time the machines wait between job $k$ and job $i$. Idle time on the first machines is punished more than idle time on the last machines.

2. The *artificial total flow time*, is the sum of the completion time of job $i$ plus the completion time of an artificial job representing the unscheduled jobs.

As LR(x) gives multiple good solutions in a short time ($\mathcal{O}(x \cdot J^2M)$) we will use the LR(x) heuristic in Chapter 6 when seeding pGOMEA with good solutions.

### 3.3.6 RZ: Rajendran and Ziegler

The constructive heuristic proposed by Rajendran and Ziegler (RZ) [29] consists of two phases. The first phase generates a seed sequence, whereas the second phase tries to improve on that using an improvement heuristic. Since improvement heuristics are explained in the next section, we limit the discussion of RZ to the first phase.
In the first phase, $M$ schedules are generated. Schedule $k$ is generated by sorting the jobs on the sum of their weighted processing times on machines $k$ to $M$:

$$T_{i,k} = \sum_{j=k}^{M} (M - j + 1) \cdot p(i,j). \tag{3.10}$$

The best of the $m$ schedules is determined and used in the improvement phase. The complexity in the first phase is based on $m$ created schedules in $\mathcal{O}(M + J \log J)$ time and $m$ schedule evaluations in $\mathcal{O}(JM)$ time. This results in a complexity of $\mathcal{O}(Jm^2 + MJ \log J)$. When incorporating the RZ-local search (RZ-LS) in the second phase, the complexity becomes $\mathcal{O}(J^3M)$ but results become better than the results of LR. As the RZ heuristic only gives a few solutions, we will not use RZ in population seeding. We will however use the RZ heuristic in dependency seeding in Chapter 8

### 3.3.7 Summary

In this section, the most fundamental constructive heuristics for the PFSP have been discussed. A summary of the discussed heuristics is given in Table 3.2. Most constructive heuristics that are not described here are either composite heuristics using the discussed heuristics or simple heuristics using similar ideas for *indexing* and *solution construction*; and *improvement heuristics* as described in the next section. Though other constructive heuristics are often able to give better results (especially with respect to the TFT criterion), these heuristics also need much more time. This makes these constructive heuristics less suitable for incorporation in an EA.

| Algorithm | Solves Criterion | Complexity | Performance | #Solutions |
|---|---|---|---|---|
| CDS | $C_{max}$ | $\mathcal{O}(M^2J + MJ\log J)$ | OK | $M-1$ |
| Palmer | $C_{max}$ | $\mathcal{O}(JM + J\log J)$ | OK | 1 |
| RA | $C_{max}$ | $\mathcal{O}(JM + J\log J)$ | OK | 1 |
| NEH | $C_{max}/TFT$ | $\mathcal{O}(J^2M)/\mathcal{O}(J^3M)$ | Good/Bad | 1 |
| LR(x) | $TFT$ | $\mathcal{O}(x \cdot J^2M)$ | OK | $x \leq J$ |
| RZ | $TFT$ | $\mathcal{O}(J^3M)$ | OK | 1 ($M$ seeds) |

Table 3.2: Summary of constructive heuristics for the PFSP.

## 3.4 Local Search methods

In many NP-hard problem solvers, LS methods have been incorporated. LSs use a defined neighborhood in order to walk through the search space from one solution towards better solutions. In this section, we will not discuss the concrete implementations of LS methods like Simulated Annealing and Tabu Search; some of them will be discussed in the next section. We limit the discussion to simple neighborhood searchers like the simple first-improvement and best-improvement neighborhood searchers. Other LS methods can be easily derived from these basic neighborhood searchers.

In worst case, one iteration in a neighborhood searcher evaluates the objective function for all neighbors. The complexity of one iteration in an neighborhood search for the PFSP is thus bounded by $\mathcal{O}(|N| \cdot J \cdot M)$, where $|N|$ is the neighborhood size and $J \cdot M$ is the time needed to calculate the objective value of a neighbor. Especially close to local optima, a large part of the neighborhood has to be evaluated before an improvement is found. However, when not optimizing from a BBO perspective, neighborhood searchers can often re-use information about the problem in order to compute the objective value faster. In our experiments we will limit optimizations in fitness evaluations to the level of Big-O notations.

In this section, we discuss neighborhood searchers for the PFSP. For each of the discussed neighborhood searchers we discuss three properties: the *neighborhood space complexity*, the *worst case time complexity* and the *algorithms/heuristics* where they are used in, or derived from. Most improvement heuristics for the PFSP are based on the two most fundamental heuristics for permutation problems: the swap and insertion heuristic. Since other heuristics are often based on these two heuristics, they will be included in the discussion of the basic heuristics.

### 3.4.1 Swap heuristics

The first basic heuristic for permutation problems is the swap heuristic. This heuristic takes the permutation sequence and swaps two items in the sequence. Since each value of the permutation still remains in the new solution, the new solution is a valid permutation. The neighborhood space of the swap heuristic consists of $\frac{J\cdot(J-1)}{2}$ possible swaps, resulting in an $\mathcal{O}(J^2)$ neighborhood space complexity. In the PFSP problem, a swap simply translates to swapping the positions of two jobs in the schedule. A complete search in the space of all swap-neighbors is quite expensive, so heuristics used in literature often use a reduced neighborhood.

Ho and Chang [18] use the *gap* between pairs of jobs in order to optimize a schedule for both criteria. First, a range of considered jobs is defined starting with the full schedule (i.e. range =

$[1, J]$). The two jobs with the largest difference in gap are iteratively tried to swap with the jobs at the ends of the range; after which the range is reduced. When the range is too small, the heuristic is finished. This swapping heuristic does not consider all possible swaps, but uses an intuition for which swaps will be useful. This results in quite a fast heuristic, that can be used in GAs and LSs.

The Forward Pairwise Exchange (FPE) and Backward Pairwise Exchange (BPE) heuristics are introduced by Liu and Reeves [24] in order to increase the performance of their LR-heuristic. FPE($\alpha$) considers swaps starting from first job in the schedule sequence. For the job in position $k$, only swaps are considered with jobs in the positions $[k+1 \cdots k+\alpha]$. One iteration of FPE($\alpha$) considers $\mathcal{O}(J \cdot \alpha)$ swaps. FPE(1) denotes adjacent pairwise exchange and FPE(J) results in the standard swap heuristic.
In contrast to FPE($\alpha$), BPE($\alpha$) starts from the back of the schedule and only considers swaps with jobs preceding the current job. For the LR heuristic, the choice of BPE was better, since it uses a reverse direction of the LR-constructive phase. For schedules that do not contain such a structure, one can use both BPE and FPE.

Closed order search, as used in RA to create RACS, uses adjacent pairwise exchange [10]. Every possible pairwise exchange is considered and the best of the resulting $J-1$ neighboring schedules is selected. Rapid Access with Exhaustive Search (RAES) applies RACS multiple times. RAES is able to find good solutions to the $C_{max}$ PFSP, though it takes considerably more time than RACS.

A last neighborhood search heuristic worth mentioning is the swap heuristic of Suliman [34]. Suliman reduces the neighborhood space by assuming that once a job is moved forward it is never profitable to move it backward.

In general, we can state that the basic swap heuristic is not used very often in literature. Instead, similar heuristics are derived from the swap heuristic. Most discussed reduced-neighborhood swap heuristics have been designed for either the $C_{max}$ or the TFT criterion. This does not imply that they cannot be used or adapted to improve schedules for the other criterion. One should also keep in mind that some improvement heuristics are designed to optimize solutions generated with a specific constructive heuristic (e.g. FPE($\alpha$)). In this case, the design of such an improvement heuristics can be based on assumptions about the schedule to be optimized.

### 3.4.2 Insertion heuristics

The second basic heuristic for permutation problems is the insertion heuristic. This heuristic takes a permutation sequence and puts one item in the sequence at another place. For each item, $J-1$ new places can be considered. Therefore the neighborhood space of the insertion is also quadratic (i.e. $\mathcal{O}(J^2)$). This is often considered too big, so more specialized heuristics have been designed.

The RZ-LS, as proposed by Rajendran and Ziegler [29] relies on the insertion heuristic. The RZ-LS calculates the best insertion for a job and applies that insertion. The jobs are tried to move, starting from the first job in the sequence. One iteration of this heuristic applies at most $J$ insertions while computing $J^2$ insertions. Tseng et al. [40] limit the insertion positions of the job at position $k$ to the positions in $[k-\alpha, k+\alpha]$, resulting in a $\mathcal{O}(J \cdot \alpha)$ neighbor space complexity.

The result of insertions can sometimes be calculated in a faster way. The NEH-insertion operator is able to select the best insert-position for a job in $\mathcal{O}(JM)$ time when the $C_{max}$ criterion is used. Normally, this would take $\mathcal{O}(J^2M)$ time. The NEH-insertion heuristic uses the same speedup as the NEH constructive heuristic uses. A job is simply removed from the schedule, whereafter the optimal insertion position is calculated in the same way as NEH does. NEH-insertion can also remove multiple jobs and add them using the NEH heuristic method. Unfortunately, the NEH-heuristic cannot be sped up for the TFT criterion, which reduces it to a simple insertion heuristic.

Tseng et al. [40] incorporate a Cut-and-Repair approach in their GLS for solving PFSP with the TFT criterion. Instead of randomly taking jobs that will be inserted in a random position, the two best positions for insertion are identified. The two pairs of jobs which have the largest idle time between them are selected. The schedule is cut between these pairs; for each job, the effect

of inserting it in the cut is estimated using the maximum idle time. For the best eight insertions, the unapproximated TFT-value is computed. The best insertion of this eight is selected. The amount of time needed can be written as $2 \cdot n \cdot A + 2 \cdot 8 \cdot B$, where $A$ is the time for computing the maximum idle time and $B$ is the evaluation of the objective function. Therefore, one iteration of this algorithm takes $\mathcal{O}(JM)$ time, while a neighbor space of $\mathcal{O}(J)$ is taken into account. Besides using simple improvement heuristics, we will also experiment with hybridizing pGOMEA with the 'smart' NEH and Cut-and-Repair heuristic in Chapter 7.

### 3.4.3 Summary

In this section, neighborhood searchers for the PFSP have been discussed that can be incorporated in a LS. A summary of the described heuristics is given in Table 3.3. Most improvement heuristics use a swap or insertion heuristic as basis. Though the insertion and swap heuristic cannot speed up the objective function evaluation, domain knowledge can be exploited in the improvement heuristics. Either swaps or insertions are smartly selected or multiple insertions or swaps can be evaluated at once. For the improvement heuristics, few results are known from literature, since they are only evaluated in relation to their meta-heuristic or constructive heuristic. Therefore, the complexities of the neighborhood and the running times are important guidelines for choosing the right improvement heuristic for a GA.

| Name | Based on | Complexity | Neigbor Space | Operations | Details |
|---|---|---|---|---|---|
| Ho-Chang | Swap | $\mathcal{O}(J^3M)$ | $\mathcal{O}(J)$ | $\mathcal{O}(J)$ | Uses gap between jobs |
| FPE(k), BPE(k) | Swap | $\mathcal{O}(k \cdot J^2M)$ | $\mathcal{O}(J^2)$ | $\mathcal{O}(J)$ | One pass heuristic |
| RACS | Swap | $\mathcal{O}(J^3M)$ | $\mathcal{O}(J^2)$ | $\mathcal{O}(J)$ | One pass heuristic |
| RAES | Swap | $\mathcal{O}(x \cdot J^3M)$ | $\mathcal{O}(k \cdot J^2)$ | $\mathcal{O}(x \cdot J)$ | Multipass RACS |
| Suliman | Swap/Insert | $\Omega(J^3M)$ | $\mathcal{O}(J^2)$ | $\Omega(J)$ | One-directional |
| RZ-LS | Insert | $\mathcal{O}(J^3M)$ | $\mathcal{O}(J^2)$ | $\mathcal{O}(J)$ | One pass heuristic |
| NEH-insertion | Insert | $\mathcal{O}(JM)$ | $\mathcal{O}(J)$ | 1 | Fast insertion for $C_{max}$ |
| Cut-and-Repair | Insert | $\mathcal{O}(JM)$ | $\mathcal{O}(J)$ | $\mathcal{O}(1)$ | Small neighborhood<br>Uses approximation |

Table 3.3: Summary of improvement heuristics for the PFSP.

## 3.5 Metaheuristic solvers

Since the PFSP is NP-hard, bigger instances cannot be solved exactly within reasonable time. Though there exist branch-and-bound algorithms that try to do so, they typically are not able to solve instances with more than 50 jobs within reasonable time [21]. Since constructive and improvement heuristics get stuck in local optima, a lot of meta-heuristic algorithms have been proposed that try to find a schedule that is as good as possible. It is not easy to determine which algorithms are state-of-the-art, since few overview papers have been written and results on important benchmarks are not collectively stored. Algorithms solving PFSP ($C_{max}$) that can be considered state-of-the-art are:

- HGA (2006): This Hybrid Genetic Algorithm starts from an initial population with one solution created with the NEH heuristic and $B_i\%$ of the initial schedules are generated with a modified NEH heuristic. In the GA, the fast NEH-insertion heuristic has been used as a first improvement hill-climber. [31]
- IG (2007): This Iterated Greedy algorithm uses the NEH constructive heuristic to build an initial solution. After that, ILS is started, which perturbs the solution by removing jobs. The jobs are then re-inserted using the NEH-insertion heuristic. ILS/IG accepts solutions based on a problem-size dependent simulated-annealing acceptance criterion. Only two parameters are needed: destruction size and a base temperature. [32]
- DDEA (2008): This algorithm uses Discrete Differential Evolution with an NEH-insertion approach in order to solve the PFSP problem with the $C_{max}$ criterion [27].

State-of-the-art algorithms solving the PFSP with the TFT criterion are:

- HGLS (2009): This is a Hybrid Genetic Local Search algorithm, using a Cut-and-Repair operator, or the insertion heuristic with an $\alpha-$range This algorithm gives good results for problems up to 100 jobs, larger instances have not been tested or reported. [40]
- PHEDA (2011): Zhang and Li propose a hybrid EDA which incorporates the longest common sub-sequence into the probability distribution. LS is added in the form of a Virtual Neighborhood Searcher (VNS) including a perturbation using random insertion. The VNS uses both the swap and the insertion neighborhood. [43]
- VNS4 (2012): Costa et al. test six types of VNS combining the classical insertion and swap heuristic. Their VNS4 outperforms all other VNSs and an Asynchronous Genetic Algorithm. [9]
- MRSILS (2013): Dong et al. propose a Multi-restart Iterated Local Search. The LR(2) heuristic is used to find the first solution. Then, a job-insertion heuristic is used to get to a local optimum. When a local optimum is found, the solution is added to a solution pool and a new LS is started from the perturbed best know solution in the pool. Once the pool is full, new LSs are started from a random pool-member. This meta-heuristic outperforms various other algorithms, and is easy to implement. [11]

As one can see, most meta-heuristics use a LS for searching the space locally, while another mechanism is used for the global search. For both the $C_{max}$ and the TFT criterion GLSs are competitive algorithms, indicating that pGOMEA with LS can also be a good algorithm for solving the PFSP. For the $C_{max}$ criterion, the NEH heuristic is present in every state-of-the-art algorithm. As the VNS4 and IG algorithm are easy to implement and since they give good results, we will compare pGOMEA with these algorithms in Chapter 10.

# Chapter 4

# Domain knowledge in Evolutionary Algorithms

GAs and other EAs search the problem space using selection and recombination of the fittest individuals. EAs are able to search the problem-space globally, by exploiting the building blocks of good solutions. For this, one should provide the EA with the right operators and solution representation. Though these factors matter a lot, a good choice does not guarantee fast discovery optimal solutions. If certain problems are tried to solve with a classical EA from a BBO perspective, exponential-time scale-up can be obtained for polynomial-time solvable problems [37] before the optimal solution is found. This is the main reason why exploitation of domain knowledge has been an important part of EA research over the years. And even if incorporating domain knowledge in EAs is not an exigency, it can still be beneficial in terms of population size or convergence speed. Domain knowledge can be incorporated in EAs in multiple ways; in this chapter we consider two categories: one can either incorporate foreknowledge in the EA or one can derive and exploit a model of the problem structure while running the EA.

Foreknowledge of the problem domain is used in almost every EA. The main form of incorporating domain knowledge in EAs is in choosing a solution encoding and designing operators. These adaptations of EAs come in various forms, from a simple binary encoding to the random keys encoding, from one-point crossovers to cycle crossovers and from bit-flip mutation to LS. Most of these adaptations are designed for effective and correct mixing of building blocks. In Section 4.1, we discuss the standard ways of inserting domain knowledge in EAs and how they may apply to pGOMEA

Model learning in EAs has been a more recent approach to guide effective mixing of good solutions. MBEAs learn a model per generation and use this model to guide the creation of a child population. MBEAs are especially useful for optimization in a BBO environment, only the search domain has to be specified, after which the MBEA learns an effective way of generating offspring solutions. A second advantage of MBEAs is the capability of adapting operators to the local structure of the search space, allowing both a local and global search strategy in the EA. As pGOMEA is an MBEA we will explain MBEAs in Section 4.2, where we will compare the exploitation of model knowledge in pGOMEA with some other MBEAs.

Since both incorporating foreknowledge and model-learning are effective strategies to improve the quality of EAs, the combination of these two methods can improve EAs even more. One important pitfall should be avoided when doing this: the more domain knowledge is incorporated, the more the search is steered, resulting in less exploration in the EA. Section 4.3 discusses this topic and describes the posibilities that this combination offers for pGOMEA.

The final topic of this chapter is the use of model-knowledge given domain knowledge. In Section 4.4 we discuss how learned structure can be exploited using a substructural neighborhood in local search.

## 4.1 Domain foreknowledge in Evolutionary Algorithms

EAs incorporate domain knowledge in different ways. First of all, the way solutions are encoded have a lot of influence on the way mixing is performed. Secondly each of the EA-phases can incorporate domain knowledge in the form of adapted operators or added LS. Since LS is not a necessity for most EAs, but an important performance enhancer, the use and pitfalls of LS in EAs gets special attention in this section.

### 4.1.1 Encoding solutions

The right encoding of solutions can help EAs to overcome the creation of invalid child solutions. The choice of a genetic representation is directly linked to the choice of genetic operators. The most common forms of genetic representations are the binary representation, the tree representation and the vector representation. These and other representations have as a main purpose to allow recombination and mutation operators to mix genes properly and effectively. For pGOMEA, we need the random keys vector encoding, further changes in solution encoding are therefore not discussed in this thesis.

### 4.1.2 Operators in the Evolutionary Phases

The operators of an EA can be tuned to fit a given problem and solution representation. Often this is an exigency: without the right operators, the right building blocks are not correctly juxtaposed or infeasible solutions are generated. Multiple phases can be defined in an EA: *Population initialization*, *Selection*, *Offspring generation* and *Mutation*. For each of these phases, the ways domain knowledge can be incorporated to steer the optimization process are (briefly) discussed below. As pGOMEA does not use selection, we skip this.

- *Population initialization:* Normally, the initial population of an EA is generated at random. A good choice for the initial population can hugely increase the convergence speed of an EA. The argument for this is simple: if we are able to approximate the population in generation $k$ in the initial population, we save $k$ population evaluations.

  With domain knowledge, we can *seed* the initial population with such good solutions. One or more good solutions are added to the initial population. Adding only one good individual can already improve the convergence speed of an EA, the good building blocks of this one solution will slowly be introduced in the other individuals (faster when using FI in pGOMEA). This improves the other solutions, while the random solutions will contribute to the exploration capability of the EA. Good solutions can be either the result of a constructive heuristic or a random solution improved by a LS.

  For instance, for the PFSP with $C_{max}$ criterion, one good individual can be generated using the NEH-heuristic or any other good heuristic. More than one good individual can be created by selecting all $m-1$ solutions of the CDS heuristic. When multiple improved solutions are added to the initial population, the diversity can drop, blocking exploration behavior of a GA. Especially when the improved solutions are correlated in some way, diversity is lost. Therefore it can be worthwhile to generate solutions based on more than one heuristic. When this also reduces diversity too much, random solutions have to be added to the initial population. In Chapter 6 we will experiment with population seeding in pGOMEA.

- *Offspring generation (recombination):* EAs can generate offspring in multiple ways. Due to selection, the offspring that is generated should be biased towards the better individuals. For this, a good recombination operator should be defined. Recombination operators always use more than one individual to generate new solutions.

  A classical GA uses crossover to generate offspring; 1-point, 2-point and uniform crossover are the most well known and simple crossover operators in a GA. For ordered or permutation chromosomes, a GA can use a cycle crossover, partially mapped crossover or another sophisticated crossover operator [41]. For partitioning problems the Greedy Partitioning Crossover is a good choice [13]. Other EAs use recombination in different ways. For instance, Evolutionary Strategies perform recombination by calculating the average of two real-valued vectors

[1]; The standard deviation vector is inherited from one of the parents.

In pGOMEA recombination is greedily performed. Since no selection is used in pGOMEA and only improvements are allowed, pGOMEA uses elitism. pGOMEA also uses a more advanced recombination operator (rescaling) using the fact that it solves permutation problems.

- *Mutation:* The goal of a mutation operator is to introduce diversity in populations and to explore the search space locally. Mutation is a unary operator, since it is applied to a one individual. Mutation operators vary from simple bit-flips in binary encodings to Gaussian-mutations in evolutionary strategies. In permutation problems, the mutation operator can be a simple insertion or swap, as introduced in Section 3.4. When LS is applied instead of a mutation, a GLS is created. The goal of this hybridization is to speed up the convergence of the GA. However, diversity can drop since the mutation is not performed anymore and solutions can converge to the same local optimum. This form of hybridizing a GA with LS is denoted Lamarckian learning instead of Baldwinian learning.

  In Lamarckian learning, LS is again seen as acquiring skills during lifetime. These skills are directly inherited by the offspring [22]. An fictional example of Lamarckism would be a blacksmith; through his work, a blacksmith strengthens the muscles in his arms, and thus his sons will have similar muscular development when they mature. Though in nature this is not observed, EAs can hugely benefit from this approach. Lamarckian learning often allows an EA to reach better solutions with less memory and time, while it introduces or removes diversity. When incorporating a LS, one can have one or more of the mentioned effects in mind, but one should note that the other effects can have (negative) impact on the performance of the EA. In Chapter 7 we will experiment with hybridizing pGOMEA with some improvement heuristic given in Section 3.4, but first we look at some practical issues when hybridizing an EA.

### 4.1.3 Issues of incorporating Local search in an EA

Hybridization of an EA has some issues that should be considered before implementation. First of all, LS takes time in the EA. More time is needed per generation when a LS is used in a EA. Though the amount of fitness evaluations increases, these evaluation are often less expensive than regular fitness evaluations. LSs can then exploit domain knowledge in order to compute the same amount of evaluations in less time (like the NEH-insertion heuristic).
If the LS and the EA work together very well (the LS gives the GA information about good areas in the search space), everything is fine. However, a LS can disrupt the building blocks that an EA needs. Therefore LS is only helpful when locally optimized solutions still contain the important global building blocks that an EA needs.
Since using LS every iteration can take too much time and because it can drop population diversity, a balance between local and global search (and exploration and exploitation) should be found. Therefore different parameters have been introduced that allow the user to specify this balance:

- Frequency of local search ($f$): LS is performed every $\frac{1}{f}$ iterations. This allows a GA to improve globally before improving locally.
- Depth of local search ($k$): The better a solution is, the less neighbors have a better fitness. The last steps of a LS take the most time, therefore stopping a LS after $k$ fitness evaluations or $k$ improvements can be beneficial. A second advantage of this limit is that diversity is not lost when two solutions tend to move to the same local optimum.
- Probability of local search ($Pr_{ls}$): LS is here applied to every individual with a probability of $Pr_{ls}$. This parameter can be seen as a combination of the other two parameters. LS is not performed every iteration for every solution, while the probability that two individuals converge to the same solution in one generation is decreased.

Note that the choice of the parameter values is important. The optimal value can vary per problem and even per generation. For the estimation of the optimal parameter values an initial experiment is often performed to determine the optimal values of the parameters. When we hybridize pGOMEA with some improvement heuristics, we will test limiting the LS by a probability and a depth. We will also research the behavior of pGOMEA with respect to different values for these parameters. Using these results, we determine the best way of hybridizing pGOMEA with LS.

## 4.2 Model Based Evolutionary Algorithms

MBEAs are designed to exploit learned problem structure in the recombination phase of an EA. An MBEA learns a model per generation, which is then used for creating a new populations. Therefore, MBEAs can be seen as a combination of EAs and machine learning. As in machine learning, MBEAs do not exhaustively search for the best model representing the domain structure, rather they find a good model in a shorter time. The right model is always a trade-off between complexity, building time and information.

The main category of MBEAs is that of the EDAs. EDAs generate offspring using sampling from a learned distribution over the selected parent population. The model learned by an EDA is often probabilistic. For instance, the extended Compact Genetic Algorithm (eCGA) learns a simple MPM [16]. Bayesian Optimization Algorithm (BOA) learns a more complex Bayesian network from which the solutions are sampled [28]. In permutation problems, sampling can be performed using the random keys encoding [3] or using a more advanced probabilistic model [7].

Unlike EDAs, GOMEA does not use a probabilistic model to create offspring. Instead, GOMEAs crossover operator uses an optimal mixing approach, GOMEA uses the dependencies captured in the FOS as crossover masks. In this way pGOMEA samples neighbors using genes from other individuals in the generation. Like pGOMEA, DSMGA-II adopts two mixing operators: restricted mixing and back mixing [19].

Though most MBEAs focus on the recombination phase, the mutation phase in EAs can also use a model. For instance, Evolutionary Strategies use self-adaptation by maintaining a vector of standard deviations for each solution. In general, MBEAs tend to perform well on BBO problems, since they exploit the (unknown) structure the best. From a white-box perspective, MBEAs can still perform very well, especially when combined with some form of LS.

## 4.3 Domain foreknowledge in Model Based Evolutionary Algorithms

MBEAs are a relatively new field of study, but in an early comparative study of Zochlin and Dorigo it was already mentioned that *Although using local search is not a common practice in the EDA research field, the results ... indicate that it certainly should be considered in the future.* and *... the use of the local search leads to a drastic improvement of performance ...* [44]. Like in normal EAs, LS can be incorporated in different ways in an MBEA. However, the functionality of an LS gets broader. Once a population is locally optimized, the model learns dependencies between variables in local optima. Therefore, the model will contain more structure and less diversity.

For EDAs, different LS approaches have been used in practice. Hauschild and Pelikan use seeding in order to enhance the performance of their EDA [17]. Duque et al. [12] show that the population size of eCGA can be significantly reduced by using LS. They also show that the amount of required fitness evaluations drops, even when a simple bit-flip operation is assumed to be expensive. Chen et al. [8] perform a study that compares the behavior of eCGA and DSMGA with a first improvement hill-climber and a best improvement hill-climber. Their experiments show that a hybrid MBEA with a greedy hillclimber is able to give better solutions with less fitness evaluations. The best improvement hill-climber has too much overhead in determining which neighboring solution is best. A lot of other algorithms show the value of adding a LS in EDAs. Overall, the results suggest that EDAs are able to improve in quality when they are combined with LS.

In contrast to EDAs, GOMEA has not been studied thoroughly with LS. A first study with GOMEA and LS shows that from a BBO-perspective (i.e. no partial fitness evaluations are possible), GOMEA does not improve when a LS is incorporated [4]. This is an indication that OM is already an effective mix of an EA and LS. However the performance of GOMEA was already very good and the performance of LS was already bad on the used problems. Therefore, the question is still open whether LS is still superfluous when considering real-world problems (e.g. permutation problems) outside a BBO setting.

In literature we have not found any algorithms exploiting domain knowledge in building the model, instead the algorithms all use knowledge derived from the population. In this thesis we will however introduce a new form of seeding for MBEAs in Chapter 8. Model seeding or informed model learning (for pGOMEA: dependency seeding) allows an MBEA to learn structure from the first generations on, by learning a model that is (partially) based on domain knowledge.

## 4.4 Exploiting model-knowledge using domain-knowledge

We have seen how (MB)EAs can be enhanced using domain knowledge. In literature, there also exist various ways of combining domain knowledge with model knowledge. In pGOMEA one such combination is already present. pGOMEA has an crossover operator that uses information from the FOS-model. The standard mixing operator takes a FOS subset as a crossover mask for recombination. Therefore, the learned domain knowledge steers the optimization process. Since the rescaling operator is incorporated in the GOM phase it can be seen as an model-dependent insertion heuristic. The rescaling operator inserts one or more variable-values from one solution into another while preserving the variable order. Though the rescaling operator is very similar to the insertion heuristic for PFSP, it is not exactly the same. Differences are:

- Rescaling works on more than one variable
- Rescaling not only moves variables, but also scales them
- Rescaling is performed during donation and not during a separate LS phase.

The re-encoding operator in pGOMEA also has some relation with the learned model. Since random keys are reassigned, the proximity dependency between two variables will mainly dependent on distance between the values in the permutation. The random keys only add randomness to this squared distance.

Like the random rescaling in pGOMEA, EDAs can also incorporate operators that use the model-knowledge. Lima et al. incorporate the model knowledge of EDAs in a LS, by defining a substructural neighborhood [23]. A substructural neighborhood is defined by linkage groups learned by the BOA EDA. A substructural neighborhood consists of sets of variables that are dependent in the learned Bayesian Network model. The authors define three neighborhoods:

- Parental neighborhood (a node plus its parents in the Bayesian Network)
- Children neighborhood (a node plus its children in the Bayesian Network)
- Parental + Children neighborhood (a node plus its parents and children in the Bayesian Network)

These neighborhoods can be incorporated in a LS, by considering (all) other assignments for the variables in a substructure. BOA hugely benefits from this approach and eCGA can also be greatly sped up using substructural neighborhoods.

A form of substructural neighborhood has also been used by Iclazan and Dimitrescu [20]. Their Building-Block hill climbing algorithm learns building blocks based on the results of a hill-climbing algorithm. The building blocks are then used in the same hill-climbing algorithm. In this way, the experience of the hill-climber is used to improve itself.

For GOMEA, a substructural neighborhood can be defined as a set in the FOS. This approach has already been researched, and it did not give very promising results. [4]. However, this does not give an indication whether substructural neighborhoods will be helpful for pGOMEA or not. In the given research, only problems are considered that are efficiently solved by GOMEA and are unsolvable using LS. In more real-world problems, LS is more effective and a discrete (binary) search space is way smaller than permutation space. Therefore, using a substructural neighborhood in LS is still a possible worthwhile extension for pGOMEA, which we will examine in Chapter 9.

# Chapter 5

# Improvement heuristics on pGOMEA solutions: Experimental Study

In order to get a first intuition about the local optimality of schedules in pGOMEA we investigate the improvability of pGOMEA solutions with respect to basic improvement heuristics. This part of the research is conducted to see if there are any quick wins, by using simple improvement heuristics on the resulting schedule of pGOMEA. We experimentally compare the quality of the basic swap and insertion neighborhood as described in Section 3.4.

Since pGOMEA works with an population sizing scheme, multiple populations are evaluated contemporaneously. Some populations are evaluated once, while others have already converged. Therefore it is expected that recently added populations will contain a lot of 'random' solutions that can be easily optimized using improvement heuristics. On the other hand, converged populations have used a lot of donations with rescaling. As rescaling on a donation with a singleton linkage-set is equivalent to an insertion operator, the converged populations are expected to have locally optimized solutions (with respect to the insertion heuristic).

Because of this we do not focus on the optimizability of solutions in general. Instead we target the optimizability of the elitist solution over all populations. The elitist solution is supposed to be the least optimizable solution after termination of pGOMEA. Since the elitist solution has the highest fitness of all solutions, it can be optimized less compared to all other solutions. All results of this experiment are conservative about the performance of the improvement heuristics. If the elitist solutions cannot be optimized, this does not necessarily imply that improvement heuristics are unable to improve pGOMEA. On the other hand: if elitist solutions can be easily optimized, this is an indication that improvement heuristics can be very worthwhile for pGOMEA. We will use the results of this first experiment as guide for hybridizing pGOMEA in Chapter 7.

The contents of this chapter are organized as follows: Section 5.1 gives the setup of the experiments. Section 5.2 discusses the results of the different sub-experiments. An overview of the conclusions is given in 5.3. The main question that the experiments try to answer is:

*How do different improvement heuristics perform on the solutions of pGOMEA?*

## 5.1 Experimental setup

### 5.1.1 Benchmark and computational budget

For this and most other experiments, we use instances from the *Taillard benchmark* (see Section 3.2) to test the performance of algorithms. The used instances have a larger size than those optimally solved by pGOMEA in [2], therefore, we use instances with *sizes* given in 5.1. Since the computational budget of this research is limited, we use a lower amount of fitness evaluations than in [2]. Instead, we use amounts of *fitness evaluations* that are one to three orders of magnitude lower, details are given in 5.2. The given computational budgets result in reasonable computation times starting from a few seconds to a few minutes.

| Jobs/Machines | 5 | 10 | 20 |
|---|---|---|---|
| 50 | $(50 \times 5)$ | $(50 \times 10)$ | $(50 \times 20)$ |
| 100 | $(100 \times 5)$ | $(100 \times 10)$ | $(100 \times 20)$ |

Table 5.1: Used Taillard-instances for experiments.

| Denotation | Low | Standard | High |
|---|---|---|---|
| Evaluations | 220,712 | 2,207,121 | 22,071,125 |

Table 5.2: Used fitness evaluations and their denotation in text.

### 5.1.2 pGOMEA Configuration

As configuration of pGOMEA, we used the settings that give the best results for pGOMEA on the TFT criterion. These settings are described by Bosman Et Al. [2]. The settings include both rescaling (with probability 0.1) and re-encoding of solutions. The proximity and relative ordering information are used to build a linkage tree model of the variables. Population sizing is used with each population being evaluated four times as often as the population of twice its size. Populations are only stopped on convergence, not when they are overtaken by another population. Experiments are initially performed on both the TFT and $C_{max}$ criterion.

### 5.1.3 Neighborhood searchers

The elitist solution obtained by pGOMEA will be optimized using both an exhaustive swap and insertion heuristic as well as a VNS combining both heuristics. For information about these heuristics, we refer to Section 3.4.

### 5.1.4 Comparing results

We report whether the elitist solutions can be improved using the heuristics and what difference it makes on the resulting solution. For this, we use two measures:

- **Probability of improvement:** Each of the ten instances in a $J \times M$ benchmark-set is run five times. The probability of improvement gives the proportion of these 50 runs that can be improved using a heuristic.
- **Average MRPD:** In order to measure the quality of a solution, we should use a resistant statistic measuring RPD values. Since each set $J \times M$ contains 10 instances, we cannot take the median value over 50 runs. The median does not take the hard and easy instances into account, median fitness or median RPD is therefore not a representative measure. Instead we use an average-of-medians approach. Each of the ten instances in a $J \times M$ benchmark-set is run five times. For each of the instances we take Median RPD (see 3.2). The average over the 10 MRPD values is reported as the average MRPD (AMRPD) value. In a similar way, we report the average median fitness.

## 5.2 Results

### 5.2.1 Probability of improvement

First we determine the probability that a pGOMEA elitist solution can be improved using a swap or insertion heuristic. Table 5.3 shows for each instance-set the probability that the elitist solution of pGOMEA can be improved using the simple improvement heuristics.

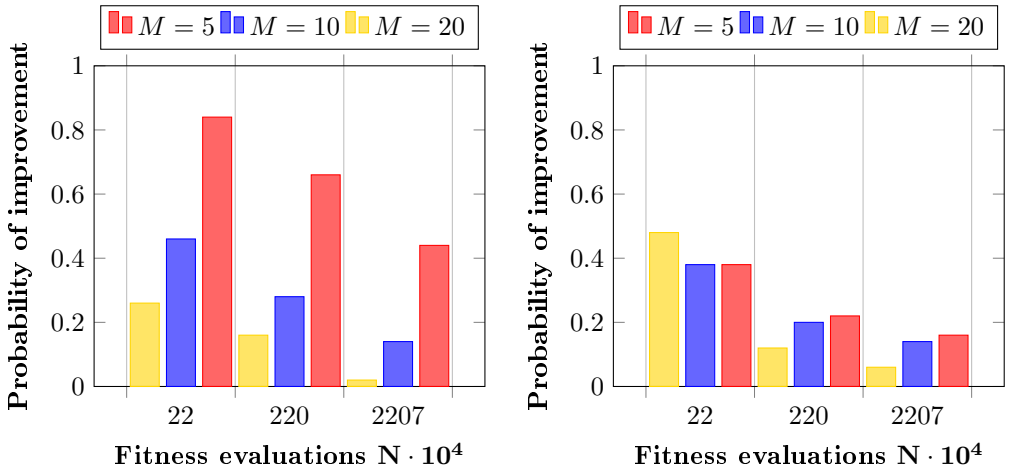
The major observation is the difference in probability of improvement for the swap and the insertion heuristic. The swap heuristic gives a higher probability of improvement than the insertion heuristic, especially for the TFT criterion. Secondly we observe that $M$ has a lot of influence on the probability of swap-improvement for the TFT values. As $M$ gets higher, the probability of improvement drops very fast. However, when $J$ gets higher, the probability of improvement also gets higher. Lastly, we see a big difference in probability of improvement between the TFT and $C_{max}$ criterion.

| | TFT | | $C_{max}$ | |
|---|---|---|---|---|
| Instances | Swap | Insert | Swap | Insert |
| $50 \times 5$ | 0.66 | 0.22 | 0.02 | 0.00 |
| $50 \times 10$ | 0.28 | 0.20 | 0.04 | 0.04 |
| $50 \times 20$ | 0.16 | 0.12 | 0.06 | 0.08 |
| $100 \times 5$ | 0.98 | 0.46 | 0.02 | 0.04 |
| $100 \times 10$ | 0.72 | 0.46 | 0.00 | 0.04 |
| $100 \times 20$ | 0.42 | 0.36 | 0.16 | 0.18 |

Table 5.3: Probability of improving the elitist solution using improvement heuristics. Solution found after 2,207,121 pGOMEA fitness evaluations.

### 5.2.2 Probability of improvement: Machine influence

In order to get more insight in the behavior of improvement heuristics with respect to $M$, we look in more detail to the probability of improvement with respect to the value of $M$ in this experiment. We now limit the experiment to the TFT criterion as this is most influenced by the amount of machines. In this experiment we vary the amount of fitness evaluations and machines. We report the probability of optimization for each combination of $J$ and $M$ with respect to a low, standard and high amount of fitness evaluations by pGOMEA. The results of this experiment are shown in Figures 5.1 and 5.2 for $J = 50$ and $J = 100$ respectively.




(a) Swap heuristic

(b) Insertion heuristic

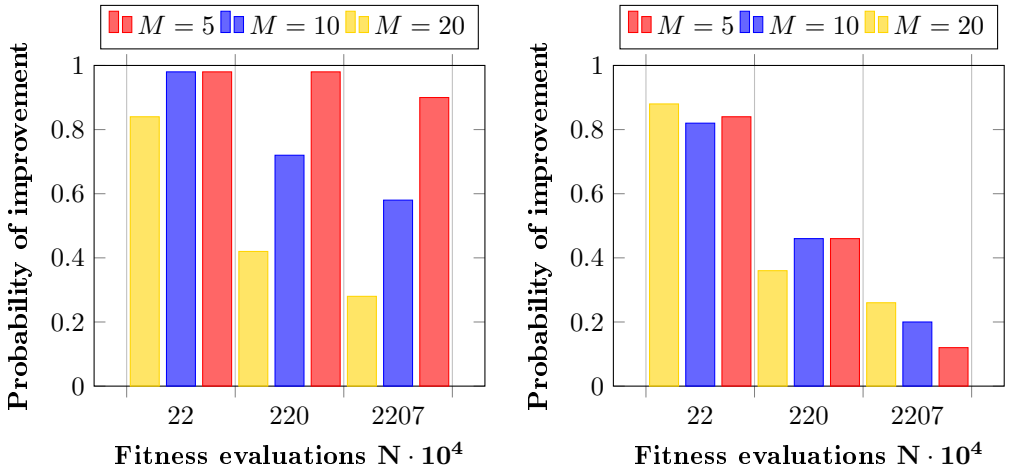
Figure 5.1: Optimizability of elitist solutions with respect to the swap and insert heuristic using a different amount of machines and a fixed amount of jobs ($J = 50$).

These figures confirm that the larger values of $M$ give a lower probability of improvement for the swap heuristic. Also, a higher amount of jobs results in a higher probability of improvement for the swap heuristic. The insertion heuristic on the other hand seems not to be influenced by the amount of machines. When considering 100 jobs and 22,071,215 fitness evaluations for pGOMEA, we can even see the opposite: more machines results in a higher probability of improvement.

The results in Figures 5.1 and 5.2 make that we cannot conclude that the swap heuristic always performs equally well or better than the insertion heuristic. Most of the time, the swap heuristic has a higher probability of optimization, but the insertion heuristic can perform better when a lot of fitness evaluations are used by pGOMEA and the problem instances have a lot of machines.

### 5.2.3 Improvement heuristics for TFT: Quality and resources

Since the probability of improvement is not necessarily strongly correlated with the quality improvement, we perform a second experiment. Here, we also look at the amount of swaps or insertions that are performed and the average MRPD and average median fitness values after improvement by the heuristics. As we expect that the amount of swaps and insertions is not very dependent




(a) Swap heuristic

(b) Insertion heuristic

Figure 5.2: Optimizability of elitist solutions with respect to the swap and insert heuristic using a different amount of machines and a fixed amount of jobs ($J = 100$).

on the instance, we do not use a average-of-medians approach for the amount of improvements. Instead, we report the average amount of swaps and insertions that are performed.
Besides the quality of improvement, we also discuss the resources used in the improvement. We measure this in terms of the amount of fitness evaluations used in the heuristics. We report the average amount of fitness evaluations.

The results of this experiment, using a small amount of fitness evaluations are shown in Table 5.4 for the $J \times 10$ instances with the TFT criterion.

| ($M = 10$) | $J = 50$ | | $J = 100$ | |
|---|---|---|---|---|
| | Swap | Insert | Swap | Insert |
| Avg. heuristic fitness evaluations | 3261 | 4004 | 30095 | 41128 |
| Avg. heuristic improvements | 0.98 | 1.68 | 12.98 | 19.92 |
| Avg. of Median amount of fitness improvements | 32.0 | 42.2 | 811.6 | 901.4 |
| Avg. of MRPDs after heuristic | 2.396 | 2.385 | 3.686 | 3.653 |

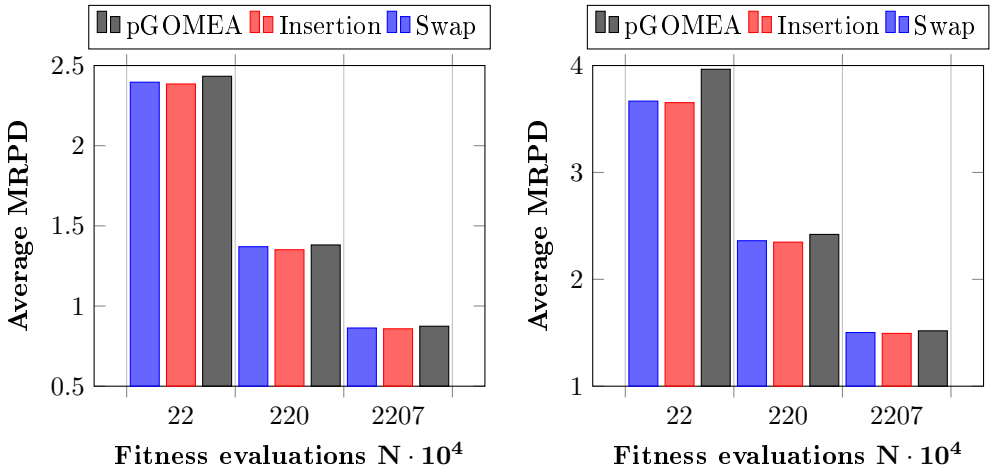
Table 5.4: Improvement results using improvement heuristics. For 220,712 pGOMEA evaluations using the TFT criterion.

The results show that, though the swap heuristic has a higher probability of improvement, the insertion heuristic improves the solutions more. On average the insertion heuristic performs more changes than the swap heuristic. This also results in more fitness evaluations used by the insertion heuristic than the swap heuristic, since after each improvement, the complete neighborhood should be revisited before the heuristic terminates.

Since more insertions than swaps are performed, the fitness is improved more by the insertion heuristic than by the swap heuristic, resulting in better RPD scores. We can therefore conclude that the probability of improvement has no strong relation with the expected quality of improvement. Though initially swaps are often possible, they can be applied only a few times. Insertions on the other hand are less often possible, but if one insertion is possible, it creates new possibilities for insertions.

These observations are further tested, the average MRPD values are also calculated with more fitness evaluations used by pGOMEA. The results of these experiments are shown in Figure 5.3 for the TFT criterion.

The results of this experiment confirm that pGOMEA generates solutions for the TFT criterion that can be easily improved using the swap heuristic, but that the insertion heuristics creates slightly better solutions. The results also show the impact of an improvement heuristic with respect




(a) $J = 50$ (b) $J = 100$

Figure 5.3: Optimization quality of the insertion and swap heuristics on elitist solutions of pGOMEA. Optimization measured using different amounts of fitness evaluations and a fixed amount of jobs an Machines ($M = 10$)

to the amount of pGOMEA fitness evaluations. Whereas a few pGOMEA fitness evaluations give a lot of room for improvement, more evaluations only decrease the MRPD slightly.

### 5.2.4 Improvement heuristics for $C_{max}$: Quality and resources

The results in Table 5.3 show that the solutions of pGOMEA generated by the $C_{max}$ criterion cannot often be improved using the swap and insertion heuristic. Therefore, we conduct an experiment in order to get more insight in the improvability of pGOMEA solutions for the $C_{max}$ criterion. The experiment focuses more on quality of improvement and used resources; we use Taillard instances with sizes $(50 \times 10)$ and $(100 \times 10)$. Since improvements are hard on the $C_{max}$ criterion, we let pGOMEA use a small amount of fitness evalutations. The results of the experiment can be seen in Table 5.5.

| $(M = 10)$ | $J = 50$ | | $J = 100$ | |
|---|---|---|---|---|
| | Swap | Insert | Swap | Insert |
| Avg. heuristic fitness evaluations | 2671.4 | 2793.2 | 10639 | 13181 |
| Avg. heuristic improvements | 0.16 | 0.22 | 0.22 | 1.14 |
| Avg. improvement per swap/insert | 3.0 | 2.55 | 1.91 | 2.63 |
| Avg. of MRPDs after heuristic | 1.811 | 1.828 | 0.886 | 0.866 |

Table 5.5: Improvement results using improvement heuristics. For 220,712 pGOMEA evaluations using the $C_{max}$ criterion.

These results show that the small probability of improvement is not due to the $C_{max}$ solutions being optimal. The MRPD values are far from optimal, even after further improvement. When a solution can be improved using either the swap or insertion heuristic, this does not introduce a lot of new improvement possibilities. In the best case, an average of only 1.14 improvements are performed, with an improvement of fitness of 2.63 per insertion. Consequently, the MRPD only decreases very slightly and a near-optimal solution is not found. Experiments using more fitness evaluations for pGOMEA resulted in even lower optimization rates for the heuristic. When using more evaluations, pGOMEA was able to find much better solutions.

### 5.2.5 Variable Neighborhood Searching

As the swap and insertion heuristics have their own strengths, combining them into a VNS can possibly create a much stronger improvement heuristic. Since we are only interested in the improvability of the elitist solution, this VNS does not contain any shake procedure, thus terminating when a local optimum is found for both the insertion and swap heuristic. Ideally, this combination of the swap and insertion heuristic should inherit the high probability of improvement from the swap heuristic and the solution quality from the insertion heuristic. Since the VNS searches two neighborhoods, this will evidently cost more fitness evaluations and time. The used VNS heuristic is based on the VNS4 heuristic by Costa [9]. The VNS first searches the swap neighborhood until a local optimum is found. Then, the insertion neighborhood is used.

| | | Improvement prob. | | | Average MRPD | | |
|---|---|---|---|---|---|---|---|
| Instance | Evaluations | Insert | Swap | VNS | Insert | Swap | VNS |
| (50x10) | 220712 | 0.46 | 0.38 | 0.62 | 2.396 | 2.385 | 2.319 |
| | 2207121 | 0.28 | 0.20 | 0.32 | 1.370 | 1.351 | 1.344 |
| | 22071215 | 0.14 | 0.14 | 0.22 | 0.863 | 0.858 | 0.850 |
| (100x10) | 220712 | 0.98 | 0.82 | 0.98 | 3.687 | 3.654 | 3.496 |
| | 2207121 | 0.72 | 0.46 | 0.82 | 2.361 | 2.348 | 2.292 |
| | 22071215 | 0.58 | 0.20 | 0.58 | 1.494 | 1.502 | 1.470 |

Table 5.6: Comparing Insertion, Swap and VNS heuristic on the TFT criterion

Table 5.6 shows how VNS combines the benefits of the swap and insertion heuristic. Using the VNS gives the highest probability of improvement and the lowest MRPD values. This does however come at some cost. The amount of fitness evaluations used is more than three times as big as for the insertion or swap improvement. Also, though the MRPD values get better, the improvement is still very small when pGOMEA has used a high amount of fitness evaluations.

## 5.3 Conclusions

In this section, we explain the observations made in the analysis of the experimental results. We then motivate the expected reason for these results. Finally, we draw some conclusions about pGOMEA and its interaction with simple improvement heuristics.

- **Results for the $C_{max}$ criterion by pGOMEA can hardly be improved.**
  Even when a small amount of fitness evaluations is considered, the elitist solution for the $C_{max}$ criterion can improved less than 20% of the time. This is a big difference with the TFT criterion, which can be optimized more than 40% of the time when considering the same amount of fitness evaluations. Multiple factors can be the cause of this behavior.

  First of all, it can be due to the nature of the $C_{max}$ criterion. This criterion only considers the completion time of the last scheduled job. Therefore, an insertion or a swap that does not change this completion time, will not be considered an improvement. For the TFT criterion, the same insertion or swap can still change the completion times of other jobs, leading to an improvement of the objective value. Solutions have therefore less neighbors with a different (and better) fitness for the $C_{max}$ criterion, than neighbors with a different fitness for the TFT criterion.

  A second reason for the optimizability of $C_{max}$ and TFT solutions can be the problem difficulty. If the instances are easier solved for the $C_{max}$ criterion than for the TFT criterion, the probability of improvement is much lower for $C_{max}$ results. The experiments however have shown that solutions can hardly be optimized, even when the MRPD values are quite large $> 1.5$. TFT solutions however can still be optimized when MRPD values are below 1.5. The main reason for the difference between TFT and $C_{max}$ lies therefore in the amount of improving neighbor solutions due to the way the fitness function is defined.

Since the $C_{max}$ solutions are hardly optimized using simple improvement heuristics, there is an extra need for non-BBO improvement heuristics, like the NEH improvement heuristic. If one works in the BBO context, the use of an improvement heuristics has little effect on the results, while a lot of fitness evaluations are needed when working in a BBO context.

- **Improvement strength decreases when more fitness evaluations are considered**
  As shown in Figures 5.1 and 5.2, a larger amount of fitness evaluations used by pGOMEA results in a lower probability of optimization using an improvement heuristic. This trend can be observed for both the swap and insertion heuristic on the TFT criterion. For this behavior, there is an easy explanation. More fitness evaluations in pGOMEA results in better solutions. Better solutions have less neighbors that are better than them. Therefore, the results of these pGOMEA runs are less likely to be optimized by the swap and insertion heuristic.

- **The more machines an instance contains, the less often the swap heuristic can improve the elitist solutions**
  For the TFT criterion, extensive experiments indicate that (for both $J = 50$ and $J = 100$), the probability of improvement using the swap heuristic is related with the amount of machines. The insertion heuristic on the other hand is hardly influenced by the amount of machines. A possible explanation lies in the probability that jobs look alike.

  When only two machines are used, jobs can have different processing times in at most two variables. Swapping two jobs having one variable in common has little destructive effects, since the other value either gives a better or worse schedule between the two jobs that are swapped. When more machines are used, jobs are less likely to look alike. Therefore there is a higher probability that one of the variables that differ a lot has a destructive effect on the schedule between the two swapped jobs. Consequently, the more machines are used, the higher the probability of a destructive swap operation and the lower the probability that a swap will improve the solution quality.

- **The insertion heuristic has a lower probability of improving a solution, but its effect when improving a solution is larger**
  In our experiments we observed that, though the swap heuristic can improve more solutions of pGOMEA, the insertion heuristic improves the pGOMEA results more in terms of fitness. Table 5.4 shows how the insertion heuristic improves the fitness more than the swap heuristic does, even though the swap heuristic improves with a higher probability. This result can possibly be explained by the local effect of an insertion. Though the job is moved towards a good position, it connects its original predecessor and successor. This connection might contain a gap, where any other job can fit in.

- **A Variable Neighborhood Searcher is able to combine the benefits of insertion and swap improvement, though this does come at a big cost.**
  The insertion and swap heuristic can be combined in a VNS. Though this does indeed give better results for both the probability of improvement and the MRPD value, this does take more fitness evaluations. The VNS uses about three times as much fitness evaluations as the insertion heuristic and swap heuristic. On instances that are the result of a lot of pGOMEA fitness evaluations, the change in MRPD value is still not very high.

In general, we can conclude that for now, the $C_{max}$ criterion shows little room for improvement using simple improvement heuristics, while solutions for the TFT criterion are more easily optimized using simple improvement heuristics. For the TFT criterion the insertion heuristic is the best choice for finding a good solution, while the swap heuristic has the highest probability of improving a solution. Though we saw that more fitness evaluations led to a lower probability of optimization, this does not mean that improvement heuristics are less useful when considering a high amount of fitness evaluations. When using a high amount of fitness evaluations, the first solutions in a generation still benefit from improvement heuristics. Since we have only seen that the elitist solution does not benefit very much from improvement heuristics, other solutions in an almost converged population can still be optimizable, which improves pGOMEA. In the Chapter 6 we use this observation by optimizing the initial members of each population. In Chapter 7 we will also experiment with hybridizing pGOMEA using improvement heuristics.

# Chapter 6

# Solution seeding pGOMEA: Experimental Study

As we have seen in Chapter 5, pGOMEA solutions can be improved using improvement heuristics. The less pGOMEA has optimized solutions, the more effect local search has. In this chapter we therefore use heuristics at the start of pGOMEA i.e. we use solution seeding as introduced in Section 4.1.2. We call this well-known form of seeding *solution seeding* to distinguish it from dependency seeding as introduced in Chapter 8. We will research the effects of combining constructive heuristics for the PFSP problem with TFT and $C_{max}$ criterion with pGOMEA. Using constructive heuristics, we seed pGOMEA with good solutions, which pGOMEA can possibly learn and exploit. The main question that this experimental study will answer is:

*What is the effect of solution seeding on the performance of pGOMEA?*

The examined forms of solution seeding are introduced in 6.1. The setup of the experiment is given in Section 6.2, after which we experimentally test seeding in Section 6.3. In Section 6.4 we draw conclusions based on these experiments.

## 6.1 Forms of seeding

Seeding in pGOMEA should take into account that pGOMEA uses a population sizing scheme. Adding one good solution in the first population has little to no effect on later populations. The only way that populations interact is via the forced improvement (FI) phase, where the overall-best solution is used as a donor. It is therefore expected that each population should be seeded in order to improve on the quality of pGOMEA. Combining pGOMEA with heuristics will be done in three different ways:

- **Elitist seeding:** The initial elitist solution will be generated using a constructive heuristic. The contents of a population remain unchanged. Using this form of seeding, we can prevent the seed from becoming too dominant, since it will only be used in the FI phase of pGOMEA.

- **Single-solution population seeding:** Each new population will contain one solution that is generated using a constructive heuristic. This form of seeding leaves room for diversity, provided that it does not dominate the random solutions too much.

- **Multi-solution population seeding:** Each new population will contain multiple solutions that are generated by a constructive heuristic. With this form of seeding we can further analyze the aspects of diversity and convergence, with respect to the amount of domain knowledge given.

The first two forms of seeding are a form of single solution seeding, which allows one to use any heuristic algorithm that return a valid solution. The last two forms of seeding are a form of population seeding, where the seed is entered in every population.

## 6.2 Experimental setup

For this experiment, the Taillard instances are used as a benchmark. The used sets of instances have 50 or 100 jobs and 5, 10 or 20 machines, making a total of 60 instances. Each result is obtained by running pGOMEA five times on the instances, with the given amount of fitness evaluations. Quality measures as introduced in Section 5.1 are used.

Three experiments are performed, related to each of the ways in which pGOMEA can be seeded. First, single solution population seeding is researched, secondly elitist-seeding is used. For these experiments, we use the RZ and NEH constructive heuristics for the TFT and $C_{max}$ criterion respectively. The performance of the seeded pGOMEA algorithms is compared to the performance of pGOMEA without any seeding. Lastly, we experiment with multi-solution seeding. Here we use the LR and CDS constructive heuristics for the TFT and $C_{max}$ criterion respectively. This implies that for CDS at most $M-1$ solutions are seeded and for LR at most $J$ solutions are seeded. Finding the right amount of seeds will be further elaborated in the results. The pGOMEA settings are the same as in the previous chapter. For more information about the used heuristics, we refer to Section 3.2.

## 6.3 Results

### 6.3.1 Single-solution seeding: solution quality

In this first experiments we look at seeding with a single solution for either elitist-seeding or single-solution population seeding. The seeds are generated using the RZ or NEH constructive heuristic. Whereas the population seeding enters the solution in the population, elitist-seeding only enters the solution as the elitist solution.

Single-solution population seeding can possibly lead to a lack of diversity in the population, especially in small populations. At the start of a population, the seeded solution is much better than all random others. Therefore, a donation with the seed will often be accepted, whereas a donation with a random donor will often be rejected. Thus, the genes of the seeded solution can possibly take over the whole population, resulting in a loss of exploration behavior. In this case, the population does converge to a solution that looks a lot like the seeded solution which is not necessarily a near-optimal solution.

Elitist seeding can possibly overcome this problem. Since the elitist solution is only used as a donor when the FI phase is entered, it is less likely that it takes over the whole population. Elitist seeding also has its disadvantage, the elitist solution is shared between all populations. Once the seeded elitist solution has been overtaken in any population, it is replaced. The seeded elitist solution only has effect to the point that a population of pGOMEA would have found an equal-quality solution as the seed. Also note that elitist seeding is a part of single-solution population seeding; if the seed is entered in the first population, it will probably be the elitist.

In order to test single-solution population seeding and elitist seeding in practice, pGOMEA with and without these forms of seeding has been tested on 60 Taillard instances, using the standard amount of 2,207,121 fitness evaluations. For each of the pGOMEA variants and instance sizes, the average MRPD values of the solutions are shown in Table 6.1.

These results lead to some interesting observations. First of all, for the $C_{max}$ criterion, population seeding does improve pGOMEA in only two cases. In these cases, population seeding does hardly improve on the already good solutions found. For the TFT criterion we see that for $J = 50$, population seeding does not (greatly) improve pGOMEA. For $J = 100$, this observation does not hold anymore. Instead, pGOMEA is consistently improved by adding a heuristic solution to each population.

Secondly we see that elitist seeding only results in small changes in the average MRPD values for both the TFT and the $C_{max}$ criterion. This small change often has the same positive or negative effect as population seeding has. Figure 6.1 shows how the results of population seeding compares

| | TFT | | | $C_{max}$ | | |
|---|---|---|---|---|---|---|
| Instances | pGOMEA | pGOMEA Pop.seed | pGOMEA Elit.seed | pGOMEA | pGOMEA Pop.seed | pGOMEA Elit.seed |
| $50 \times 5$ | 1.105 | 1.018 | 1.105 | 0.041 | 0.015 | 0.014 |
| $50 \times 10$ | 1.381 | 1.489 | 1.488 | 0.977 | 1.074 | 0.952 |
| $50 \times 20$ | 1.290 | 1.609 | 1.338 | 1.879 | 2.151 | 1.873 |
| $100 \times 5$ | 2.027 | 1.297 | 1.829 | 0.021 | 0.025 | 0.044 |
| $100 \times 10$ | 2.420 | 1.884 | 2.319 | 0.427 | 0.392 | 0.407 |
| $100 \times 20$ | 2.486 | 2.259 | 2.426 | 2.340 | 2.502 | 2.389 |

Table 6.1: Average MRPD values for single-solution seeded pGOMEA instances. Solutions found after 2,207,121 pGOMEA fitness evaluations.

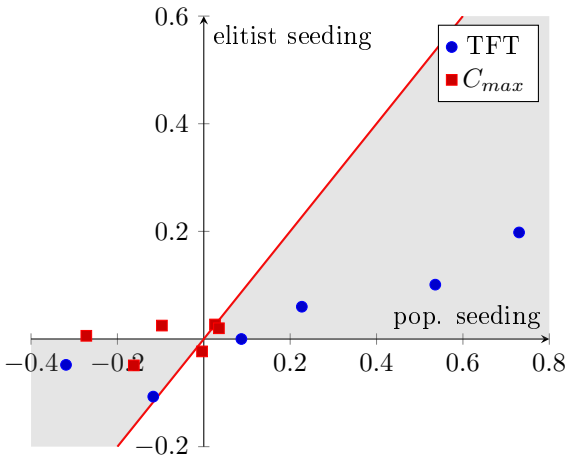


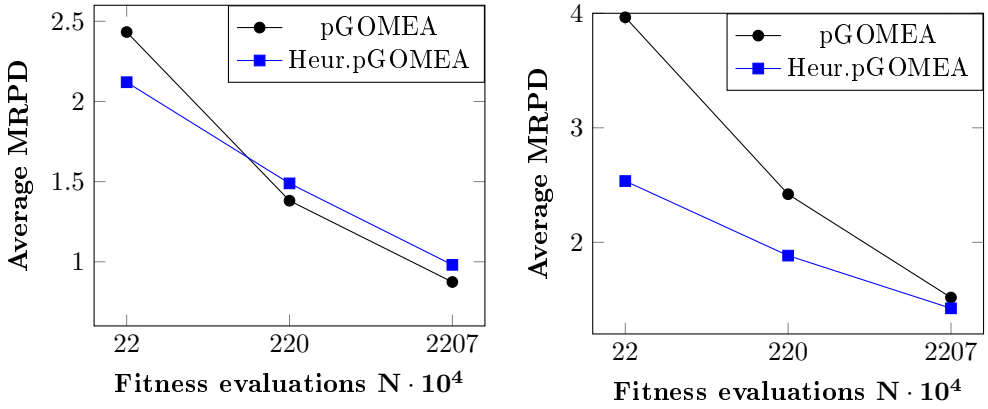

Figure 6.1: Elitist seeding versus population seeding: difference in average MRPD value with respect to pGOMEA for the TFT (blue) and $C_{max}$ (red) criterion

to elitist seeding. Every point represents the difference between the pGOMEA average MRPD value and the elitist/population seeded pGOMEA average MRPD value on one set of Taillard instances. Every plotted point in the grey area is a Taillard set where population seeding has a stronger positive or negative effect than elitist seeding. Around the origin, there are some $C_{max}$ results where elitist seeding has a positive effect and population seeding has a negative effect or vice versa. In general, this experiment shows that elitist seeding cannot be seen as better or worse than population seeding, it only has smaller effect than single solution population seeding. Therefore elitist seeding has no benefits over single-solution population seeding.

### 6.3.2 Single-solution population seeding: Quality and fitness evaluations

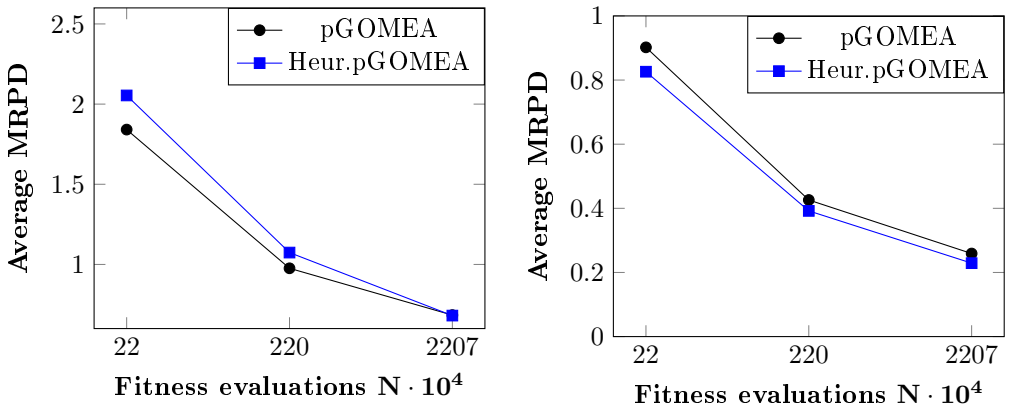
As we have seen from the results in Table 6.1, population seeding does sometimes improve solutions and sometimes it does not. In order to further identify the behavior of pGOMEA when seeded with a single solution, we look into more detail to the results of pGOMEA on the $(J \times 10)$ Taillard instances. For these instances we look at the average MRPD values with respect to the amount of fitness evaluations used by pGOMEA, the results of this experiment are shown in Figures 6.2 and 6.3 for the TFT and $C_{max}$ criterion respectively.

The results show that the effect of single solution population seeding is present both with a few and a lot of fitness evaluations by pGOMEA. The results clearly show that the effect is highest when pGOMEA uses little fitness evaluations. In Figure 6.2a It can be clearly seen that seeded pGOMEA performs better when a few evaluations are considered, though it performs worse when more evaluations are used. More fitness evaluations for pGOMEA result in larger populations, reducing the positive or negative dominating effect of a single seed in that population. Consequently, these pGOMEA instances might benefit from multiple seeds in the population, as diversity is ensured by the large population. Given the results in this experiment, we cannot state




(a) $J = 50$ (b) $J = 100$

Figure 6.2: Solution quality of pGOMEA and single-solution seeding using the TFT criterion.




(a) $J = 50$ (b) $J = 100$

Figure 6.3: Solution quality of pGOMEA and single-solution seeding using the $C_{max}$ criterion.

that single solution seeding is always or never beneficial for pGOMEA. We can only conclude that single-solution seeding has the advantage of guiding the population and the disadvantage of possibly misguiding the population.

### 6.3.3 Multi-solution population seeding: Fixed amount of seeds

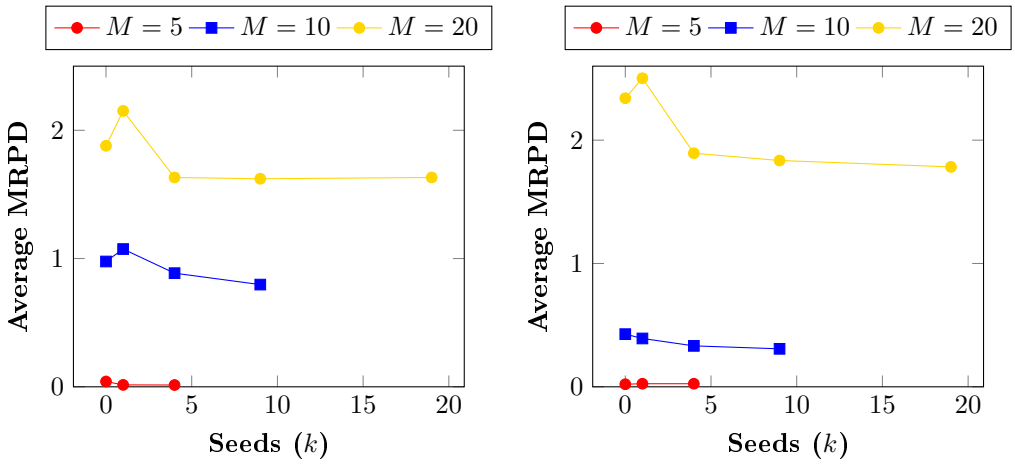
Since single-solution seeding either shows little effect (elitist seeding) or unpredictable results (population seeding), these approaches cannot easily be incorporated to improve pGOMEA. In order to further analyze the effect of seeding, we will look at multi-solution population seeding. Using multiple seeds in pGOMEA should take the population sizing scheme into account. This can be done in several ways. The first approach treats every new population the same, meaning that every population containts seeded solutions. The second approach takes the population size into account and every population starts with $x\%$ of seeded solutions. This latter approach does not easily scale in population size, since constructive heuristics generate a fixed amount of seeds.

Here we consider pGOMEA with a fixed amount of seeds per population. For $C_{max}$ we use the CDS heuristic, to generate solutions, this results in at most $M - 1$ seeds. For the TFT criterion we use the LR heuristic, resulting in at most $J$ seeds. We test fixed seeding on the six Taillard sets as used in the previous experiment. Since we want to take both constructive maximums into account, we use $k \in \{4, 9, 19, 30, 50, 75, 100\}$ seeds per population where possible.

Using a fixed amount of seeds in every generation, we find that multi-solution seeding is very beneficial when pGOMEA uses the standard amount of fitness evaluations. Figures 6.4 and 6.5

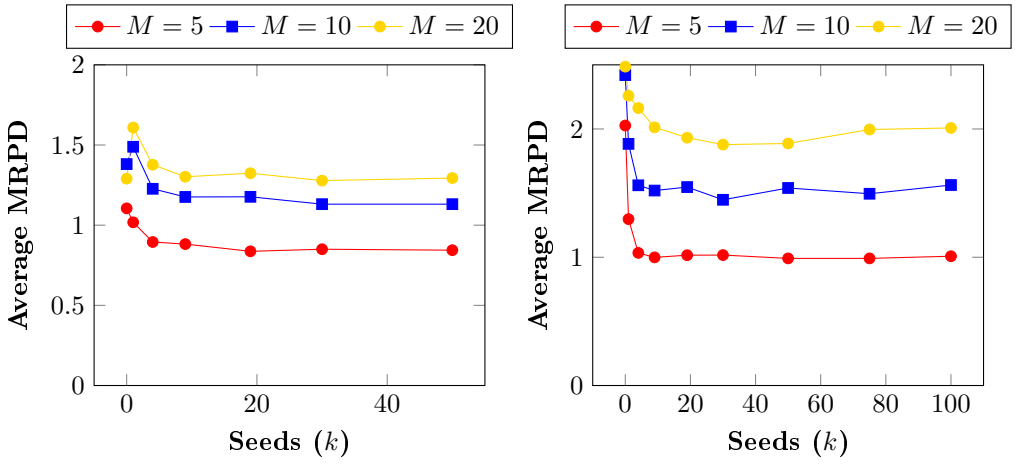
show how the average MRPD values decline for the TFT and $C_{max}$ criterion, when more than one seed is given. On virtually every tested set of Taillard instances, pGOMEA gave much better results when only four seeds were added. When the results of pGOMEA were already close to optimal, seeding does not have any effect however. We also observe that for the $(50 \times 20)$ instance, seeding does not improve on pGOMEA.//

Secondly, we observe a difference between single-solution seeding and multi-solution seeding. Whereas single-solution seeding can decrease the solution quality, multi-solution seeding does not suffer from this problem. This behavior is shown as a peek in Figures 6.4 and 6.5. A possible explanation for this observation is the use of different constructive heuristics in single-solution and multi-solution population seeding. For the TFT criterion this is a possible explanation, since the LR heuristic is considered better than the RZ heuristic. For the $C_{max}$ criterion, multi-solution seeding is done using the CDS heuristic, which generally performs worse than the NEH heuristic used for single-solution seeding. Therefore, we cannot conclude that the quality of the seeds is the reason why multi-solution seeding improves pGOMEA while single-solution seeding does not. Instead the amount of seeds are causing the difference in quality, which can also be seen in the ongoing decline after $k = 4$.




(a) $J = 50$ (b) $J = 100$

Figure 6.4: Multi-solution seeding for $C_{max}$: pGOMEA ($k = 0$) and single-solution population seeding ($k = 1$) versus multi-solution seeding.

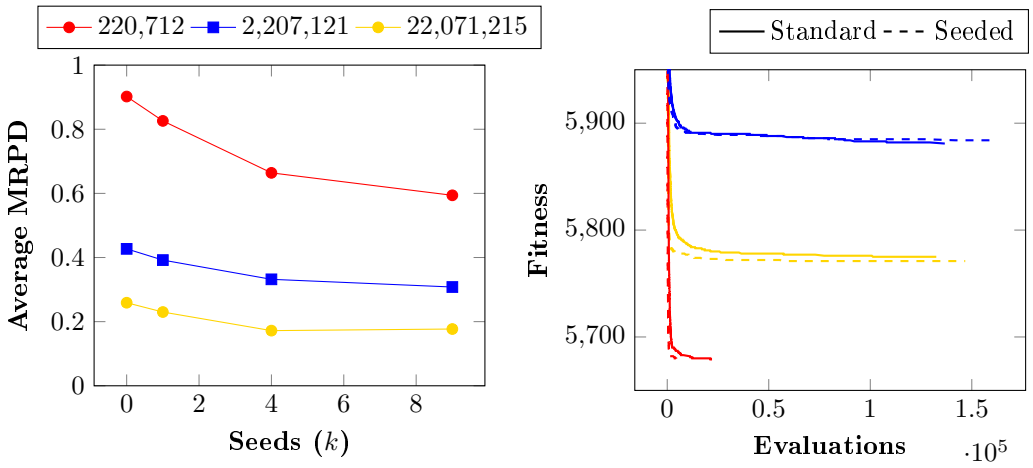



(a) $J = 50$ (b) $J = 100$

Figure 6.5: Multi-solution seeding for TFT: pGOMEA ($k = 0$) and single-solution population seeding ($k = 1$) versus multi-solution seeding.

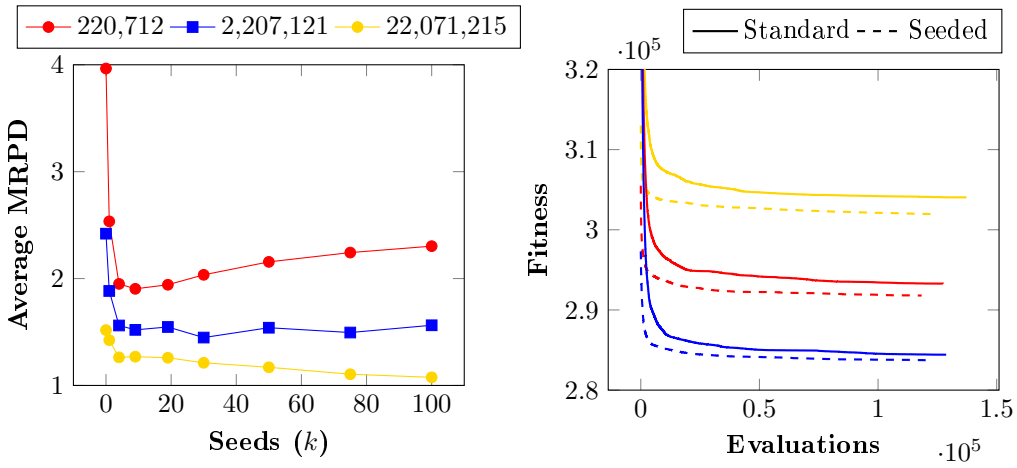
Thirdly, one can see that the solution quality does not keep increasing when more seeds are added. With more than 10 seeds, the solution quality does not improve very much for the $C_{max}$ criterion. For the TFT criterion, 20 seeds are sufficient to improve pGOMEA. Though adding more seeds does not improve pGOMEA, there is no trend observed which suggests that too much seeds lead to bad solutions. The reason why more seeds do not improve more possibly lies in the way seeds are generated. More seeds generated by the same constructive heuristic contain the same type of structure, thus no new information can be learned by pGOMEA. Another reason for this behavior could lie in the generation where the elitist solution is found. If this generation only contains a few members, only few constructive heuristics are sufficient to add structure in that population.

Now we have seen that seeding is beneficial when using a standard amount of fitness evaluations, we investigate the effect of this parameter on the effect of multi-solution seeding. Therefore, an extra experiment has been conducted, we take the first three $100 \times 10$ Taillard instances and observe the effect of multi-solution seeding with respect to the amount of fitness evaluations. The results of the experiment are shown in Figures 6.6 and 6.7 for the $C_{max}$ and TFT criterion.




(a) $(100 \times 10)$ (b) Scaling behavior over time

Figure 6.6: Scalability of multi-solution seeding for $C_{max}$: Seeds vs. Quality for different times (a) and Evaluations vs. Quality with maximum seeds (b)




(a) $(100 \times 10)$ (b) Scaling behavior over time

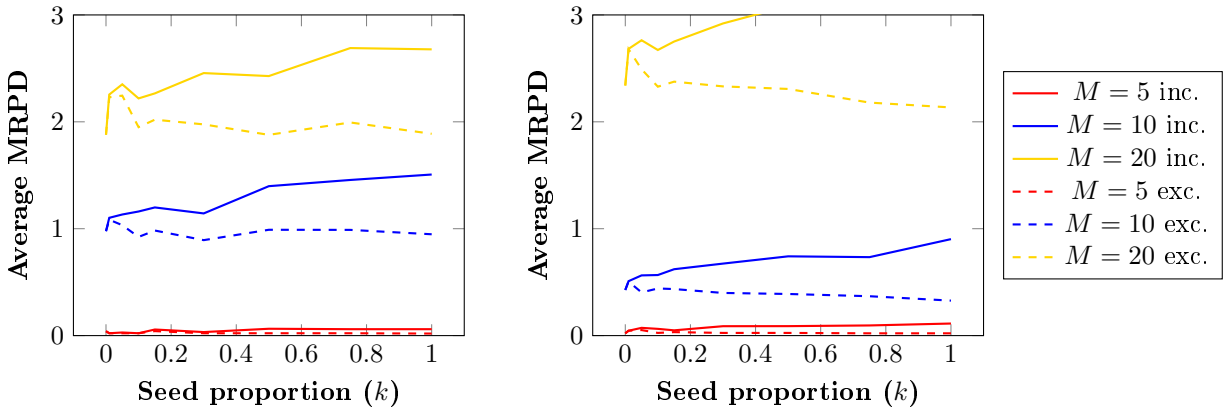
Figure 6.7: Scalability of multi-solution seeding for TFT: Seeds vs. Quality for different times (a) and Evaluations vs. Quality with maximum seeds (b).

First of all, the results show that multi-solution seeding is still useful when a lot of fitness evaluations are used. Multi-solution seeding is therefore a very promising addition to pGOMEA. We observe however that more seeds do not always give better results. This can best be seen in Figure 6.7a. When only a few fitness evaluations are used by pGOMEA, adding a lot of seeds is reducing the solution quality. When a lot of fitness evaluations are used by pGOMEA, adding seeds is beneficial. This can be explained using two properties of the pGOMEA implementation. First of all, pGOMEA uses a population sizing scheme, thus the result of pGOMEA with 220,712 evaluations is from a population with only a few seeds ($k' : k' < k$). In our implementation, this are not necessarily the top-$k'$ seeds, instead this are $k'$ random solutions from the top-$k$ seeds. Therefore, when more seeds are added to pGOMEA, these are worse seeds that replace good seeds instead of being added to them. When more fitness evaluations are used, the best solution comes from a large population where there is no replacement of seeds, adding seeds results in more seeds in that population. Therefore, pGOMEA gets better when more seeds are generated and a lot of fitness evaluations are used. This explanation is in accordance with the observation that $k' \in \{8, 16\}$ for a few evaluations, $k' \in \{32, 64, 128\}$ for a standard amount of evaluations and $k' > 100$ for many evaluations.

### 6.3.4 Multi-solution population seeding: Proportionate seeding and improvement heuristics

In order to further identify the behavior of multi-solution seeding in pGOMEA, we do a second experiment using multi-solution seeding. Here, we construct an amount of seeds proportionate to the population sizes. This is not possible using constructive heuristics, as they are limited in the amount of seeds they generate. Therefore we use the seed the population with the results of an improvement heuristic (insertion) over set a random solutions. This means that the seeds are less correlated than those generated by a constructive heuristic. However, we should keep in mind that standard improvement heuristics need a lot of fitness evaluations, therefore we report the result when fitness evaluations of improvement heuristics are ignored as well as when they are taken into account in the computation budget.

Figure 6.8 shows that for the $C_{max}$ criterion, pGOMEA hardly improves when seeded. The seeds generated by the improvement heuristic seem to be ineffective to improve pGOMEA. This can be caused by the improvement heuristic being stuck on a plateau, since the constructive heuristic for the $C_{max}$ criterion suffers less from this problem. Taking into account that the improvement heuristic takes computational budget, we can conclude that this form of seeding does not improve pGOMEA for the $C_{max}$ criterion.



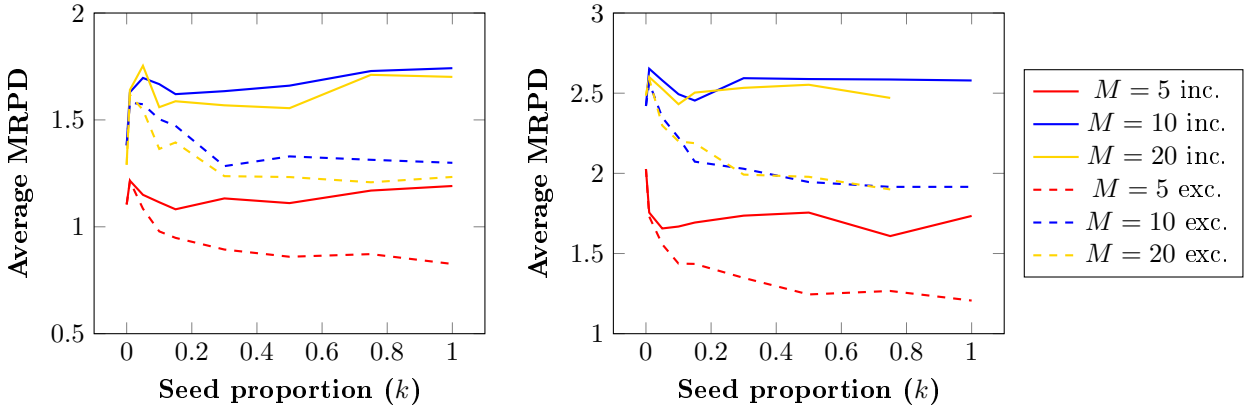

(a) 50 Jobs

(b) 100 Jobs

Figure 6.8: Population-proportionate multi-solution seeding for $C_{max}$: 2207121 fitness evaluations, including and excluding LS fitness evaluations.

Figure 6.9 shows how pGOMEA for the TFT criterion improves when seeded. The more seeds are used, the better pGOMEA performs. However, when we include the fitness evaluations of

the heuristic in the computational budget, this is not true anymore. Sometimes seeding does not improve pGOMEA. When it does, the optimal seed proportion is always between 0.0 and 0.2. A small amount of seeds will dominate the population, leading to a lack of diversity. Too much seeds will take too much fitness evaluations. Therefore, seeding using the computational expensive simple improvement heuristics needs careful tuning for each problem. Though constructive heuristics do not allow infinitely many seeds, they are computationally less expensive and do not need such careful tuning. Therefore, constructive heuristics can be considered to be better for seeding in pGOMEA.




(a) 50 Jobs (b) 100 Jobs

Figure 6.9: Population-proportionate multi-solution seeding for TFT: 2207121 fitness evaluations, including and excluding LS fitness evaluations.

## 6.4 Conclusions

After experimenting with different types of solution seeding, we can draw multiple conclusions about its performance:

- **Single-solution population seeding sometimes improves pGOMEA, while at other times it reduces the quality of pGOMEA solutions.**
  Since single-solution seeding uses only one seed, this seed can either lead or mislead the population towards the structure in the seed. If the structure of the seed has a lot in common with the structure of the optimal solution, the seed will help the population. If the structure of the seed is not present in the optimal solution, then the recombination process is misled and the population converges to a non-optimal solution.

- **The positive effect of single-solution population seeding is less visible when pGOMEA uses more fitness evaluations.**
  When pGOMEA uses a more fitness evaluations, the best known solution comes from a larger population, meaning that the effect of a single seeded solution is very small. As the amount of fitness evaluations by pGOMEA increases, the effect of single-solution population seeding decreases. Both the good and bad structure is used in recombination and the diversity in the large populations will add exploration behavior in pGOMEA.

- **Elitist-seeding behaves as a weak variant of single-solution population seeding**
  As the elitist solution is less often used for recombination than other solutions, using elitist-seeding has a smaller effect than single-solution population seeding. The results show that the improvement or decline in solution quality is correlated for both forms of seeding. The height of this improvement or decline is in general lower for elitist-seeding than for single-solution seeding.

- **Multi-solution seeding improves on pGOMEA, it does not suffer from the problems single-solution seeding has.**

When more than one solution is used as a seed, pGOMEA can be improved. This improvement is already present when only four to twenty seeds are added to the population. Single-solution seeding has a big bias towards the single seed. Multi-solution seeding has multiple biases, resulting in a form of diversity. The different biases are building-blocks in the seeds, which will now be donated into random solutions. The combination of these different building blocks allows pGOMEA to explore the search space globally. When only one solution is seeded, this exploration is only determined by the seed and therefore the search space is limited too much.

- **The more seeds pGOMEA has, the better pGOMEA performs.**
  Every seed gives more information to pGOMEA, leading to better results. One should however take into account that the generation of the seed needs computational resources. Therefore, a balance should be found in the amount of seeds. For seed generation using improvement heuristics, this balance is often hard to find and then only gives a small improvement in fitness value.

In general, we can conclude that multi-solution seeding works best for pGOMEA. However one should be careful in choosing the heuristic that generates these solutions. A slow seeder will take too much time, leading to a decrease in quality of pGOMEA.

# Chapter 7

# Hybridizing pGOMEA: Experimental Study

In previous chapters we have seen that seeding pGOMEA with good solutions results in a great improvement of the quality of solutions and that an improvement heuristic can not always improve the elitist solution of pGOMEA.Therefore the question arises whether improving solutions during the iterative phase of pGOMEA is effective. Hybridizing genetic algorithms with local search (improvement heuristics) is one of the most common forms of incorporating domain knowledge, which we investigate in this chapter. In order to identify how improvement heuristics can help pGOMEA in its iterative phase, we research the question

*What are the options and effects of incorporating an improvement heuristic/local searcher in the iterative phase of pGOMEA?*

Hybrid genetic algorithms usually have small population sizes, since they are less prone to genetic drift. pGOMEA doesn't have a fixed population size, but uses a population sizing scheme in order to overcome the problem of finding a good population size. Therefore, we first investigate the need for such a population sizing scheme when hybridizing pGOMEA. Here we compare three forms of hybridizing pGOMEA based on their convergence.

Using the best form of hybridization, we investigate the hybridization from a BBO perspective using the simple swap and insertion improvement heuristics. These heuristic do not assume anything about the problem domain, thus they are also applicable for other permutation problems. Secondly, we investigate the (additional) benefits of hybridization when problem-specific local search is used (e.g. NEH improvement heuristic). Here, the computational budget is given in milliseconds, as improvement heuristics use partial evaluations. These results do not easily translate to other permutation problems, as they possibly don't have such local searchers.

For all experiments, the general setup is given in Section 7.1. Results of the experiments are given in Section 7.2. This chapter ends in Section 7.3 with a summary of all conclusions drawn from the experiments. As this chapter uses knowledge derived from Section 4.1.3 and improvement heuristics from 3.3 one is advised to take note of the contents of these sections before reading this chapter.

## 7.1 Experimental setup

The experiments that are performed in this part of the research each have their own setup. The experiments all fall in one or more of the following categories:

- **Black-box optimization using simple improvement heuristics**
  In order to find out how hybrid pGOMEA performs with respect to a BBO approach, we conduct some experiments, where the two simple improvement heuristics are used. The computational budget of pGOMEA is counted in terms of the amount of fitness evaluations that are used in pGOMEA and its improvement heuristic. Since we do not assume any knowledge of the PFSP, except that it is an permutation problem, we can possibly draw more general conclusions about hybridizing pGOMEA. Our insertion heuristic is implemented to try only insertions in a range of $M$ jobs of the current job position. The swap heuristic takes all possible swaps into account.

- **Domain specific local search (Non-BBO search)**
  As the PFSP with the $C_{max}$ criterion has the NEH improvement heuristic, which is able to find the best insertion of a job in a relatively short time, we can use this improvement heuristic to hybridize pGOMEA for the PFSP in a more effective way. Likewise, we can use a form of cut-and-repair heuristic in order to steer the search for the PFSP with the TFT criterion. Since such heuristics are only applicable for a specific permutation problem, the results of this experiment are not related to pGOMEA on other permutation problems. For the computational budget in these experiments, we use the computation time in milliseconds. The computation time is of course dependent on the implementation of the local searchers. Therefore, we decided to limit our run-time optimizations to the level of big-O notation. Computation times used are given in Table 7.1. When the quality of the algorithm is measured with respect to the computational budget, the used running times are a factor 10 higher or lower than reported here.

| Instance | Time | Instance | Time |
|---|---|---|---|
| $50 \times 5$ | $10.000ms$ | $100 \times 5$ | $17.500ms$ |
| $50 \times 10$ | $12.500ms$ | $100 \times 10$ | $20.000ms$ |
| $50 \times 20$ | $15.000ms$ | $100 \times 20$ | $22.500ms$ |

Table 7.1: Computational budget for non-BBO experiments

- **Depth-limited local search**
  As mentioned in Chapter 4, the trade-off between local exploration and global exploration should be carefully defined. A local search algorithm that is continuously exhaustively performed will take a lot of time and fitness evaluations, while it improves good solutions only slightly. Depth-limited local search only applies a limited amount ($k$) of insertions or swaps, before terminating the local search. Since changes in good solutions are found after more fitness evaluations, this limits the amount of time used in local search, while improvements are still found. For these experiments, we use local search depths of 1, 3, 6, 10 and 15.

- **Limited probability of local search**
  Another way of balancing local search and global search is by limiting local search to a few individuals. Each individual has a probability of undergoing local search. In our experiments, we use very low probabilities 0.001, 0.005, 0.01 and 0.05, and probabilities from 0.1 up to 1.0 with steps of 0.1.

For each type of experiment, we use the pGOMEA settings and quality measures as described in Section 5.1.

## 7.2 Results

### 7.2.1 Effects of hybridization

Hybridized GAs often use a fixed population size in the range 20-100. Larger population sizes are often unnecessary as diversity is introduced by the local search operator. When hybridizing pGOMEA with local search, we therefore consider using a fixed population size instead of a population sizing scheme. In this experiment, we test different population sizes and their effect on pGOMEA using the NEH-insertion heuristic after the GOM phase. We compare three configurations. The configurations differ in the moments when local search is performed. Table 7.2 shows for each configuration whether local search is performed or not on solutions that are (not) changed in the GOM phase of pGOMEA. Here we compare the different configurations on instances with sizes $(50 \times 5)$, $(50 \times 10)$ and $(50 \times 20)$. The results of this experiment are shown in Figures 7.1, 7.2 and 7.3. The first set of instances is easily solved by pGOMEA, while the others are harder for pGOMEA. As configuration 1 is obviously the best when performing an exhaustive local search with probability 1.0, we limit the probability of local search in this experiment. Local search is performed with a probability of 0.1, later experiments show that this is an acceptable value.

| | Local search if: | |
|---|---|---|
| | Changed | Not changed |
| Configuration 1 | × | |
| Configuration 2 | | × |
| Configuration 3 | × | × |

Table 7.2: Used pGOMEA local search configurations

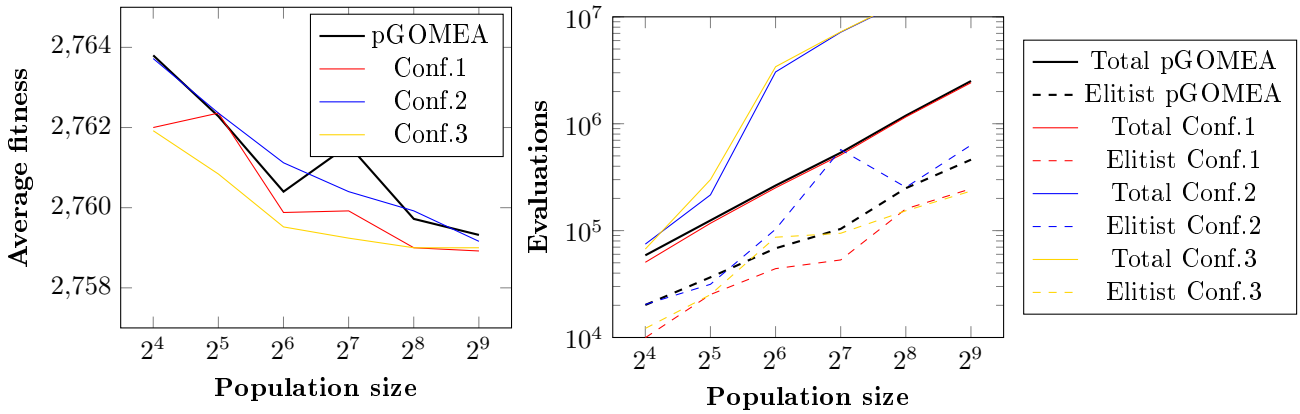
For each of the configurations and population sizes we report the time to convergence (in amount of evaluations, counting one per insertion), the time that the elitist solution was hit and the resulting fitness of the found solutions. Each result is an average over five runs over five instances with the given size. We perform experiments using pGOMEA with the $C_{max}$ criterion and the NEH-insertion heuristic. We have limited the amount of generations that pGOMEA runs to 101 generations. Results are compared with the convergence behavior of standard pGOMEA (the thick black lines).

For the first configuration we see that hybridized pGOMEA results in about the same quality of solutions as pGOMEA. Only for the small instances, this difference is much clearer. In terms of convergence, this configuration takes slightly less fitness evaluations until convergence. The elitist solution is however found much faster than without hybridization. Using a population sizing scheme can be successful in this case. Due to hybridization, less fitness evaluations are spend per population. Therefore pGOMEA spends more time in larger populations and these populations give better results as can be seen in the left graphs.

The second configuration performs equally well as standard pGOMEA. For the middle-sized instances, it is even better than standard pGOMEA. In terms of fitness evaluations, the convergence of a population takes much more time, while the elitist solution is hit at approximately the same time as pGOMEA does.

The third configuration combines the effects of the first two configurations: the quality of solution still does not change much (though results are more consistent). However, more fitness evaluations are needed to reach convergence, like in configuration 2. The elitist hitting time is similar to the times of configuration 1, as less evaluations are needed than for standard pGOMEA.

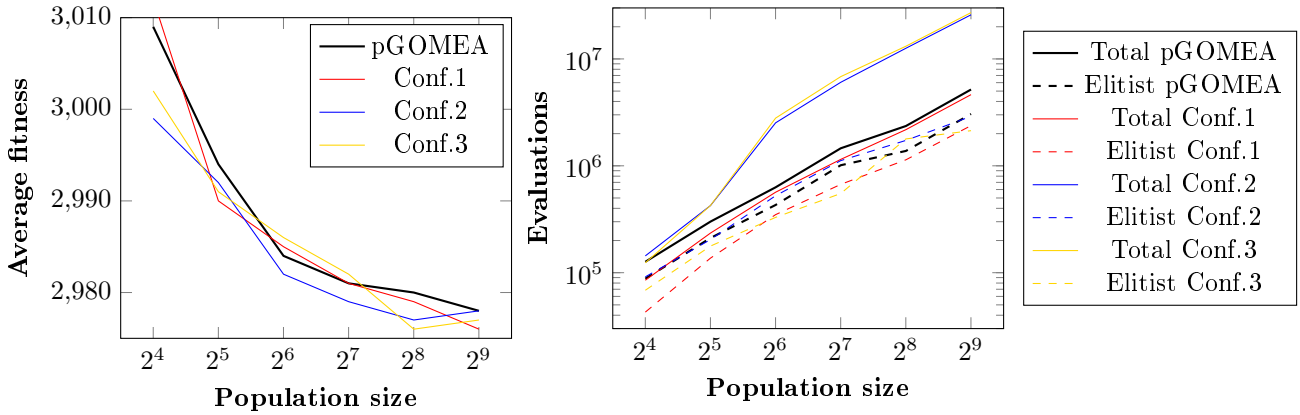
None of the researched population sizes and local search configurations show however that a small, fixed population size is sufficient for pGOMEA. We can therefore conclude that a population sizing scheme is necessary for (hybridized) pGOMEA when the optimal population size for a given computational budget has not been experimentally determined. Since configuration 1 gives the fastest convergence while retaining the solution quality, we use this configuration for our next hybridization experiments.




(a) Fitness results

(b) Convergence results

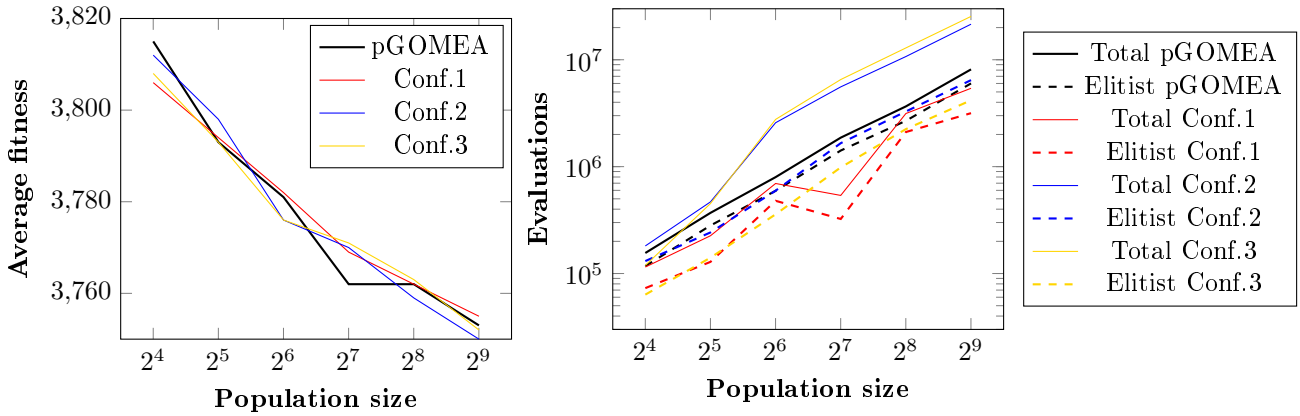
Figure 7.1: Convergence, elitist hit time and population size results for $(50 \times 5)$ using $C_{max}$.




(a) Fitness results

(b) Convergence results

Figure 7.2: Convergence, elitist hit time and population size results for $(50 \times 10)$ using $C_{max}$.



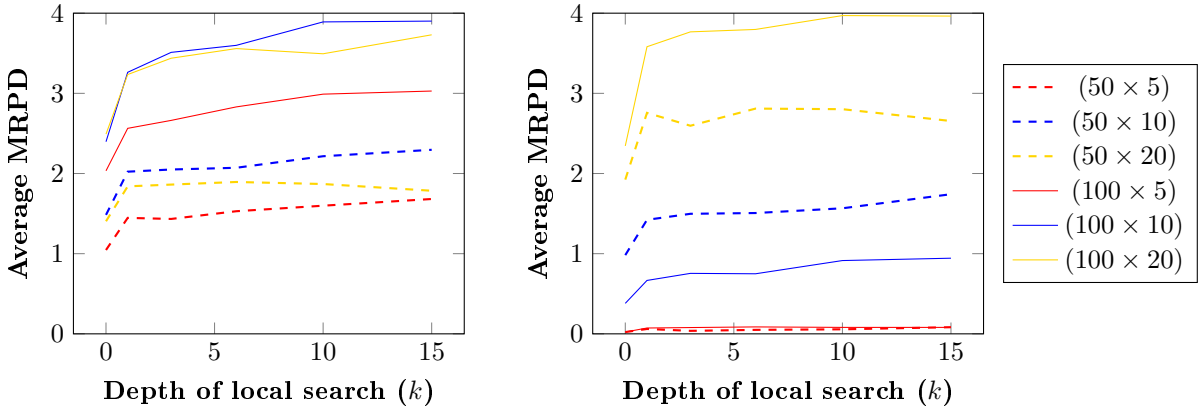

(a) Fitness results

(b) Convergence results

Figure 7.3: Convergence, elitist hit time and population size results for $(50 \times 20)$ using $C_{max}$.

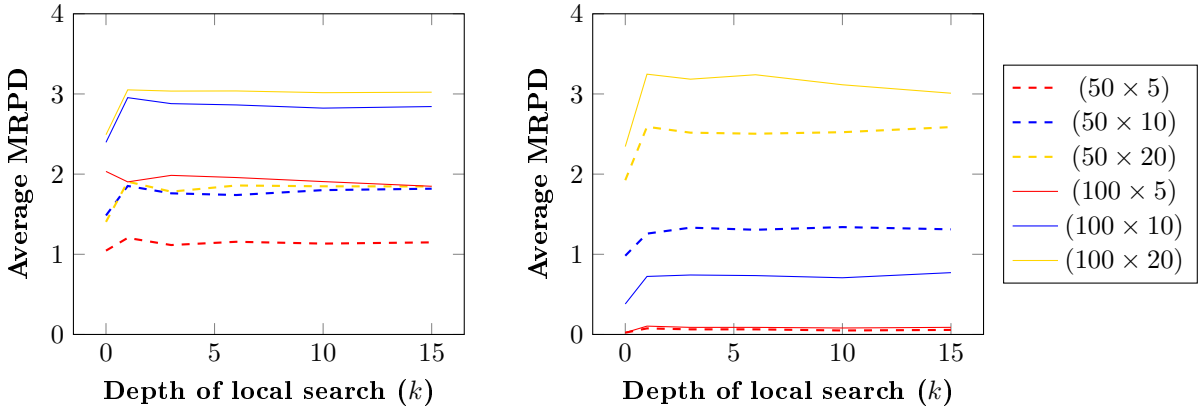
### 7.2.2 Depth limited local search using a BBO perspective

For finding a good way of hybridizing pGOMEA with simple improvement heuristics, we first identify the behavior of pGOMEA with different depths of local search. As soon as $k$ improvements have been made by the improvement heuristic, the improvement heuristic stops. Using this form of limited search, we can balance the computational budget of the GOM phase and the LS phase in pGOMEA. This creates a balance between global and local search. In Figures 7.4 and 7.5, the results of depth-limited local search are shown for the insertion and swap heuristic respectively.




(a) TFT (b) $C_{max}$

Figure 7.4: Hybrid pGOMEA quality with respect to the depth of an insertion heuristic. A total of 2,207,121 fitness evaluations is used by hybrid pGOMEA.



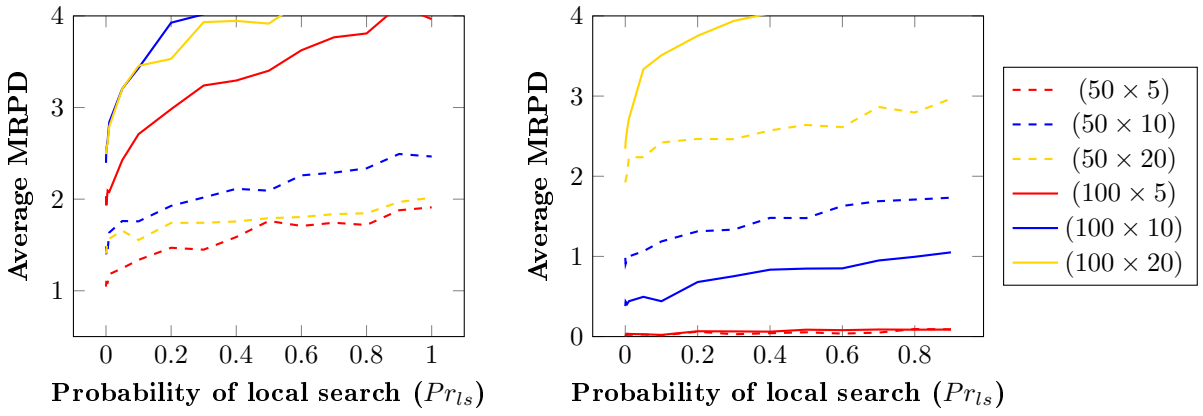

(a) TFT (b) $C_{max}$

Figure 7.5: Hybrid pGOMEA quality with respect to the depth of a swap heuristic. A total of 2,207,121 fitness evaluations is used by hybrid pGOMEA.

The graphs clearly shows how the average MRPD values rise when local search is added. Even when a few improvements are made, MRPD values are significantly higher than without using local search. For every size of problem instances, we see how depth-limited local search is not improving pGOMEA. Only in one case hybridization is effective. For the PFSP with the TFT criterion, hybridizing pGOMEA with the swap heuristic is effective if the instance contains a few machines. We can also see that for both the TFT and $C_{max}$ criterion swapping is most effective (least useless).

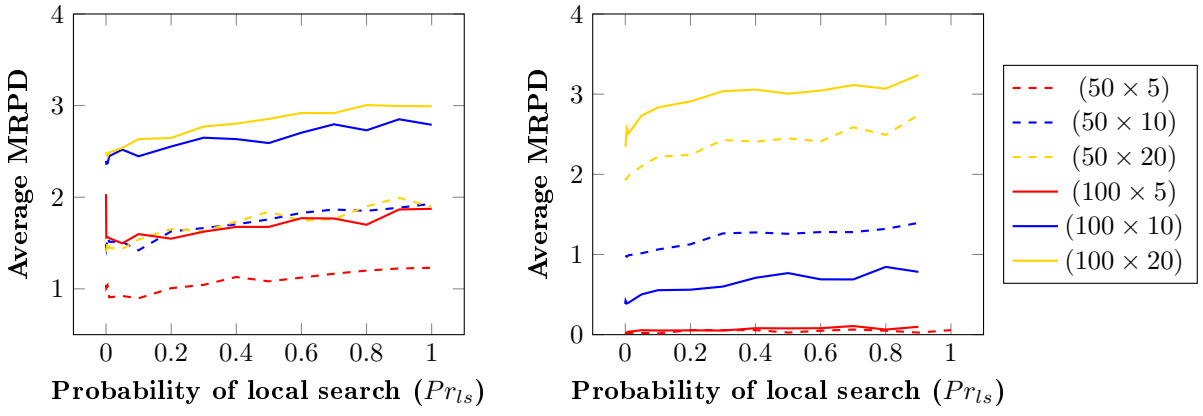
### 7.2.3 Probability of local search using a BBO perspective

As depth-limiting local search does not simply improve pGOMEA, we try hybridizing pGOMEA with heuristics using a limited probability of local search. The overall influence of the probability value is researched, by taking 11 equidistant values in $[0, 1]$. In contrast to depth-limited local search, probabilities can be infinitely close to zero, therefore, we take more samples for probabilities close to zero. Figures 7.6 and 7.7 show the probability of local search influences the performance of pGOMEA for the insertion and swap heuristic respectively.




(a) TFT (b) $C_{max}$

Figure 7.6: Hybrid pGOMEA quality with respect to the probability of using an insertion heuristic. A total of 2,207,121 fitness evaluations is used by hybrid pGOMEA.




(a) TFT (b) $C_{max}$

Figure 7.7: Hybrid pGOMEA quality with respect to the probability of using a swap heuristic. A total of 2,207,121 fitness evaluations is used by hybrid pGOMEA.

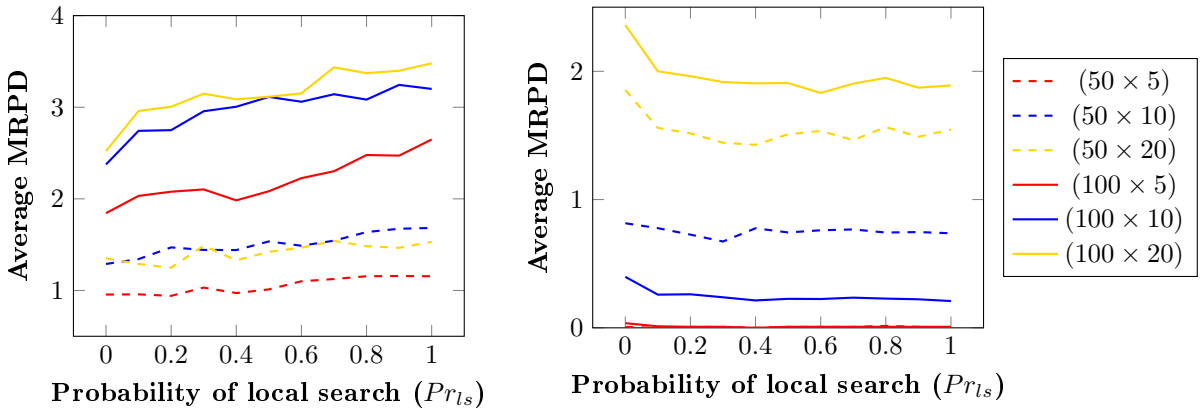
Adding heuristics to pGOMEA is often decreasing the quality of the solutions. For the insertion heuristic and for the $C_{max}$ criterion, local search is never effective. Even with a 0.1% probability of local search, the average MRPD values are higher than without using local search. Taking a higher probability of local search results in worse solutions. The average MRPD values at $pr_{ls} = 1.0$ are close to those of depth limited local search with $k \geq 5$. This indicates that the depth limiting local search with values $k \geq 5$ often results in a local optimum.

For the TFT criterion we find again that for instances with a few machines, hybridization can be effective. Local search is then best performed with probabilities below 10%. When using very low probabilities, hybridization can also be effective when more machines are used, though this difference is not significant.

As these results are seen from a BBO perspective, we can conclude that no problem- or instance-independent results are found. Therefore, when hybridizing pGOMEA one should view it from a non-BBO perspective.

### 7.2.4 Hybridizing pGOMEA using advanced local search

Though using naive heuristics does not easily improve the quality of pGOMEA, hybridization using advanced local search operators can possibly improve pGOMEA. These advanced local search methods use domain knowledge in order to be more efficient. In this experiment, we will research the pGOMEA quality when hybridized with an advanced local searcher. As improvement heuristics we use the NEH improvement heuristic (for $C_{max}$) and the Cut-and-Repair improvement heuristic (for TFT).




(a) TFT

(b) $C_{max}$

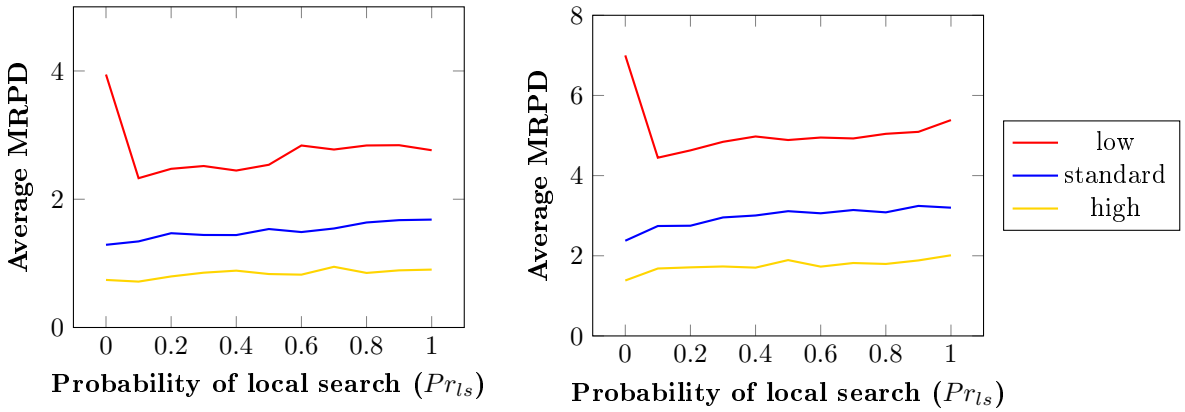
Figure 7.8: Hybrid pGOMEA quality using advanced local searchers with respect to the probability of local search. Computational budget is given in Table 7.1.

First, we examine the quality of pGOMEA with respect to the probability of local search, these results are shown in Figure 7.8 for six instance sizes. Here, we observe that the Cut-and-Repair heuristic has insufficient domain knowledge to improve pGOMEA. For each of the given probabilities, the results become worse when pGOMEA is hybridized. The higher the probability of local search, the worse the results of pGOMEA. We can therefore conclude that the Cut-and-Repair heuristic does not sufficiently exploit domain knowledge to improve pGOMEA. The swap heuristic probably performs better than this Cut-and-Repair heuristic, as it has shown to have some effect in BBO context, while spending less time than used in the GOM-phase of pGOMEA.

For the NEH-heuristic, we see the opposite as for the Cut-and-Repair heuristic. pGOMEA for the $C_{max}$ criterion is improved the most when using a high probability of local search. Figure 7.8b shows that with a low probability, pGOMEA is already improved. pGOMEA keeps improving up to a probability of 0.6, after which pGOMEA performs stable or slightly (non-significantly) worse, depending on the instance.

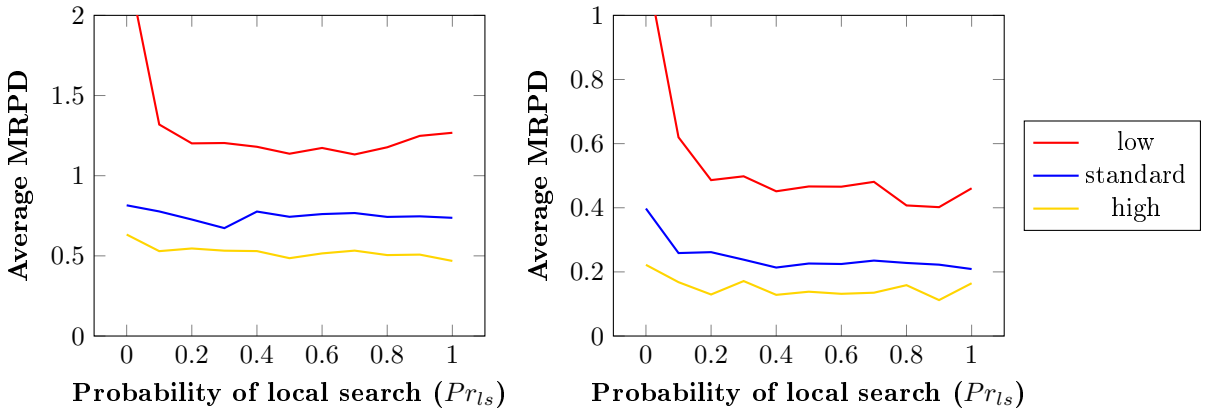
Given this potential of the NEH-heuristic to improve pGOMEA, we further examine its quality by experimenting with lower and higher running times for pGOMEA. For the running times, we use the values a factor 10 higher or lower than those in Table 7.1. We also examine the behavior of the Cut-and-Repair heuristic for the TFT criterion with lower and higher running times. Results of these experiments are shown in Figure 7.9 for the TFT criterion and Figure 7.10 for the $C_{max}$ criterion.

These results show that the effects of local search are still present when using more fitness evaluations. Using more local search is better for the $C_{max}$ criterion and worse for the TFT criterion. For the TFT criterion we also observe that with a small computational budget, pGOMEA performs better with local search. With such a small budget, one pass of local search is probably




(a) $J = 50$ (b) $J = 100$

Figure 7.9: TFT Hybrid pGOMEA quality using advanced local searchers with respect to the probability of local search. Computational budget is given in Table 7.1.




(a) $J = 50$ (b) $J = 100$

Figure 7.10: $C_{max}$ Hybrid pGOMEA quality using advanced local searchers with respect to the probability of local search. Computational budget is given in Table 7.1.

better than doing recombination within small populations, for the larger computational budget, pGOMEA for the TFT criterion is not improved by adding local search.

## 7.3 Conclusions

- **Local search is best performed when pGOMEA has changed a solution.**
  The experiments using different configurations for local search have shown that local search is best performed if pGOMEA has changed a solution. This has a trivial reason when local search is exhaustively applied, but our results show that this configuration is also best for a low probability of local search. This configuration improves the time in which a good solution is found. If local search is (also) performed when pGOMEA has not changed a solution, the convergence of the populations decreases. Since equal-fitness solutions are often changed by local search, the population does not easily converge. For the population sizing scheme, this means that a lot of computational budget is wasted in populations that are not promising.

- **Using a small fixed population size is not sufficient for hybrid pGOMEA.**
  As shown in the population sizing experiment, larger populations produce better results, while using more fitness evaluations. The smaller populations sizes cannot benefit from a local searcher that adds diversity in the population. Therefore, the population sizing scheme should still be employed in pGOMEA. The local searcher does influence this population sizing scheme in two ways. A fast convergence results in a early termination of populations, leaving

more computational budget to larger populations. A fast detection of a good solution leads to better recombination in the forced improvement phase.

- **Hybridization of pGOMEA for the PFSP is not applicable in a BBO context.**
  Hybridization of pGOMEA in a BBO context uses a lot of fitness evaluations in the local searcher. Therefore, less fitness evaluations are left for the GOM phase of pGOMEA. As also shown in Chapter 5, the quality of the heuristic is dependent on the problem and instance to be solved. For the same reason we see that the hybridization of pGOMEA is only successful in some cases. From a BBO perspective one has no prior knowledge about the problem, therefore no local searcher can be designed that is guaranteed to improve pGOMEA.

- **Hybridization with advanced local searchers can improve pGOMEA.**
  Though from BBO perspective heuristics cannot improve pGOMEA, advanced heuristics can improve pGOMEA. The NEH-heuristic, which finds the best point of insertion for a variable fast, does not waste too much computational budget as it calculates multiple evaluations at once. When an NEH-insertion is performed with any probability higher than 0.0, pGOMEA is improved. This is also the case when considering a higher or lower computational budget than standard. Though the Cut-and-Repair heuristic also uses domain knowledge to find good solutions, this does not give a speedup fast enough not to waste computational budget from pGOMEA. This indicates that local searchers need a significant reduction in fitness evaluations in order to be efficient for hybridization of pGOMEA.

- **The less computational budget is given to hybrid pGOMEA, the more advantageous local search is.**
  pGOMEA using a low computational budget evaluates only some fast-converging, small populations. In these populations, pGOMEA cannot find good solutions, as exploration is limited and no good model can be estimated from the population. The low computational budget is then better spend by local search in case of a low computational budget.

# Chapter 8

# Informed model learning: Experimental Study

In Chapter 6 we have seen how pGOMEA can be improved when good solutions are seeded in the population. The results showed that adding more seeds result in better performance of pGOMEA. Single-solution seeding is not improving pGOMEA and seeding using simple improvement heuristics gives only a slight improvement of pGOMEA, when the seed proportion is carefully tuned. Therefore seeding is not a straightforward option when no constructive heuristics exist that generate multiple solutions. In this chapter we will introduce a new form of seeding pGOMEA. We use the hypothesis that one of the reasons for pGOMEAs improvement when seeded is that the linkage tree has useful information from the start of each population. In this chapter, we investigate this hypothesis in Section 8.1. After this hypothesis is research, we use this hypothesis to improve pGOMEA using dependency seeding as introduced in Section 8.2. This form of seeding is experimentally tested in Section 8.3 where we compare two forms of dependency seeding. The question that we try to answer in this chapter is:

*How can model-building be improved using domain knowledge?*

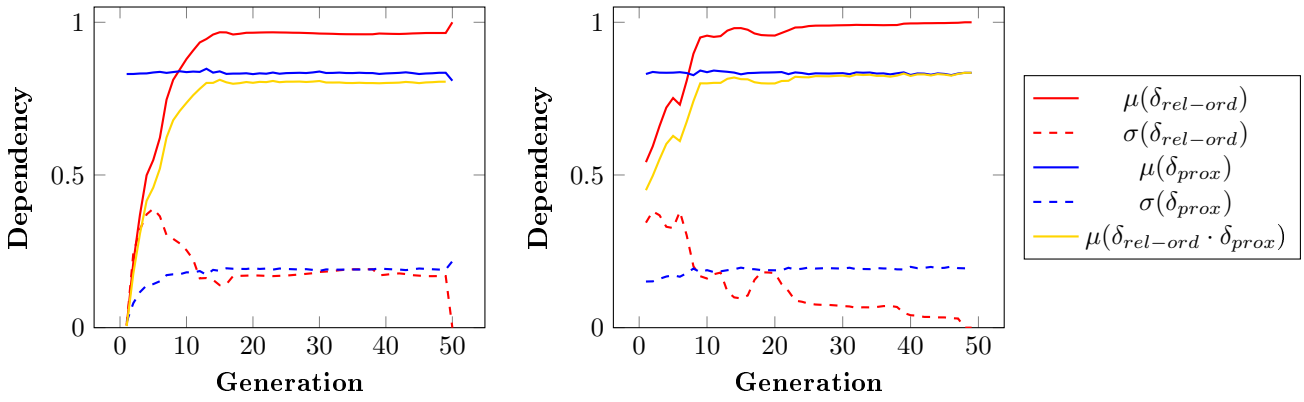
## 8.1 Seeding: effect on model building

The first generation of a population consist of random solutions. When the linkage tree is build, it does therefore not contain any domain knowledge. The successful random crossovers that are performed will create patterns in the population leading to more information in the linkage tree. It takes some time before the right dependencies are learned and one can see this time as a waste of computational budget. We assume that by seeding solutions in the population, pGOMEA is able to learn valid dependencies from the start of a population. With a few seeds, this new information can hardly be detected through the noise in the other solutions. With some more seeds, the linkage tree will already find good dependencies. This is a possible reason for the the rapid improvement of pGOMEA when only a few seeds are added. Adding more seeds does not significantly change the linkage tree.

In order to test these assumptions, we compare the linkage-behavior of pGOMEA with and without seeding. We track the behavior of pGOMEA on a $(100 \times 10)$ Taillard set. Here we use a fixed population of size 100 which is either fully seeded or not seeded at all. We measure both dependency values: relative ordering dependency ($\delta_{rel-ord}$) and proximity dependency ($\delta_{prox}$). For both dependency measures, we identify the mean ($\mu$) and the standard deviation ($\sigma$) over all dependencies between two variables. The results of this experiment for the TFT criterion are shown in Figure 8.1.

We will discuss the results for each property:

$\mu(\delta_{rel-ord})$: When pGOMEA starts with a random population, the relative ordering is initially 0. Because the variables are in random order, $Pr(r_{x_i} < r_{x_j}) \approx 0.5$. This means that the entropy is very close to one and the relative ordering dependency is thus very close to 0. When the




(a) Standard pGOMEA (b) Seeded pGOMEA

Figure 8.1: (seeded) pGOMEA dependency values: mean ($\mu$) and standard deviation ($\sigma$) of both dependency measures from initialization to convergence.

population is converged, this probability is either one or zero, leading to a dependency of 1. When pGOMEA is seeded, there are already stronger dependencies in the first generations. At other points curve of pGOMEA remains almost the same, though there seems to be a more gradient convergence to the dependency of 1, instead of a sudden step to full dependency after the NIS is reached.

$\sigma(\delta_{rel-ord})$: Initially, no differences exist in the relative ordering dependency, it is zero for every individual. The standard deviation of the relative ordering initially increases very fast, since some dependencies are obvious and learned very easily. When more dependencies are learned and the average dependency value hardly changes, the standard deviation stabilizes. At the end, the dependency is one for every individual, leading to a standard deviation of zero. When seeded, pGOMEA is again 5-10 generations ahead of the standard pGOMEA. A second observation is that the gradient convergence of the relative ordering leads to a slow decrease of the standard deviation instead of a sudden one.

$\mu(\delta_{prox})$: The average value for the proximity dependency is initially somewhat surprising. At any time, for both seeded and standard pGOMEA, the proximity dependency is $\frac{5}{6}$. However, this is a logical result of the way this dependency is computed and how random keys are assigned. Since re-encoding assigns random values from a uniform distribution $[0, 1]$ to the random keys, the expected squared distance between two random keys in an individual can be calculated by

$$\frac{1}{1}\int_0^1 \int_0^1 (x-y)^2 \, dx \, dy = \frac{1}{6}. \tag{8.1}$$

Sampling over $\frac{100 \cdot 99}{2}$ pairs in 100 individuals therefore results in an average dependency of $1 - \frac{1}{6} = \frac{5}{6}$. Between pairs, there can however be significant differences, which we see in the standard deviation.

$\sigma(\delta_{prox})$: For a random population, jobs are randomly placed in the individuals, the dependencies between jobs are all approximately $\frac{5}{6}$. For a seeded population, jobs have a tendency to be used either at the start or end of a schedule. Therefore, seeded pGOMEA has a high standard deviation in the initial population. Though this standard deviation increases, it only changes slightly. Again, seeded pGOMEA is 5-10 generations ahead of standard pGOMEA.

In general, we can see that seeded pGOMEA allows the linkage tree to learn dependencies from the start of the population. Standard pGOMEA has to figure these dependencies out using the random crossovers. Especially when a lot of variables are used, seeding should help pGOMEA to detect dependencies. With a lot of variables, it takes a lot of time to learn the right dependencies, giving seeded pGOMEA an advantage. We can therefore safely assume that the ability of pGOMEA to learn dependencies faster, is one reason why population-seeding is improving pGOMEA.

## 8.2 dependency seeding

We have seen that population-seeding has a positive effect on model building, which improves the exploting behavior of pGOMEA. If multi-solution seeding is not possible, we might still be able to improve pGOMEA by adding domain knowledge in the tree-learning phase. For this, we propose the following extension of pGOMEA's dependency measure (Equation 2.7):

$$\delta(i,j) = \delta(j,i) = \underline{w_{pop.}} \cdot \delta_{rel-ord}(i,j) \cdot \delta_{prox}(i,j) + \underline{w_{dom.} \cdot \delta_0(i,j)}. \tag{8.2}$$

Here, we add a dependency from domain knowledge ($\delta_0(i,j) \in [0,1]$) to the dependencies known from the population. In order to balance domain knowledge and population model building, $w_{pop.}$ gives a weight for the population knowledge and $w_{dom.}$ specifies how much domain knowledge is taken into account.

In practice, the choice of function $\delta_0(i,j)$ heavily depends on the type of problem that should be solved. For the PFSP problem, we present two function targeting the $C_{max}$ and TFT criterion and one function that can be generally applied to any problem for which a multi-solution constructive heuristic exists.

### 8.2.1 Indexing dependency

For the $C_{max}$ criterion, we have seen how Johnson's rule and the CDS and RA heuristic use the time spend on the first and later machines to order the jobs. Using this same concept, the Palmer heuristic assigns a slope value $S_i$ to each job $i$ The jobs are then scheduled ordered by slope value. We can exploit this value by the observation that jobs with a large difference in slope should be far away in the schedule. These jobs therefore have a low dependency in terms of proximity. We therefore define the Palmer dependency as

$$\delta_{Palm}(j,i) = \delta_{Palm}(i,j) = 1 - \left| \frac{S_i - S^{min}}{S^{max} - S^{min}} - \frac{S_j - S^{min}}{S^{max} - S^{min}} \right|, \tag{8.3}$$

where $S^{max}$ and $S^{min}$ are the maximum and minimum slope found. This dependency first scales the Palmer values to the range $[0,1]$ after which the inverted distance between the two scaled Palmer values is taken as the dependency value between the jobs.

For the TFT criterion, we have a similar indexing approach used in the RZ heuristic. The RZ heuristic calculates $k$ indices $T_{i,k}$ per job $i$. The most straightforward idea is to use $T_{i,0}$ as it uses information from all machines. We can now define the RZ dependency as

$$\delta_{RZ}(j,i) = \delta_{RZ}(i,j) = 1 - \left| \frac{T_{i,0} - T_0^{min}}{T_0^{max} - T_0^{min}} - \frac{T_{j,0} - T_0^{min}}{T_0^{max} - T_0^{min}} \right|. \tag{8.4}$$

where $T_0^{max}$ and $T_0^{min}$ are the maximum and minimum slope found. This dependency is almost similar to the Palmer dependency, except that it uses the RZ index and that it is thus more suitable for the TFT criterion.

### 8.2.2 Dependency over constructive population

A second way of adding dependencies is by using the relative ordering and proximity dependency over an amount of seeds generated by a constructive heuristic. For instance, the CDS heuristic can generate at most 19 schedules for the Taillard instances. These are possibly too few solutions to influence the linkage tree, especially in larger populations. One might therefore (additionally) compute the relative ordering and proximity dependency over the seeds of CDS. These dependency values can then be used as domain knowledge in the extended dependency measure.

This form of of dependency seeding can possibly be worthwhile when both diversity as well as a good linkage structure is wanted. Now the random population ensures diversity, while the dependencies over the seed-population are used in creating a good linkage structure.

### 8.2.3 Determining the weight

The optimal value for the weights $w_{pop.}$ and $w_{dom.}$ can experimentally be determined. For perfect domain knowledge $w_{pop.} = 0$ and without domain knowledge we have $w_{dom.} = 0$. In our experiments, we make an initial guess for the value of $w_{dom.}$ given that $w_{pop.} = 1$, using the knowledge about the $\delta_{rel-ord}$ and $\delta_{prox}$. The values in Figure 8.1 show how these dependency values change over time. Initially, population dependencies are close to zero and in fifteen generations, this converges to approximately $\frac{5}{6}$. Since the initial values are so low, weights starting from 0.05 will already steer the building of the linkage tree. As the standard deviations are at most 0.2 at the end of the algorithm, a weight higher than 0.2 will possibly exploit domain knowledge too much. Values close to 0.05 will however use this domain knowledge much less.

We can also use weights that change over time. This can be done using an exponential cooling scheme that reduces the weight per generation. The exponential cooling scheme starts with an initial weight $w^1_{dom.}$ for generation 1. The weight for generation $i$ is computed by:

$$w^i_{dom.} = \alpha \cdot w^{i-1}_{dom.}, \tag{8.5}$$

where $\alpha$ is the cooling down rate, in the range $[0, 1]$.

## 8.3 Experimental results

In order to test the effect of dependency seeding on pGOMEA, we will compare the different forms of dependency seeding and two types of weighting the domain knowledge. For these tests pGOMEA settings and Taillard instances are used as described in Section 5.1. Mostly, the TFT criterion is used as quality measure for the schedules. First, we investigate the effect of different weights (Section 8.3.1) and cooling schemes (Section 8.3.2) on the quality of generated schedules; here we use RZ-dependency for $\delta_0$. Using weights from these experiments, we compare heuristic-dependency seeding and index-dependency seeding in Section 8.3.3. When presenting the results, we will mark the best result using bold text. Results that are signifficantly worse than the best result have a grey cell.

### 8.3.1 Fixed weight dependency seeding

As we don't know whether dependency seeding influences pGOMEA signifficantly, we conduct a first experiment that checks how good RZ-dependency seeding performs when fixed weights are used. The seeded pGOMEA instances are compared with standard pGOMEA and a dependency measure giving a random value from [0.1]. In order to do statistical testing, a limited set of instances is used, for which the algorithms are run 20 times. Table 8.1 and 8.2 show the MRPD values (ARPD in brackets) for each of these instances and weights. We use population knowledge and domain knowledge with weights $w_{pop}$-$w_{dom}$ as shown in the table. Bold values represent the best dependency configurations, configurations that perform statistical significantly ($p = 0.05$) worse than the best solution are shaded in gray. Statistical test are performed using the Mann-Whitney-Wilcoxin test as described in 3.2.

The results show that a random dependency measure performs significantly worse than most optimal configurations. This confirms that pGOMEA benefits from learning a good model. This model, should not necessarily only match the population. Especially on smaller instances ($J = 50$), it is sufficient or even optimal to learn a fixed model ($w_a = 0.0$, $w_b = 1.0$) using the RZ heuristic. Finding the right weights is however very dependent on the problem size: not using population knowledge is best for instances with 50 jobs, but the weights $w_{pop} = 2.0$, $w_{dom} = 1.0$ are best for problems with 100 jobs. In general, we can say that dependency seeding can significantly improve pGOMEA provided that the right weights are given, good weights are $(4.0, 1.0)$, close to the ratio $(1, 0.2)$ as theoretically described.

### 8.3.2 Exponential weight cooling scheme

As we have seen, dependency seeding is promising for pGOMEA. However,it is hard to find robust weights. In this experiment we will therefore test the cooling scheme as alternative for weighting

| | $(50 \times 5)$ | | | $(50 \times 10)$ | | | $(50 \times 20)$ | | |
|---|---|---|---|---|---|---|---|---|---|
| Best | 64803 | 68062 | 63162 | 87207 | 82820 | 79987 | 125831 | 119259 | 116459 |
| 1.0-0.0 | **0.87 (0.86)** | 1.02 (1.03) | 1.45 (1.47) | 1.67 (1.71) | 1.73 (1.66) | 1.27 (1.32) | 1.32 (1.39) | 1.34 (1.35) | 1.61 (1.61) |
| 4.0-1.0 | 0.87 (0.89) | 1.09 (1.08) | **1.38 (1.38)** | 1.67 (1.67) | **1.63 (1.68)** | 1.38 (1.31) | 1.38 (1.42) | 1.29 (1.19) | 1.45 (1.45) |
| 2.0-1.0 | 0.94 (0.93) | 1.08 (1.08) | 1.48 (1.50) | 1.67 (1.65) | 1.68 (1.64) | 1.40 (1.39) | 1.42 (1.43) | 1.07 (1.06) | **1.32 (1.33)** |
| 1.0-1.0 | 1.02 (1.15) | 1.07 (1.08) | 1.40 (1.38) | 1.66 (1.67) | 1.67 (1.64) | **1.20 (1.27)** | 1.43 (1.42) | 1.37 (1.31) | 1.47 (1.45) |
| 1.0-2.0 | 1.15 (1.15) | 1.10 (1.11) | 1.53 (1.44) | 1.62 (1.59) | 1.74 (1.82) | 1.67 (1.69) | 1.34 (1.31) | 1.28 (1.31) | 1.67 (1.64) |
| 1.0-4.0 | 1.10 (1.06) | 1.08 (1.08) | 1.48 (1.49) | **1.59** (1.60) | 1.85 (1.86) | 1.69 (1.70) | 1.37 (1.34) | 1.30 (1.27) | 1.61 (1.61) |
| 0.0-1.0 | 0.91 (1.06) | **0.93 (0.93)** | 1.40 (1.40) | 1.65 (1.66) | 1.64 (1.67) | 1.58 (1.55) | **1.28 (1.29)** | **0.92 (0.93)** | 1.40 (1.38) |
| Random | 1.08 (1.06) | 1.34 (1.26) | 1.75 (1.76) | 1.72 (1.73) | 1.75 (1.80) | 1.58 (1.56) | 1.56 (1.52) | 1.31 (1.30) | 1.51 (1.51) |

Table 8.1: Quality of RZ-dependency seeding with different weights on multiple $(50 \times M)$ instances

| | $(100 \times 5)$ | | | $(100 \times 10)$ | | | $(100 \times 20)$ | | |
|---|---|---|---|---|---|---|---|---|---|
| Best | 253713 | 242777 | 238180 | 229431 | 274593 | 288630 | 367267 | 374032 | 371417 |
| 1.0-0.0 | 2.00 (2.00) | **2.00 (2.09)** | 2.10 (2.07) | 2.31 (2.26) | 2.66 (2.56) | 2.52 (2.53) | 2.51 (2.51) | 2.31 (2.34) | 2.40 (2.37) |
| 4.0-1.0 | 1.97 (1.89) | 2.23 (2.20) | 1.99 (1.95) | 2.31 (2.31) | 2.76 (2.72) | 2.54 (2.54) | 2.44 (2.44) | 2.48 (2.42) | 2.33 (2.28) |
| 2.0-1.0 | 1.91 (1.89) | 2.23 (2.22) | 1.96 (1.95) | **2.25 (2.23)** | **2.57 (2.55)** | 2.51 (2.42) | **2.42 (2.43)** | **2.20 (2.25)** | 2.30 (2.28) |
| 1.0-1.0 | 1.98 (1.96) | 2.34 (2.31) | 1.88 (1.87) | 2.26 (2.18) | 2.62 (2.53) | **2.24 (2.27)** | 2.43 (2.43) | 2.29 (2.22) | 2.23 (2.26) |
| 1.0-2.0 | 1.91 (1.89) | 2.41 (2.37) | **1.75 (1.80)** | 2.37 (2.36) | 2.64 (2.64) | 2.82 (2.69) | 2.46 (2.42) | 2.23 (2.30) | **2.23 (2.30)** |
| 1.0-4.0 | **1.82 (1.89)** | 2.24 (2.23) | 1.95 (1.87) | 2.43 (2.41) | 2.62 (2.60) | 2.77 (2.69) | 2.46 (2.50) | 2.34 (2.34) | 2.36 (2.42) |
| 0.0-1.0 | 2.61 (2.62) | 2.88 (2.78) | 2.62 (2.61) | 2.50 (2.41) | 2.73 (2.72) | 2.74 (2.66) | 2.73 (2.70) | 2.30 (2.32) | 2.25 (2.29) |
| Random | 2.26 (2.31) | 2.55 (2.57) | 2.29 (2.24) | 2.48 (2.53) | 2.83 (2.77) | 2.52 (2.48) | 2.46 (2.50) | 2.44 (2.41) | 2.34 (2.24) |

Table 8.2: Quality of RZ-dependency seeding with different weights on multiple $(100 \times M)$ instances

| | $(50 \times 5)$ | | | $(50 \times 10)$ | | | $(50 \times 20)$ | | |
|---|---|---|---|---|---|---|---|---|---|
| Best | 64803 | 68062 | 63162 | 87207 | 82820 | 79987 | 125831 | 119259 | 116459 |
| pGOMEA | 0.87 (0.86) | 1.02 (1.03) | 1.45 (1.47) | 1.67 (1.71) | 1.74 (1.66) | 1.27 (1.32) | 1.32 (1.39) | 1.34 (1.34) | 1.61 (1.61) |
| $\alpha = 1.00$ | 0.87 (0.89) | 1.09 (1.08) | 1.38 (1.38) | 1.67 (1.67) | 1.63 (1.68) | 1.37 (1.30) | 1.38 (1.42) | 1.29 (1.18) | 1.45 (1.45) |
| $\alpha = 0.90$ | **0.74 (0.76)** | 0.96 (0.96) | 1.38 (1.38) | 1.55 (1.54) | 1.72 (1.73) | 1.33 (1.24) | 1.38 (1.35) | **1.01 (1.07)** | 1.33 (1.33) |
| $\alpha = 0.80$ | 0.80 (0.82) | **0.93 (0.95)** | 1.53 (1.48 | **1.49 (1.51)** | 1.60 (1.57) | **1.06 (1.12)** | **1.21 (1.23)** | 1.07 (1.05) | 1.47 (1.38) |
| $\alpha = 0.67$ | 0.80 (0.82) | 1.05 (1.08) | **1.36 (1.36)** | 1.61 (1.58) | 1.60 (1.62) | 1.40 (1.37) | 1.28 (1.28) | 1.17 (1.18) | 1.45 (1.45) |
| $\alpha = 0.50$ | 0.79 (0.79) | 1.03 (0.94) | 1.53 (1.53) | 1.60 (1.69) | **1.45 (1.47)** | 1.28 (1.30) | 1.37 (1.29) | 1.17 (1.12) | 1.30 (1.30) |
| $\alpha = 0.25$ | 0.90 (0.83) | 1.04 (1.05) | 1.42 (1.36) | 1.84 (1.77) | 1.52 (1.52) | 1.34 (1.32) | 1.33 (1.37) | 1.24 (1.20) | 1.52 (1.44) |
| $\alpha = 0.00$ | 0.91 (0.90) | 1.08 (1.07) | 1.46 (1.45) | 1.85 (1.78) | 1.67 (1.71) | 1.22 (1.26) | 1.36 (1.36) | 1.17 (1.13) | **1.29 (1.31)** |

Table 8.3: Quality of RZ-dependency seeding with different cooling values on multiple $(50 \times M)$ instances

| | $(100 \times 5)$ | | | $(100 \times 10)$ | | | $(100 \times 20)$ | | |
|---|---|---|---|---|---|---|---|---|---|
| Best | 253713 | 242777 | 238180 | 229431 | 274593 | 288630 | 367267 | 374032 | 371417 |
| pGOMEA | 2.01 (2.00) | 2.00 (2.09) | 2.10 (2.07) | 2.31 (2.26) | 2.66 (2.56) | 2.52 (2.53) | 2.51 (2.51) | 2.30 (2.34) | 2.40 (2.37) |
| $\alpha = 1.00$ | 1.97 (1.98) | 2.23 (2.20) | 1.99 (1.95) | 2.31 (2.31) | 2.76 (2.72) | 2.54 (2.53) | 2.44 (2.44) | 2.48 (2.42) | 2.33 (2.28) |
| $\alpha = 0.90$ | 1.84 (1.84) | 2.08 (2.06) | 1.73 (1.70) | 2.18 (2.14) | 2.43 (2.48) | 2.23 (2.15) | 2.29 (2.25) | 2.30 (2.31) | 2.22 (2.19) |
| $\alpha = 0.80$ | 1.75 (1.75) | 1.98 (1.99) | 1.70 (1.69) | **1.99 (2.05)** | 2.41 (2.42) | **2.20 (2.15)** | 2.41 (2.32) | **2.20 (2.19)** | 2.23 (2.18) |
| $\alpha = 0.67$ | **1.74 (1.74)** | 2.12 (2.14) | 1.72 (1.73) | 2.12 (2.08) | **2.32 (2.27)** | 2.23 (2.21) | 2.41 (2.39) | 2.37 (2.27) | 2.20 (2.24) |
| $\alpha = 0.50$ | 1.87 (1.87) | 2.09 (2.10) | 1.73 (1.73) | 2.13 (2.15) | 2.44 (2.48) | 2.33 (2.36) | 2.36 (2.28) | 2.26 (2.27) | **2.17 (2.18)** |
| $\alpha = 0.25$ | 1.90 (1.92) | **1.90 (1.91)** | **1.65 (1.66)** | 2.26 (2.23) | 2.48 (2.50) | 2.33 (2.32) | **2.26 (2.27)** | 2.24 (2.24) | 2.28 (2.24) |
| $\alpha = 0.00$ | 1.93 (1.90) | 2.16 (2.16) | 1.85 (1.83) | 2.14 (2.15) | 2.61 (2.66) | 2.56 (2.55) | 2.52 (2.49) | 2.30 (2.31) | 2.32 (2.22) |

Table 8.4: Quality of RZ-dependency seeding with different cooling values on multiple $(100 \times M)$ instances

the domain knowledge. The convergence speed of different population sizes can ask for different cooling schemes. Finding the right cooling scheme can again be very difficult, since no fixed population size is used in pGOMEA. In this experiment we will test this assumption. We compare a limited amount of cooling schemes starting with $w_{pop.} = 1.0$ and $w^1_{dom.} = 4.0$, meaning that domain knowledge dominates. We vary the cool-down rate $\alpha$ from 0.0 to 1.0, i.e. we compare immediate cooling, no cooling and cooling rates in between.

Tables 8.3 and 8.4 show the results of this experiment. As can be seen, a cooling scheme can hugely improve pGOMEA. In no case, pGOMEA performs best and in about three quarters of the cases, pGOMEA performs worse than optimal dependency seeded pGOMEA. The cooling scheme also improves on pGOMEA without a cooling scheme. If $\alpha = 1.0$, the algorithm never performs best and often performs significantly worse than the best cooling scheme. This best cooling scheme has an alpha rate of 0.8. In only one case, this cooling scheme gives significantly worse results than another cooling scheme; seven times, it gives the best result itself. We can therefore conclude that the choice of parameters is easier and often gives better results than choosing fixed weights. We also see that the best cooling scheme is good on any type of instance, while best results for fixed weights depend on the amount of jobs in the instance.

### 8.3.3 Heuristic dependencies

Now that we know what a good weight-configuration is, we can compare different forms of dependency seeding. Here, we compare heuristic-dependency seeding with the index-dependency seeding. For the weight-configuration we use the cooling scheme with $w_{pop.} = 1.0$, $w^1_{dom.} = 4.0$ and $\alpha = 0.8$, which performed best in the previous experiment. The heuristic dependency values are found over a population with a maximum amount of schedules generated from the CDS heuristic (creating $M - 1$ schedules) for $C_{max}$ and the LR heuristic (creating $J$ schedules) for the TFT criterion.

The results, shown in Tables 8.5 and 8.6, show how dependency seeding almost always gives the best solution. Though not visible in the tables, index-dependency seeding never performs significantly worse than pGOMEA, index-dependency seeding is twice outperformed by heuristic-dependency seeding. While heuristic-dependency seeding sometimes performs very good, the constructive heuristic can have a bias, which is ipresent in all its results. This leads to very high dependencies in terms of relative ordering. Index-dependency seeding is able to capture the dependencies better by calculating the relative distance between the indices.

| | (50 × 5) | | | (50 × 10) | | | (50 × 20) | | |
|---|---|---|---|---|---|---|---|---|---|
| Best | 2724 | 2834 | 2621 | 2991 | 2867 | 2839 | 3850 | 3704 | 3640 |
| pGOMEA | **0.00 (0.00)** | 0.07 (0.07) | **0.00 (0.00)** | 1.60 (1.57) | 1.55 (1.66) | 1.13 (1.23) | 1.84 (1.78) | 1.70 (1.61) | **2.14 (2.15)** |
| Palmer-depencency | **0.00 (0.00)** | 0.07 (0.07) | **0.00 (0.00)** | **1.57 (1.46)** | **1.53 (1.63)** | 1.23 (1.28) | **1.64 (1.64)** | 1.70 (1.70) | 2.17 (2.10) |
| Heuristic-dependency | **0.00 (0.00)** | **0.07 (0.05)** | **0.00 (0.00)** | 1.65 (1.67) | 1.53 (1.58) | **1.13 (1.17)** | 1.78 (1.81) | **1.46 (1.45)** | 2.35 (2.33) |
| | (100 × 5) | | | (100 × 10) | | | (100 × 20) | | |
| Best | 5493 | 5268 | 5175 | 5770 | 5349 | 5676 | 6202 | 6183 | 6271 |
| pGOMEA | **0.00 (0.00)** | 0.13 (0.10) | **0.00 (0.00)** | **0.23 (0.20)** | 0.52 (0.43) | **0.05 (0.08)** | 2.89 (2.89) | 2.54 (2.52) | 2.29 (2.26) |
| Palmer-depencency | **0.00 (0.00)** | **0.13 (0.09)** | **0.00 (0.00)** | 0.23 (0.23) | **0.24 (0.32)** | 0.05 (0.11) | **2.53 (2.59)** | **2.35 (2.33)** | **2.09 (2.18)** |
| Heuristic-dependency | **0.00 (0.00)** | 0.13 (0.14) | **0.00 (0.00)** | 0.25 (0.29) | 0.52 (0.42) | 0.05 (0.11) | 2.81 (2.75) | 2.46 (2.48) | 2.18 (2.14) |

Table 8.5: Quality of two forms of dependency seeding and pGOMEA for the $C_{max}$ criterion.

These results also indicate that the main reason for the quality improvement for multi-solution population-seeding does not lie in the improved dependency learning by pGOMEA. Instead, the amount of different good solutions are the main contribution of the population-seeding improvement. This is also supported by side-experiments showing that multi-solution population seeding performs significantly better than index-dependency seeding and heuristic-dependency seeding.

| | (50 × 5) | | | (50 × 10) | | | (50 × 20) | | |
|---|---|---|---|---|---|---|---|---|---|
| Best | 64803 | 68062 | 63162 | 87207 | 82820 | 79987 | 125831 | 119259 | 116459 |
| pGOMEA | 0.87 (0.86) | 1.02 (1.03) | **1.45 (1.47)** | 1.67 (1.71) | 1.74 (1.66) | 1.27 (1.32) | 1.32 (1.39) | 1.34 (1.34) | 1.61 (1.61) |
| RZ-depencency | 0.80 (0.82) | **0.93 (0.95)** | 1.53 (1.48) | **1.49 (1.51)** | **1.60 (1.57)** | **1.06 (1.12)** | **1.21 (1.23)** | **1.07 (1.05)** | 1.47 (1.38) |
| Heuristic-dependency | **0.79 (0.80)** | 1.06 (1.11) | 1.46 (1.49) | 1.72 (1.79) | 1.59 (1.67) | 1.45 (1.43) | 1.29 (1.30) | 1.26 (1.29) | **1.32 (1.36)** |
| | (100 × 5) | | | (100 × 10) | | | (100 × 20) | | |
| Best | 253713 | 242777 | 238180 | 229431 | 274593 | 288630 | 367267 | 374032 | 371417 |
| pGOMEA | 2.01 (2.00) | 2.00 (2.09) | 2.10 (2.07) | 2.31 (2.26) | 2.66 (2.56) | 2.52 (2.53) | 2.51 (2.51) | 2.30 (2.34) | 2.40 (2.37) |
| RZ-depencency | **1.75 (1.75)** | **1.98 (1.99)** | **1.70 (1.69)** | **1.99 (2.05)** | **2.41 (2.42)** | **2.20 (2.15)** | **2.41 (2.32)** | **2.20 (2.19)** | **2.23 (2.18)** |
| Heuristic-dependency | 2.20 (2.17) | 2.26 (2.30) | 1.98 (1.99) | 2.28 (2.24) | 2.44 (2.47) | 2.44 (2.41) | 2.53 (2.51) | 2.23 (2.29) | 2.45 (2.37) |

Table 8.6: Quality of two forms of dependency seeding and pGOMEA for the TFT criterion.

## 8.4 Conclusions

- **Seeding enables pGOMEA to build a better model from the first generation of a population.**
  As shown in Section 8.1, a fully seeded population gives non-random results in building the linkage tree. Instead, pGOMEA is able to learn dependencies with strengths that would normally be found after a few generations. The dependency values per generation show no sign of changes in later generations, indicating that seeding has the biggest effect on the linkage tree in the first generations.

- **Weight cooling schemes give best results for dependency seeding.**
  As shown in the experiments, fixed-weight dependency seeding gives difficulties in finding the right values for the weights. As pGOMEA should be able to find new, better dependencies from the population in later generations, the weights for the seeded dependency should not be to high. In order to exploit the seeded dependencies in early generations, the weights should also not be too low. A cooling scheme is able to combine both high weights for the seeds in the first generations and low weights in the later generations. Using the seeded dependencies only in the first generation already improves on pGOMEA. Better cooling parameters result in significantly better results than pGOMEA. The close to optimal cooling parameter of $\alpha = 0.8$ has shown to be effective for instances with different amounts of jobs and machines.

- **Index-dependency seeding performs better than heuristic-dependency seeding.**
  As index-dependency seeding directly transforms domain knowledge (in the form of indices) into dependencies, no extra bias is introduced. Incorrect assumptions are likely to introduce only a small error. Heuristic-dependency seeding however uses domain knowledge in order to create a population. Every solution in this population suffers from this small error. When calculating dependency values, this error becomes larger as it is present throughout the population.

- **dependency seeding is a good alternative if multi-solution population-seeding is not possible.**
  Though multi-solution population-seeding outperforms dependency seeding, dependency seeding can still improve pGOMEA. For instance in cases where multi-solution population-seeding is not possible (e.g. only the RZ heuristic is known). One can then seed dependency-knowledge in pGOMEA, leading to better crossovers in the first iterations of pGOMEA. After these iterations, pGOMEA can learn better dependencies using population-knowledge. Therefore, the weight of dependency-seeds should be decreased using a cooling scheme. This approach leads to good results, given that the seeded dependencies reflect true dependencies between variables.

# Chapter 9

# Substructural neighborhoods: Experimental study

In previous chapters we have seen how pGOMEA can be improved when information from domain knowledge is added (e.g. (dependency) seeding and hybridization). In this chapter, we will look into the combination of domain knowledge and model learning from another perspective. Here, we study the concept of substructural neighborhoods as explained in Section 4.4. We have researched switching between domain knowledge and model knowledge (seeding and hybridization) we have also seen how domain knowledge can enhance the model (dependency-seeding). What is left, is to seen how model knowledge can enhance domain specific operators (heuristics). Substructural neighborhood searching is such a form of combining model-knowledge and domain-knowledge. As research guideline we use the following question:

*How can model knowledge be used in domain knowledge-based search methods?*

In order to research this question, we first identify the ways a substructral neighborhood can be defined and used in Section 9.1. Using this information, we experiment with various forms of neighborhood searchers in Section 9.2. We end this chapter with some conclusions in Section 9.3.

## 9.1 Substructural neighborhoods for pGOMEA

Substructural neighborhood searching uses the building blocks of a problem in order to define the neighborhood for a local searcher. Building blocks are defined as a set of variables that are dependent on each other. MBEAs try to learn the building blocks for problems using model-building. GOMEA has sets in the linkage tree as building blocks, eCGA uses sets in the marginal product model as building blocks, while building blocks for BOA can be defined using the Parental and/or Child-neighborhood of nodes in the Bayesian network.
A simple substructural neighborhood searcher for problems in Carthesian space uses the variable-sets to search in the neighborhood of a solution. A solution is updated using a set with $k$ m-valued variables, by selecting the best out of the $m^k$ neighbors differing in the $k$ variables. Substructural neighborhood searchers have been analyzed as stand-alone optimization algorithms [33] and as a part of an MBEA [4, 23].

### 9.1.1 Difficulties for substructural neighborhoods in pGOMEA

Substructural neighborhoods in this form have some drawbacks for hybridization or combination with pGOMEA.

- **Effectiveness:** Substructural neighborhoods have only been effective when not hybridized in an MBEA or when a fitness approximation can be used as possible for BOA [23].
- **Overlapping subsets:** Sastry and Goldberg claim that *since the effect of overlapping variable interactions is similar to that of exogenous noise, .. a crossover is likely to be more useful*

*than mutation* [33]. As the linkage tree assumes that the problem has overlapping building blocks and variables in permutation problems have some relation to any other variable, this suggests that substructural neighborhoods are not suitable for permutation problems and pGOMEA.

- **Permutation neighborhood:** As pGOMEA tackles permutation problems, a neighborhood is different than in GOMEA. Suppose we want to perform a substructural neighborhood search on the permutation $(x_1, x_3, x_5, x_2, x_4)$ using the linkage-set $\{x_1, x_2\}$, how can this be done effectively?
  Swapping the variables $x_1$ and $x_2$ can be considered a disturbance of the building-block, as building blocks are based on relative-ordering which is broken by the swap. The swap also affects the variables between the swapped variables, meaning that it influences other building blocks. For bigger linkage-sets of size $k$, the neighborhood consists of $k!$ new variable-orders, which can become too large for most linkage-sets.
  A better approach is to perform insertion: variables $x_1$ and $x_2$ are kept in order, but are moved towards each other, possibly towards another position. Possible neighbors are $(x_1, x_3, x_2, x_5, x_4)$ and $(x_3, x_1, x_2, x_5, x_4)$. This form of insertion is however already incorporated in pGOMEA in the form of rescaling.

Using the knowledge about the difficulties of combining pGOMEA and substructural neighborhoods, we define two methods combining pGOMEA and substructural neighborhood searchers worth investigating:

### 9.1.2 Insertion-based substructural neighborhood searcher: Description

Like the substructural neighborhood searcher introduced by Sastry and Goldberg [33], we can design a mutation-only pGOMEA variant. First we create a linkage tree using pGOMEA's dependencies over a good population. The substructural neighborhood searcher then improves the best solution of that population by repeatedly applying the rescaling operator on it. Rescaling is tried until a local optimum has been found or until the computational budget is spend. The outline of this substructural neighborhood searcher based on pGOMEA is shown in Algorithm 3.

**Result:** A good/optimal solution with respect to fitness function $f$
$Pop \leftarrow rand\_Pop(n)$;
$Pop \leftarrow Selection(Pop)$;
$FOS \leftarrow build\_FOS(Pop)$;
$solution \leftarrow best(Pop)$;
**while** $\neg termination\_criterion$ **do**
  **foreach** $set \in FOS$ **do**
    $solution^* \leftarrow Rescale(solution, set, solution)$;
    **if** $f(solution^*) \geq f(solution)$ **then**
      $solution \leftarrow solution^*$
  Re-encode(solution);
**return** *solution*

**Algorithm 3:** Insertion based substructural neighborhood searcher

Instead of using selection over a random population, as shown in the algorithm outline, one can also create a good population using a constructive heuristic. Also, the algorithm uses re-encoding, as it add random key diversity, which is effective for pGOMEA. Instead of using the rescaling operator, one can also try to scramble the values in a linkage set. Early experiments however have shown that this is not very effective. This will not be researched any further.

### 9.1.3 Model-based swapping in pGOMEA: Description

Insertions are already present in pGOMEA in the form of rescaling, similarly we can add a swap operator. This swap operation is performed as mutation after donations are performed. The operator changes the schedule by swapping the values of variables that are contained in two linkage sets with the same cardinality. The change is only accepted if this results in an improvement of the solution. This model-based swapper is used to hybridize pGOMEA.

**Input:** $FOS$, $solution$
**Result:** An solution, improved using the FOS
**foreach** $(S_1, S_2) \in FOS \times FOS$ **do**
  **if** $|S_1| \neq |S_2|$ **then**
    **continue**;
  $solution^* \leftarrow solution$;
  $solution^* \leftarrow Swap(solution^*, S_1, S_2)$;
  **if** $f(solution^*) \geq f(solution)$ **then**
    $solution \leftarrow solution^*$
**return** $solution$

**Algorithm 4:** Substructural swapper.

An outline of this substructural swapping is given in Algorithm 4. Though the pseudocode suggests that in worst case $\mathcal{O}(|FOS|^2)$ sets are matched, this can be reduced. First, the contents of the FOS are sorted based on size ($\mathcal{O}(|FOS| \cdot \log |FOS|)$, after which matching pairs can easily be found. Note that in a linkage tree, all singleton sets are present, still leading to $\mathcal{O}(|FOS|^2)$ matches and fitness evaluations.

## 9.2 Experimental results

We have experimentally tested the quality of the two proposed algorithms. Both algorithms are tested against variants using only the singleton sets in the linkage tree (i.e. the simple improvement heuristics). As a computational budget, we use a standard amount of fitness evaluations.

### 9.2.1 Insertion-based substructural neighborhood searcher: Experiments

To determine the quality of the insertion-based substructural neighborhood searcher, we compare three forms of this substructural neighborhood searcher. The first type learns the linkage tree over the top-1000 solutions in a population with 2000 random solutions. The second type learns the linkage tree over a population with a maximal amount of constructive solutions. The last type constructs the linkage tree using RZ-dependency seeding. Each form starts its substructural search from the best solution in the population. When the constructive heuristic is used, the best schedule in the population is already very good, leading to an advantage. This is also visible in results of Table 9.1, where the best result is marked bold and a signifficantly worse result has a grey cell. Here, we see that though the constructive approach works very well, this is mainly contributed to the fact that a good solution is taken as starting point. This can be seen as the simple insertion hill-climber performs equally well.
Wecondly, we see that the RZ heuristic is better in building a tree for the substructural neighborhood searcher than the Top-1000 searcher. The substructural neighborhood searcher here outperforms the simple insertion searcher, though it uses about three times more fitness evaluations until convergence. Therefore, we might be able to use this form of model building to improve local searchers like tabu-search.

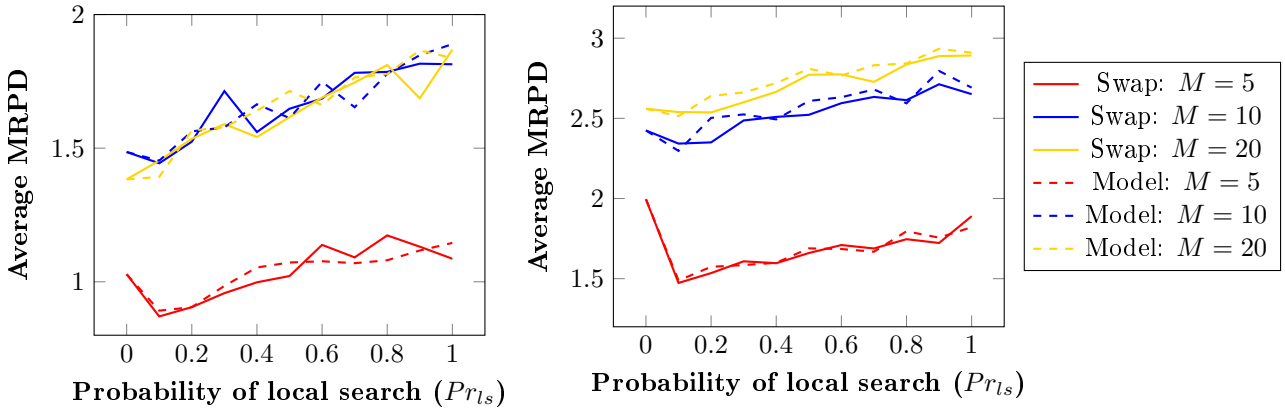
| | | Top-1000 | | RZ | | Constructive | |
|---|---|---|---|---|---|---|---|
| | Best known | Insert | Substruct | Insert | Substruct | Insert | Substruct |
| 50x5 | 64803 | 3.71 | **2.94** | 3.04 | **2.38** | **2.53** | **2.53** |
| 50x5 | 68062 | 4.12 | **3.96** | 4.26 | **2.53** | **0.86** | 0.90 |
| 50x5 | 63162 | **4.44** | 4.61 | 4.68 | **3.14** | **1.77** | 2.00 |
| 50x10 | 87207 | 4.38 | **4.31** | 4.38 | **4.05** | 1.79 | **1.69** |
| 50x10 | 82820 | **4.23** | 4.57 | 4.68 | **4.26** | **2.13** | 2.48 |
| 50x10 | 79987 | 4.61 | **4.34** | 4.52 | **3.30** | 3.36 | **2.81** |
| 50x20 | 125831 | **3.77** | 4.23 | 4.28 | **3.31** | **2.32** | 2.44 |
| 50x20 | 119259 | **3.68** | 3.83 | 3.67 | **3.08** | **1.86** | 2.06 |
| 50x20 | 116459 | 4.34 | **3.67** | 4.12 | **3.17** | 2.49 | **2.25** |

Table 9.1: Simple insertion heuristic versus insertion-based substructural neighborhood searcher.

In order to combine the advantages from the constructive and RZ substructural neighborhood searcher, we create a new searcher. This substructural searcher starts from the best constructive result, while it uses the RZ-dependency linkage tree. This does however not show significant changes with respect to the behavior of the constructive-based substructural neighborhood searcher. This indicates that on more optimized solutions, the difference between a simple insertion neighborhood searcher and a substructural neighborhood searcher is not significant, except in the amount of fitness evaluations before a local optimum is found (the substructural searcher uses more fitness evaluations). We can therefore conclude that a substructural insertion searcher is not able to outperform the simple insertion heuristic.

### 9.2.2 Model-based swapping in pGOMEA: Experiments

In order to test the quality of pGOMEA with the model-based swap heuristic, we perform experiments on the PFSP with the TFT criterion. As the standard swap heuristic is able to improve pGOMEA, the model-based swap heuristic might perform even better. Here, we compare the one-pass model-based swap heuristic with the same swap heuristic, ignoring sets with cardinality larger than one. Figure 9.1 shows for different probabilities of local search how model-based swapping differs from normal swapping.




(a) $J = 50$

(b) $J = 100$

Figure 9.1: Hybrid pGOMEA with the standard and model-based swap heuristic.

Here we see that there is not a big difference between model-based searching and a simple swap heuristic. If there is any significant difference, standard swapping outperforms model-based swapping. Therefore, we can conclude that model-based swapping does not help pGOMEA. Instead, we see that limiting the swaps to a single pass does help pGOMEA. In contrast to exhaustive swapping (Chapter 7), one-pass swapping improves pGOMEA with low probabilities of local search for almost any instance of the PFSP with the TFT criterion. It can therefore be worthwhile to design a (parameter-based) local searcher that limits the swap heuristic in the right way to help it improve pGOMEA.

## 9.3 Conclusions

- **A substructural insertion-based neighborhood searcher does not perform significantly better on good solutions than a simple insertion-based neighborhood searcher.**
  The insertion-based substructural neighborhood of pGOMEA is significantly better than the insertion neighborhood, when starting from a random solution. On more optimized solutions (e.g. constructed with a constructive heuristic) the substructural neighborhood is not better than the insertion neighborhood. Therefore a meta-heuristic using a substructural neighborhood searcher with the linkage tree has no great benefits with respect to an insertion local search.

- **A model-based swap heuristic has no benefits over a standard swap heuristic.** The standard swap heuristic performs better or equally well as the model-based swap heuristic, when incorporated in pGOMEA. The quality of the standard and model-based swap heuristic both depend on the amount of jobs and machines in the problem instances.
- **A one-pass swap heuristic is more promising than an exhaustive swap heuristic.** In contrast to the exhaustive swap heuristic, the one-pass heuristic has low probabilities of local search for which it improves pGOMEA for almost any instance size. Therefore, hybridizing pGOMEA for the TFT with a form of a swap heuristic is worthwhile to research. As local search is cheaper than optimal mixing, hybridizing pGOMEA might be useful when we use the running time as computational budget for pGOMEA.

# Chapter 10

# Comparative results

In previous chapters we have determined how pGOMEA can be improved by adding domain knowledge to it. We are can now compare pGOMEA with other algorithms designed to solve the PFSP. In this chapter we will therefore compare pGOMEA with two well-performing PFSP algorithms. We compare our best implementation of pGOMEA for the $C_{max}$ criterion with an Iterative Greedy (IG) algorithm. We also compare our best pGOMEA implementation for the TFT criterion with the VNS4 algorithm. Besides the standard Taillard benchmarks, we also use structured instances in order to review pGOMEA's quality. For these problems, we also research the influence of the dependency measure, by comparing pGOMEA with a variant that uses random dependencies. The main question answered in this chapter is:

*How does pGOMEA with domain knowledge compare to state-of-the-art algorithms for the permutation flowshop scheduling problem and how does pGOMEA's performance depend on structure in the instances?*

First, we will introduce the used algorithms in Section 10.1. Sections 10.2 and 10.3 compare the algorithms pairwise on the Taillard and Watson instances. After this we test pGOMEA on a new benchmark set in Section 10.4. Here we compare pGOMEA both with the state-of-the-art algorithms as well as a dependency-free version of pGOMEA. We end this chapter with conclusions on the quality of pGOMEA with respect to state-of-the-art algorithms under different circumstances.

## 10.1 Algorithms

For the comparison of pGOMEA with state-of-the-art algorithms, we use two easy-to-implement algorithms: VNS4 and IG. These algorithms solve the PFSP with the TFT and $C_{max}$ criterion respectively. We compare these algorithms with pGOMEA using the best types of hybridization and seeding. In this section we shortly introduce the algorithms in text and pseudocode. Our implementations of the algorithms have been optimized to the level of big-O notation and its correctness has been verified by comparision with code of the original authors.

### 10.1.1 VNS4

VNS4 is a variable neighborhood searcher, switching between the swap neighborhood and insertion neighborhood. VNS4 optimizes the balance between the swap and insertion neighborhood. Therefore VNS4 starts with an exhaustive search in the swap neighborhood, whereafter a one-pass insertion heuristic is started. The best solution from an $LR(J/M)$ heuristic is used as initial solution. When a local optimum is found with respect to both neighborhoods, the best solution is updated and a new search is started from a perturbation of the best solution. An outline of this algorithm is given in Algorithm 5.

### 10.1.2 pGOMEA for PFSP with the TFT criterion

pGOMEA for the TFT criterion can greatly benefit from population seeding, as already shown in Chapter 6. Therefore our pGOMEA implementation will use the $LR(n)$ heuristic for seeding the

population. As the swap heuristic can sometimes improves pGOMEA, we used parameter tuning in order to find a good swap heuristic that is globally applicable.

#### Parameter tuning for the swap heuristic

In previous chapters we found that the swap heuristic was the most promising heuristic for hybridizing pGOMEA when solving the PFSP with the TFT ccriterion. The results also showed that the swap heuristic performed best on instances with a large amount of jobs and a low amount of machines. Limiting the amount of considered swaps can potentially give better results. Therefore, we propose a swap heuristic that considers an amount of random swaps that is dependent on the amount of machines and jobs in the instance. For that amount we propose the following value:

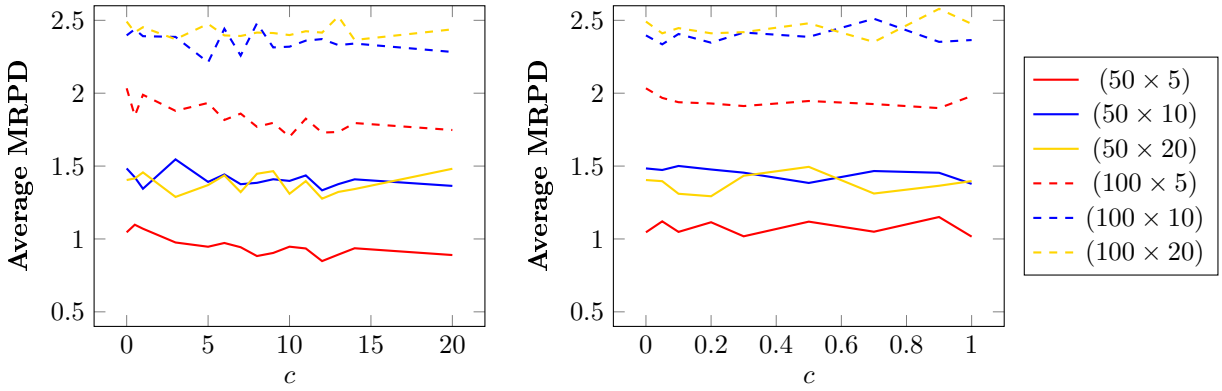
$$Swaps = c \cdot \frac{J}{M}, \tag{10.1}$$

where $c$ is some constant. Since the swap neighbor-space is quadratic, this formulation implies that: $0 < c \leq \frac{M \cdot (J-1)}{2}$, when every swap is tried at most once. Because the quadratic nature of the neighborhood-space we will also consider values of $c$ for

$$Swaps = c \cdot \frac{J \cdot (J-1)}{M}, \tag{10.2}$$

where $0 < c \lesssim \frac{M}{2}$, when every swap is tried at most once.

Given the limits of $c$ for both swap types, we research the quality of hybridized pGOMEA with respect to $c$. In our implementation, swaps are tried in a random order and lcal search is applied with a probability of 1.0 when a solution has been changed by optimal mixing.




(a) Job-linear (b) Job-quadratic

Figure 10.1: Instance-dependent swapping in pGOMEA.

Figure 10.1 shows the results for different values of $c$ when using a standard amount of fitness evaluations. The optimal value of $c$ is still dependent on the amount of jobs and machines in the problem. Therefore, we will not use an improvement heuristic in pGOMEA for the TFT criterion. The resulting pGOMEA algorithm for solving the PFSP with the TFT criterion is shown in Algorithm 6.

### 10.1.3 Iterative Greedy

The Iterative Greedy algorithm for the PFSP with the $C_{max}$ criterion uses the NEH-heuristic as basis. First, an initial solution is generated using the NEH constructive heuristic. After that the iterative phase is started. In the iterative phase, the current solution is first destructed by removing six jobs. After destruction, the solution is reconstructed by re-inserting the removed jobs in random order using the NEH-heuristic. Hereafter the NEH improvement heuristic is used to further optimize the schedule. At the end of each iteration, the best solution is updated. The new iteration is started from best or currrent solution, depending on the solution quality and randomness. An overview of the IG algorithm is shown in Algorithm 7.

**Result:** A good/optimal PFSP solution with respect to the TFT criterion
$Sol \leftarrow LR(N/M)$; // Solution constructed with LR heuristic
$Best \leftarrow Sol$;
**while** $\neg termination\_criterion$ **do**
  $condition \leftarrow true$;
  **while** $condition \wedge \neg termination\_criterion$ **do**
    $Job\_Swap\_LS(Sol)$; // Swap to local optimum
    $condition \leftarrow Reduced\_Job\_Insert(Sol)$ ; // One pass of insertions
  **if** $C_{max}(Sol) < C_{max}(Best)$ **then**
    $Best \leftarrow Sol$ ; // Update best
  $Sol \leftarrow Best$;
  $Rand\_Insertions(Sol, 14)$ ; // Shake procedure
**return** $Best$

**Algorithm 5:** Outline of VNS4 searcher

**Result:** A good/optimal PFSP solution with respect to the TFT criterion
$Pop \leftarrow rand\_Pop(P-n) \cup LR(n)$ ; // Population seeded using LR(n)
**while** $\neg termination\_criterion$ **do**
  $FOS \leftarrow build\_FOS(Pop)$;
  **foreach** $receiver \in Pop$ **do**
    $receiver^* \leftarrow receiver$;
    **foreach** $set \in FOS$ **do**
      $donor \leftarrow Random(Pop)$;
      $child \leftarrow Donate(receiver^*, set, donor)$;
      **if** $f(child) \geq f(receiver^*)$ **then**
        $receiver^* \leftarrow child$;
        $improved = true$;
**return** best solution from $Pop$

**Algorithm 6:** Outline of pGOMEA for the TFT criterion

**Result:** A good/optimal PFSP solution with respect to the TFT criterion
$Sol \leftarrow NEH()$; // Solution constructed with NEH heuristic
$Sol \leftarrow NEH\_Improve(Sol)$;
$Best \leftarrow Sol$;
**while** $\neg termination\_criterion$ **do**
  $Destruct \leftarrow Random\_Jobs(6)$;
  $Sol \leftarrow Sol/Destruct$;
  **foreach** $j \in Destruct$ **do**
    $NEH\_Insert(Sol, j)$ ; // Insert back at best possible position
  $Sol \leftarrow NEH\_Improve(Sol)$;
  **if** $TFT(Sol) < TFT(Best)$ **then**
    $Best \leftarrow Sol$ ; // Update best, continue with best
  **else if** $\neg(Random < exp^{\frac{TFT(Best)-TFT(Sol)}{4}})$ **then**
    $Sol \leftarrow Best$; // Start again from best
**return** $Best$

**Algorithm 7:** Outline of IG searcher

```
Result: A good/optimal PFSP solution with respect to the C_max criterion
Pop ← rand_Pop(P − (m − 1)) ∪ CDS(m − 1) ; // Population seeded using CDS(m-1)
while ¬termination_criterion do
    FOS ← build_FOS(Pop);
    foreach receiver ∈ Pop do
        receiver* ← receiver;
        foreach set ∈ FOS do
            donor ← Random(Pop);
            child ← Donate(receiver*, set, donor);
            if f(child) ≥ f(receiver*) then
                receiver* ← child;
                improved = true;
        if improved then
            NEH_Improve(receiver);                    // Improve using NEH-insertion
return best solution from Pop
```

**Algorithm 8:** pGOMEA for PFSP with $C_{max}$

### 10.1.4 pGOMEA for PFSP with the $C_{max}$ criterion

pGOMEA for the $C_{max}$ criterion can benefit from population seeding. In Chapter 6 we have seen that the CDS heuristic works best when a maximal amount of seeds are generated. Therefore, the initial population in our algorithm will be seeded with $M-1$ schedules generated by the CDS heuristic. pGOMEA will also incorporate the NEH-improvement heuristic. This heuristic will be used with probability 1.0 on schedules that are changed by optimal mixing. An overview of this pGOMEA algorithm is given in Algorithm 8.

## 10.2 Benchmarking: Taillard instances

For comparing pGOMEA with state-of-the-art algorithms we first use unstructured instances. For this, we use the Taillard instances, as they have often been used in literature. All algorithms get the same computational budget of $400 \cdot M \cdot J$ milliseconds, as also used in the VNS4 paper. We experiment on all $50 \times M$ and $100 \times M$ instances. For each instance we perform 20 runs of the algorithms, after which the Mann-Whitney-Wilcoxon statistical test ($p < 0.05$) is used to determine whether the best algorithm performs statistically significantly better than the others.

### 10.2.1 Solving PFSP with the TFT criterion

The quality of pGOMEA with respect to the TFT criterion is compared with VNS4. The outcomes of this experiment are shown in Table 10.1. Here, we see that pGOMEA often outperforms VNS4 as can be seen in the grey cells, where VNS4 performs significantly worse than pGOMEA. VNS4 however also sometimes performs significantly better than pGOMEA. This might be caused by the random nature of the instances. As no clear structure exists, pGOMEA might learn a misleading structure, resulting in too little exploration.

### 10.2.2 Solving PFSP with the $C_{max}$ criterion

Secondly we tested pGOMEA with respect to the $C_{max}$ criterion by comparison with IG. The results of this experiment are shown in Table 10.2. The results clearly show that IG easily finds good or optimal solutions, while pGOMEA performs significantly worse (marked grey). As IG does not use complete fitness evaluations, IG explores more results. pGOMEA however uses fitness evaluations in its optimal mixing phase. As a lot of time is used there, the running time of pGOMEA scales worse with respect to the amount of jobs and machines.

The results also show how pGOMEA does not always find the optimal solution on $(J \times 5)$ instances. Even though pGOMEA finds a close to optimal solution very fast, pGOMEA is not able to find a better solution. Possibly the forced improvement and seeds steers pGOMEA to much to a local optimum which is not strongly related to the global optimum.

| | Best | pGOMEA | VNS4 | | Best | pGOMEA | VNS4 | | Best | pGOMEA | VNS4 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $50 \times 5$ | 64803 | **0.47** | 0.54 | $50 \times 10$ | 87207 | 1.26 | **1.23** | $50 \times 20$ | 125831 | **0.78** | 1.06 |
| | 68062 | **0.63** | 0.63 | | 82820 | **0.79** | 1.42 | | 119259 | **0.56** | 0.94 |
| | 63162 | **0.92** | 1.02 | | 79987 | **0.84** | 1.07 | | 116459 | **0.71** | 1.11 |
| | 68226 | 0.85 | **0.84** | | 86581 | **0.71** | 0.99 | | 120712 | **0.92** | 0.96 |
| | 69392 | **0.65** | **0.65** | | 86450 | **0.48** | 1.15 | | 118184 | 1.25 | **1.16** |
| | 66841 | **0.65** | 0.71 | | 86637 | **0.97** | 1.00 | | 120703 | **0.82** | 1.04 |
| | 66253 | **0.60** | 0.69 | | 88866 | **0.63** | 1.07 | | 122962 | **1.01** | 1.03 |
| | 64359 | **0.52** | 0.77 | | 86824 | 1.18 | **1.09** | | 122489 | **1.02** | 1.03 |
| | 62981 | 0.69 | **0.62** | | 85526 | **1.07** | 1.24 | | 121872 | **0.99** | 1.06 |
| | 68853 | 0.93 | **0.84** | | 88077 | **0.62** | 1.11 | | 124064 | **0.90** | 1.06 |
| $100 \times 5$ | 253713 | **0.80** | 0.94 | $100 \times 10$ | 299431 | **1.02** | 1.43 | $100 \times 20$ | 367267 | 1.63 | **1.57** |
| | 242777 | **1.04** | 1.12 | | 274593 | **1.55** | 1.77 | | 374032 | **1.41** | 1.50 |
| | 238180 | **0.57** | 0.94 | | 288630 | **1.15** | 1.54 | | 371417 | **1.47** | 1.56 |
| | 227889 | **0.82** | 1.01 | | 302105 | **1.55** | 1.59 | | 373822 | **1.54** | 1.75 |
| | 240589 | **0.77** | 0.95 | | 285340 | **1.09** | 1.19 | | 370459 | 1.58 | **1.50** |
| | 232936 | **0.82** | 1.12 | | 270817 | **1.14** | 1.56 | | 372768 | **1.60** | 1.66 |
| | 240669 | **0.85** | 0.90 | | 280649 | **1.21** | 1.30 | | 374483 | **1.30** | 1.71 |
| | 231428 | **1.00** | 1.06 | | 291665 | **0.81** | 1.52 | | 385456 | **1.39** | 1.56 |
| | 248481 | 1.00 | **0.93** | | 302624 | **1.20** | 1.44 | | 376063 | **1.43** | 1.58 |
| | 243360 | **0.87** | 0.88 | | 292230 | **1.02** | 1.58 | | 379899 | **1.45** | 1.68 |

Table 10.1: Quality of pGOMEA and VNS4 on Taillard instances, using the TFT criterion.

| | Best | pGOMEA | IG | | Best | pGOMEA | IG | | Best | pGOMEA | IG |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $50 \times 5$ | 2724 | **0.00** | **0.00** | $50 \times 10$ | 2991 | 1.14 | **1.14** | $50 \times 20$ | 3850 | 1.34 | **1.12** |
| | 2834 | 0.07 | **0.00** | | 2867 | 1.53 | **0.12** | | 3704 | 1.23 | **0.42** |
| | 2621 | **0.00** | **0.00** | | 2839 | 1.06 | **0.46** | | 3640 | 1.77 | **1.02** |
| | 2751 | 0.04 | **0.00** | | 3063 | 0.18 | **0.00** | | 3723 | 1.26 | **0.83** |
| | 2863 | **0.00** | **0.00** | | 2976 | 1.01 | **0.10** | | 3611 | 1.44 | **0.55** |
| | 2829 | **0.00** | **0.00** | | 3006 | 0.50 | **0.00** | | 3681 | 1.40 | **0.60** |
| | 2725 | **0.00** | **0.00** | | 3093 | 0.71 | **0.26** | | 3704 | 1.35 | **0.61** |
| | 2683 | **0.00** | **0.00** | | 3037 | 0.23 | **0.03** | | 3691 | 1.87 | **1.04** |
| | 2552 | 0.08 | **0.00** | | 2897 | 0.28 | **0.17** | | 3743 | 1.51 | **0.81** |
| | 2782 | **0.00** | **0.00** | | 3065 | 0.85 | **0.42** | | 3756 | 1.05 | **0.43** |
| $100 \times 5$ | 5493 | **0.00** | **0.00** | $100 \times 10$ | 5770 | 0.05 | **0.00** | $100 \times 20$ | 6202 | 1.80 | **0.74** |
| | 5268 | 0.13 | **0.00** | | 5349 | 0.24 | **0.00** | | 6183 | 1.52 | **0.49** |
| | 5175 | **0.00** | **0.00** | | 5676 | 0.05 | **0.05** | | 6271 | 1.28 | **0.58** |
| | 5014 | 0.08 | **0.00** | | 5781 | 0.43 | **0.00** | | 6269 | 1.55 | **0.54** |
| | 5250 | 0.02 | **0.00** | | 5467 | 0.53 | **0.00** | | 6314 | 1.60 | **0.78** |
| | 5135 | **0.00** | **0.00** | | 5303 | 0.09 | **0.00** | | 6364 | 1.93 | **0.35** |
| | 5246 | **0.00** | **0.00** | | 5595 | 0.13 | **0.05** | | 6268 | 1.52 | **0.63** |
| | 5094 | **0.00** | **0.00** | | 5617 | 0.41 | **0.11** | | 6401 | 1.87 | **0.77** |
| | 5448 | **0.00** | **0.00** | | 5871 | 0.34 | **0.07** | | 6275 | 1.57 | **0.52** |
| | 5322 | 0.04 | **0.00** | | 5845 | 0.05 | **0.00** | | 6434 | 1.50 | **0.70** |

Table 10.2: Quality of pGOMEA and IG on Taillard instances, using the $C_{max}$ criterion.

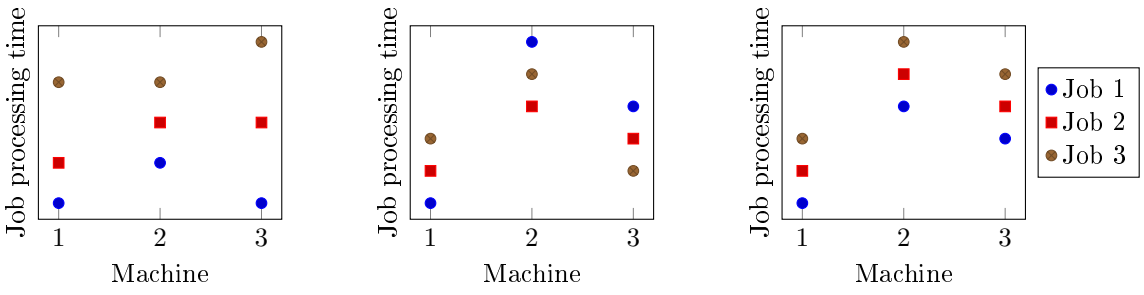
## 10.3 Benchmarking: Existing structured instances

Watson et al. [42] have shown that for the PFSP with the $C_{max}$ criterion the problem structure has a big impact on how easy a problem is solved. The majority of structured problems are easily solved by both simple and complex algorithms. As pGOMEA models structure in order to solve permutation problems, it is expected that these structured instances are particularly suitable to be solved by pGOMEA. Since real-world instances often contain structure, we will experiment with pGOMEA on structured instances to verify its quality in practical situations.

### 10.3.1 Structured instances

Watson et al. use three types of structured instances: Job-correlated, Machine-correlated and Mixed-correlated instances (see Figure 10.2). In job-correlated instances, processing times are dependent on the job and not on the machines, therefore Watson et al. create a distribution per job for sampling processing times. Therefore the processing times of operations in one job are related. In machine-correlated instances the structure goes the other way around. Here, processing times on one machine are related as they are sampled from one distribution. Processing times within one job are unrelated. Mixed-correlated instances are equal to Machine-correlated instances, but here the relative ranks of job processing times are largely independent of the machine.

Every distribution (job/machine) has a range of less than 12 values that can be chosen. Using the alpha-parameters, the distribution means are more (high $\alpha$) or less (low $\alpha$) spread apart. Therefore with $\alpha = 1.0$, distributions hardly overlap, while for $\alpha = 0.0$ the distributions all overlap. Unfortunately $\alpha = 0.0$ does not create Taillard-like instances. Taillard samples from a distribution with range of 99 values with mean 50, Watson uses multiple distributions with a range smaller than 12 having the same random mean in $[1, 99]$.




(a) Job-correlated (b) Machine correlation (c) Mixed correlation

Figure 10.2: Types of structure in PFSP instances.

For the $C_{max}$ criterion, instances with a few machines are easily solved. Therefore Watson et al. only generate instances with twenty machines. Therefore we also limit our experiments to instances with twenty machines. We will experiment both with very structured instances ($\alpha = 1.0$), half structured instances ($\alpha = 0.5$) and slightly structured instances ($\alpha = 0.1$ and $\alpha = 0.2$). Experiments will be performed on the first three problems provided by Watson et al.

### 10.3.2 Solving PFSP with the TFT criterion

For the PFSP with the TFT criterion we have run pGOMEA and VNS4 on the Watson instances. Unfortunately, no results are known about lower bounds and previous results. Therefore the resulting MRPD values as given in Table 10.3 use the best result found by pGOMEA and VNS4 as upper bound. The best results are marked bold and if the other algorithm performs significantly worse, its result is highlighted grey.

In these results, we see that the MRPD values are very low. As our reference point is the best solution found by pGOMEA and VNS4, this does not say very much by itself. However, we also observe very small differences within the results of one algorithm. For the Taillard instances, these differences were sometimes larger than 1000, whereas for the Watson instances this differences are usually smaller than 100. Since this difference goes down when algorithms find solutions close to

the optimum, we can assume that these structured instances are very easy for the TFT criterion. Though these instances are so easy some significant differences were still found. The general trend is that for unstructured instances pGOMEA is best. More correlation makes pGOMEA perform worse than VNS4 (until pGOMEA gives optimal results). The amount of problem structure VNS4 needs to outperforms pGOMEA is higher for larger instances (higher correlation factors are needed).

| | | Job correlated | | | Machine Correlated | | | Mixed correlated | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Best | pGOMEA | VNS4 | Best | pGOMEA | VNS4 | Best | pGOMEA | VNS4 |
| $50 \times 20$ | $\alpha = 0.1$ | 159817 | **0.001** | 0.005 | 145084 | **0.010** | 0.012 | 193977 | **0.001** | 0.002 |
| | | 95140 | 0.008 | **0.003** | 191223 | **0.006** | 0.019 | 43898 | **0.011** | 0.013 |
| | | 33617 | 0.006 | **0.000** | 37324 | 0.042 | **0.035** | 89242 | **0.000** | **0.000** |
| | $\alpha = 0.2$ | 104206 | **0.000** | **0.000** | 104893 | **0.017** | 0.033 | 195884 | **0.000** | **0.000** |
| | | 47081 | **0.000** | **0.000** | 144323 | **0.000** | **0.000** | 57859 | 0.003 | 0.003 |
| | | 171661 | **0.000** | **0.000** | 188073 | 0.012 | **0.008** | 102555 | 0.006 | **0.005** |
| | $\alpha = 0.5$ | 63632 | **0.000** | **0.000** | 134775 | 0.010 | **0.006** | 189823 | **0.000** | **0.000** |
| | | 136888 | **0.000** | **0.000** | 152219 | **0.005** | 0.007 | 131033 | **0.000** | **0.000** |
| | | 103623 | **0.000** | **0.000** | 186151 | **0.000** | **0.000** | 157945 | **0.000** | **0.000** |
| | $\alpha = 1.0$ | 109907 | **0.000** | **0.000** | 153329 | **0.000** | **0.000** | 154222 | **0.000** | **0.000** |
| | | 69177 | **0.000** | **0.000** | 160736 | **0.000** | **0.000** | 172350 | **0.000** | **0.000** |
| | | 95357 | **0.000** | **0.000** | 169213 | **0.000** | **0.000** | 168092 | **0.000** | **0.000** |
| $100 \times 20$ | $\alpha = 0.1$ | 64021 | 0.015 | **0.014** | 589024 | **0.002** | 0.007 | 566746 | **0.004** | 0.006 |
| | | 338828 | 0.029 | **0.013** | 109702 | **0.012** | 0.016 | 97046 | **0.058** | 0.250 |
| | | 611015 | 0.023 | **0.018** | 250483 | 0.010 | **0.003** | 245350 | **0.007** | 0.012 |
| | $\alpha = 0.2$ | 559950 | 0.016 | **0.011** | 582125 | **0.003** | 0.020 | 585337 | **0.003** | 0.021 |
| | | 256739 | **0.018** | 0.020 | 170913 | **0.038** | 0.111 | 153385 | **0.026** | 0.052 |
| | | 499133 | 0.022 | **0.006** | 288881 | **0.007** | 0.025 | 287695 | 0.006 | **0.004** |
| | $\alpha = 0.5$ | 218030 | 0.002 | **0.001** | 608557 | 0.011 | **0.004** | 581895 | **0.001** | 0.001 |
| | | 353962 | **0.001** | **0.001** | 319449 | 0.008 | **0.005** | 289704 | 0.001 | **0.001** |
| | | 173033 | **0.000** | **0.000** | 411090 | 0.014 | **0.010** | 415477 | **0.001** | 0.002 |
| | $\alpha = 1.0$ | 244157 | **0.000** | **0.000** | 509180 | 0.008 | **0.002** | 608588 | 0.018 | **0.005** |
| | | 246583 | **0.000** | **0.000** | 517291 | **0.000** | **0.000** | 565043 | **0.000** | **0.000** |
| | | 328199 | **0.000** | **0.000** | 556120 | **0.000** | **0.000** | 572648 | 0.001 | **0.000** |

Table 10.3: pGOMEA on Watson instances using the TFT criterion.

### 10.3.3 Solving PFSP with the $C_{max}$ criterion

For the PFSP with the $C_{max}$ criterion we have run pGOMEA and IG for each types of correlation and for multiple problem sizes. In Table 10.4 the MRPD values are given for these results, best results are marked bold and if the other algorithm performs significantly worse, its result is highlighted grey. Again both pGOMEA and IG find the best known solution nearly every time. Only for job correlated instances pGOMEA and IG are significantly different. For large instances, pGOMEA works best. Smaller, non-structured (or slightly structured) instances are best solved using IG. Both algorithms however find better results than Watson et al. for these instances. As the results are very close to the lower bounds computed by Watson et al., we can assume that both pGOMEA and IG work very well on these instances.

### 10.3.4 Watson instances: why so easy?

As already mentioned by Watson et al., the structured instances are easier than unstructured instances. This might explain why all algorithms perform this good. However, the algorithms also perform very well on unstructured instances ($\alpha = 0.1$) as shown in Table 10.4. Three things may cause this difference:

- Watson with $\alpha = 0.0$ samples from a uniform distribution of $[mean - x, mean + x]$, where $x$ is sampled from the uniform distribution $[1, 5]$. Taillard instances are however uniformly sampled from $[1, 99]$,

| | | Job correlated | | | Machine Correlated | | | Mixed correlated | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Best | pGOMEA | IG | Best | pGOMEA | IG | Best | pGOMEA | IG |
| $50 \times 20$ | $\alpha = 0.1$ | 5050 | 0.04 | **0.02** | 4540 | **0.00** | **0.00** | 6101 | 0.03 | **0.02** |
| | | 3062 | 0.03 | **0.00** | 5969 | 0.02 | **0.02** | 1420 | **0.00** | **0.00** |
| | | 1147 | 0.17 | **0.00** | 1226 | **0.00** | **0.00** | 2825 | **0.00** | **0.00** |
| | $\alpha = 0.2$ | 3444 | 0.06 | **0.00** | 3314 | **0.00** | **0.00** | 6168 | **0.00** | **0.00** |
| | | 1735 | **0.00** | **0.00** | 4622 | **0.00** | **0.00** | 1939 | **0.00** | **0.00** |
| | | 5582 | **0.02** | **0.02** | 5932 | 0.02 | **0.01** | 3311 | **0.00** | **0.00** |
| | $\alpha = 0.5$ | 2645 | **0.00** | **0.00** | 4465 | **0.00** | **0.00** | 6174 | **0.00** | **0.00** |
| | | 4969 | **0.00** | **0.00** | 5018 | 0.06 | **0.00** | 4337 | **0.00** | **0.00** |
| | | 3821 | **0.00** | **0.00** | 5991 | **0.00** | **0.00** | 5193 | **0.00** | **0.00** |
| | $\alpha = 1.0$ | 4697 | **0.00** | **0.00** | 5195 | **0.00** | **0.00** | 5178 | **0.00** | **0.00** |
| | | 3829 | **0.00** | **0.00** | 5572 | **0.00** | **0.00** | 5808 | **0.00** | **0.00** |
| | | 4430 | **0.00** | **0.00** | 5791 | **0.00** | **0.00** | 5731 | **0.00** | **0.00** |
| $100 \times 20$ | $\alpha = 0.1$ | 1239 | 0.48 | **0.16** | 10170 | **0.00** | **0.00** | 9856 | **0.00** | **0.00** |
| | | 5931 | 0.08 | **0.05** | 2020 | **0.00** | **0.00** | 1824 | **0.00** | **0.00** |
| | | 10604 | 0.05 | **0.02** | 4411 | **0.00** | **0.00** | 4274 | **0.02** | 0.07 |
| | $\alpha = 0.2$ | 10000 | 0.02 | **0.01** | 10150 | **0.00** | **0.00** | 10181 | **0.00** | **0.00** |
| | | 4760 | **0.02** | 0.05 | 3143 | **0.00** | **0.00** | 2811 | **0.00** | **0.00** |
| | | 8909 | 0.04 | **0.04** | 5135 | **0.00** | **0.00** | 5112 | **0.00** | **0.00** |
| | $\alpha = 0.5$ | 4714 | **0.00** | **0.00** | 10832 | **0.00** | **0.00** | 10450 | **0.00** | **0.00** |
| | | 7137 | **0.00** | **0.00** | 5951 | **0.00** | **0.00** | 5257 | **0.00** | **0.00** |
| | | 4005 | **0.00** | 0.02 | 7516 | **0.00** | **0.00** | 7534 | **0.00** | **0.00** |
| | $\alpha = 1.0$ | 6400 | 0.02 | **0.01** | 9380 | **0.00** | 0.01 | 10907 | **0.00** | **0.00** |
| | | 6444 | **0.00** | **0.00** | 9473 | **0.00** | **0.00** | 10402 | **0.00** | **0.00** |
| | | 7512 | **0.00** | **0.00** | 10422 | **0.00** | **0.00** | 10455 | **0.00** | **0.00** |

Table 10.4: pGOMEA on Watson instances using the $C_{max}$ criterion

- The Watson instances have not been researched a lot, therefore best known upper bounds of Watson instances are not very tight,
- Taillard instances are selected as the hardest from a set random instances.

From these reasons, the second one is the least probable, as our results also show that IG or pGOMEA always result in the same local optimum (which is then probably the global optimum). The first reason is most likely to cause this difference, as the values jobs can have are much more limited. Therefore, more solutions with the same fitness exist, leading to more global and local optima that can be found. The third reason can also contribute to our findings, but we expect this to be less important.

Though IG performs slightly worse than pGOMEA and VNS4 performs slightly better than pGOMEA on structured instances, the simplicity of these problems makes that we cannot draw strong conclusions about the effectiveness of pGOMEA on the different types of (semi-)structured instances with larger spread.

## 10.4 Benchmarking: New structured instances

As we couldn't draw strong conclusions using the Watson instances, we introduce a new type of structured instances. Again job-correlated and machine-correlated instances are generated, with a correlation factor $\alpha \in [0, 1]$. With $\alpha = 0.0$ these instances reflect the Taillard instances, except instances are not selected based on difficulty. For $\alpha = 1.0$, jobs or machines are completely correlated meaning that every action on a job or machine has the same processing time. Such structured instances are trivially solved in polynomial time for the $C_{max}$ criterion. Appendix A shows in detail how instances are generated using the alpha-parameter. Using multiple of these instances, we again evaluate the performance of pGOMEA on structured instances.

### 10.4.1 Experimental Setup

For our experiments we consider instances with three different alpha-values, namely 0.2, 0.4 and 0.6. For these instances, the problem is not easily solved, but we can still research high, medium and low correlations between jobs and machines. Our experiments will be performed on instances with 50 and 100 jobs and 20 machines, as these Taillard instances are hard for the $C_{max}$ criterion. For each combination of correlation (type and value) and instance size we test on the first ten instances generated. For the computational time we use $400 \cdot J \cdot M \cdot (1 - \alpha)$ milliseconds. This makes sure that we do not waste time on the easier problems with a lot of structure. In our experiments, we compare pGOMEA pairwise with IG/VNS4. We also compare pGOMEA with pGOMEA using random dependency values. This gives useful insights in the amount of structure pGOMEA is able to learn in (un)structured instances.

### 10.4.2 TFT

For the TFT criterion we tested pGOMEA on multiple structured instances. Table 10.5 shows the results of the algorithms. pGOMEA has been statistically compared with VNS4 and pGOMEA with random dependencies (RAND). When an algorithm performs signifficantly better than pGOMEA, its result is marked bold. Signifficantly worse results are marked grey. Here, we observed that pGOMEA performs better that VNS4 on job correlated and big mixed correlated instances. For (large, high correlation) machine correlated instances and smaller mixed correlated instances, VNS4 is often more effective than pGOMEA. Using random dependencies is however not useful, as it performs better on only one instance.

### 10.4.3 CMAX

For the $C_{max}$ criterion we have compared pGOMEA, IG and random-dependency pGOMEA (RAND), using the new structured instances. Table 10.6 shows the results of the algorithms. pGOMEA has been statistically compared with both other algorithms. When an algorithm performs signifficantly better than pGOMEA, its result marked in bold. Signifficantly worse results are marked grey. In these results we find that IG is still signifficantly better than pGOMEA, though machine and mixed correlated instances are solved easily. Using smart dependencies is more important when problems become harder. This is shown by the signifficant differences in job correlated instances, while the machine and mixed correlated instances are also easily solved with random dependencies.

## 10.5 Conclusions

In this chapter we have compared pGOMEA with two state-of-the-art algorithms and we have examined the influence of problem structure on the (relative) quality of pGOMEA. Here we have found that in a non-structured environment pGOMEA is outperformed by the IG algorithm. The IG algorithm is very good at calculating multiple fitness evaluations at once, in contrast to pGOMEA. As the VNS4 algorithm does not employ any speedups in fitness evaluations, pGOMEA is able to outperform that algorithm. After discarding known structured benchmarks, we used a new benchmark to create structured instances. Using these instances, we were able to observe the quality of pGOMEA when problems get more structure in it. The results have shown that pGOMEA performs better on structured instances. This can however not be fully ascribed to dependency learning. Other algorithms also perform better, as the search space is probably much more structured and the problems are thus less deceptive. We can still conclude that dependency learning is a key part of pGOMEA in structured problems. When pGOMEA uses random dependency values, pGOMEA often performs signifficantly worse.

| | | Job correlated | | | | Machine Correlated | | | | Mixed correlated | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Best | pGOMEA | VNS4 | RAND | Best | pGOMEA | VNS4 | RAND | Best | pGOMEA | VNS4 | RAND |
| 50 × 20 | $\alpha = 0.2$ | 115932 | 0.29 | 0.41 | 0.28 | 121466 | 0.36 | 0.29 | 0.30 | 119556 | 0.29 | 0.59 | 0.49 |
| | | 116443 | 0.30 | 0.62 | 0.40 | 124700 | 0.53 | **0.27** | **0.43** | 124278 | 0.26 | 0.36 | 0.38 |
| | | 121138 | 0.21 | 0.37 | 0.30 | 123541 | 0.44 | 0.38 | 0.42 | 119270 | 0.20 | 0.24 | 0.17 |
| | | 117199 | 0.31 | 0.55 | 0.39 | 120350 | 0.45 | 0.54 | 0.47 | 124036 | 0.22 | 0.16 | 0.26 |
| | | 120609 | 0.17 | 0.47 | 0.32 | 119436 | 0.39 | 0.66 | 0.34 | 120075 | 0.39 | 0.74 | 0.30 |
| | | 118530 | 0.13 | 0.53 | 0.32 | 117507 | 0.42 | 0.51 | 0.31 | 125952 | 0.15 | 0.15 | 0.30 |
| | | 118882 | 0.22 | 0.60 | 0.37 | 120691 | 0.15 | 0.39 | 0.24 | 120729 | 0.14 | 0.18 | 0.20 |
| | | 120485 | 0.32 | 0.57 | 0.38 | 120256 | 0.22 | 0.34 | 0.42 | 121931 | 0.19 | 0.24 | 0.17 |
| | | 115047 | 0.38 | 0.46 | 0.31 | 127320 | 0.34 | 0.38 | 0.40 | 127902 | 0.20 | 0.23 | 0.31 |
| | | 113643 | 0.24 | 0.42 | 0.37 | 125185 | 0.26 | 0.32 | 0.36 | 123188 | 0.24 | 0.40 | 0.28 |
| | $\alpha = 0.4$ | 112233 | 0.17 | 0.35 | 0.14 | 129585 | 0.11 | 0.08 | 0.12 | 132369 | 0.08 | 0.10 | 0.09 |
| | | 104133 | 0.14 | 0.28 | 0.22 | 121772 | 0.10 | 0.19 | 0.20 | 129112 | 0.11 | **0.02** | **0.08** |
| | | 109701 | 0.06 | 0.27 | 0.16 | 125712 | 0.18 | 0.18 | 0.20 | 118197 | 0.04 | 0.20 | 0.11 |
| | | 108567 | 0.26 | 0.26 | 0.31 | 128817 | 0.14 | 0.14 | 0.16 | 128669 | 0.18 | **0.03** | **0.07** |
| | | 114933 | 0.20 | 0.34 | 0.24 | 124572 | 0.16 | 0.19 | 0.22 | 122102 | 0.15 | 0.23 | 0.22 |
| | | 114286 | 0.05 | 0.25 | 0.10 | 126417 | 0.09 | 0.23 | 0.13 | 118722 | 0.16 | 0.44 | 0.27 |
| | | 113312 | 0.11 | 0.23 | 0.18 | 129705 | 0.21 | 0.24 | 0.21 | 123119 | 0.09 | 0.11 | 0.07 |
| | | 111915 | 0.17 | 0.38 | 0.21 | 129424 | 0.06 | 0.10 | 0.13 | 117565 | 0.02 | 0.09 | 0.07 |
| | | 111720 | 0.10 | 0.19 | 0.15 | 134549 | 0.16 | 0.23 | 0.25 | 128085 | 0.09 | 0.07 | 0.08 |
| | | 111452 | 0.13 | 0.27 | 0.15 | 119268 | 0.23 | 0.19 | 0.21 | 129006 | 0.05 | 0.08 | 0.05 |
| | $\alpha = 0.6$ | 103452 | 0.01 | 0.00 | 0.01 | 130768 | 0.08 | 0.07 | 0.11 | 137091 | 0.01 | **0.00** | **0.00** |
| | | 102201 | 0.01 | 0.02 | 0.02 | 132077 | 0.05 | **0.02** | **0.04** | 130445 | 0.00 | 0.00 | 0.00 |
| | | 107023 | 0.00 | 0.00 | 0.00 | 144648 | 0.05 | 0.09 | 0.06 | 141593 | 0.00 | 0.00 | 0.00 |
| | | 94201 | 0.00 | 0.00 | 0.00 | 143034 | 0.02 | 0.05 | 0.05 | 134738 | 0.01 | 0.00 | 0.01 |
| | | 105104 | 0.03 | 0.07 | 0.03 | 122944 | 0.05 | 0.05 | 0.05 | 129048 | 0.00 | 0.00 | 0.00 |
| | | 104715 | 0.02 | 0.06 | 0.05 | 141008 | 0.08 | 0.06 | 0.10 | 141035 | 0.01 | **0.00** | **0.00** |
| | | 95312 | 0.00 | 0.02 | 0.00 | 141875 | 0.06 | 0.06 | 0.07 | 148554 | 0.00 | 0.00 | 0.00 |
| | | 113632 | 0.00 | 0.03 | 0.00 | 135961 | 0.01 | 0.02 | 0.01 | 129213 | 0.00 | 0.00 | 0.00 |
| | | 105767 | 0.00 | 0.00 | 0.00 | 122958 | 0.12 | 0.11 | 0.10 | 133431 | 0.00 | 0.00 | 0.00 |
| | | 103308 | 0.00 | 0.01 | 0.01 | 146136 | 0.06 | 0.07 | 0.09 | 123618 | 0.00 | 0.00 | 0.00 |
| 100 × 20 | $\alpha = 0.2$ | 354483 | 0.42 | 0.46 | 0.53 | 376876 | 0.28 | 0.18 | 0.46 | 392925 | 0.35 | 0.48 | 0.54 |
| | | 352436 | 0.38 | 0.58 | 0.61 | 379781 | 0.31 | 0.42 | 0.45 | 375810 | 0.26 | 0.66 | 0.43 |
| | | 354530 | 0.34 | 0.43 | 0.47 | 395682 | 0.18 | 0.42 | 0.48 | 400652 | 0.14 | 0.36 | 0.27 |
| | | 341043 | 0.26 | 0.52 | 0.55 | 389305 | 0.36 | 0.58 | 0.63 | 376562 | 0.31 | 0.67 | 0.41 |
| | | 354807 | 0.31 | 0.69 | 0.53 | 389688 | 0.25 | 0.59 | 0.45 | 368258 | 0.29 | 0.29 | 0.40 |
| | | 370144 | 0.33 | 0.54 | 0.69 | 360776 | 0.15 | 0.54 | 0.52 | 388375 | 0.33 | 0.63 | 0.59 |
| | | 351676 | 0.32 | 0.62 | 0.50 | 389651 | 0.33 | 0.59 | 0.45 | 379751 | 0.62 | **0.38** | 0.80 |
| | | 372308 | 0.31 | 0.39 | 0.46 | 377689 | 0.38 | 0.35 | 0.44 | 372021 | 0.18 | 0.29 | 0.41 |
| | | 362577 | 0.43 | 0.48 | 0.56 | 385017 | 0.29 | 0.40 | 0.49 | 364454 | 0.29 | 0.29 | 0.53 |
| | | 360703 | 0.39 | 0.57 | 0.46 | 389271 | 0.25 | 0.35 | 0.34 | 349791 | 0.28 | 0.52 | 0.60 |
| | $\alpha = 0.4$ | 338303 | 0.27 | 0.39 | 0.38 | 432943 | 0.25 | **0.20** | 0.26 | 409404 | 0.15 | 0.23 | 0.19 |
| | | 331602 | 0.23 | 0.51 | 0.35 | 404417 | 0.20 | 0.19 | 0.35 | 408221 | 0.11 | 0.11 | 0.18 |
| | | 337449 | 0.14 | 0.23 | 0.30 | 411893 | 0.10 | 0.09 | 0.20 | 381029 | 0.07 | 0.15 | 0.16 |
| | | 343512 | 0.15 | 0.49 | 0.32 | 410707 | 0.26 | **0.16** | 0.40 | 367183 | 0.18 | 0.30 | 0.33 |
| | | 321656 | 0.28 | 0.41 | 0.40 | 437327 | 0.20 | **0.12** | 0.36 | 392645 | 0.12 | 0.33 | 0.31 |
| | | 348395 | 0.23 | 0.53 | 0.38 | 418578 | 0.23 | **0.16** | 0.37 | 428664 | 0.06 | 0.10 | 0.11 |
| | | 350807 | 0.26 | 0.28 | 0.39 | 402707 | 0.14 | 0.13 | 0.22 | 400714 | 0.15 | 0.12 | 0.22 |
| | | 336867 | 0.11 | 0.47 | 0.41 | 419103 | 0.14 | 0.14 | 0.19 | 378965 | 0.16 | 0.28 | 0.26 |
| | | 334469 | 0.16 | 0.44 | 0.27 | 440173 | 0.21 | **0.15** | 0.28 | 411678 | 0.06 | 0.15 | 0.13 |
| | | 341423 | 0.35 | 0.78 | 0.50 | 417045 | 0.13 | **0.08** | 0.24 | 392508 | 0.10 | 0.22 | 0.20 |
| | $\alpha = 0.6$ | 319476 | 0.17 | 0.32 | 0.26 | 491858 | 0.11 | 0.12 | 0.18 | 395660 | 0.03 | 0.06 | 0.05 |
| | | 335018 | 0.13 | 0.30 | 0.21 | 407094 | 0.14 | **0.05** | 0.23 | 454738 | 0.02 | 0.02 | 0.02 |
| | | 311021 | 0.06 | 0.22 | 0.07 | 450581 | 0.04 | 0.02 | 0.04 | 365546 | 0.02 | 0.02 | 0.03 |
| | | 301670 | 0.16 | 0.37 | 0.22 | 391007 | 0.14 | **0.06** | 0.18 | 400760 | 0.00 | 0.00 | 0.00 |
| | | 303487 | 0.10 | 0.21 | 0.16 | 449246 | 0.13 | **0.03** | 0.21 | 447317 | 0.01 | 0.03 | 0.02 |
| | | 299529 | 0.12 | 0.22 | 0.15 | 479164 | 0.12 | 0.14 | 0.19 | 412359 | 0.02 | 0.05 | 0.03 |
| | | 320155 | 0.14 | 0.17 | 0.19 | 469897 | 0.41 | **0.10** | 0.39 | 450060 | 0.02 | 0.02 | 0.03 |
| | | 290573 | 0.15 | 0.27 | 0.26 | 471761 | 0.18 | **0.06** | 0.29 | 484204 | 0.01 | 0.02 | 0.01 |
| | | 305220 | 0.12 | 0.26 | 0.18 | 463159 | 0.14 | **0.10** | 0.18 | 414879 | 0.02 | 0.02 | 0.04 |
| | | 328872 | 0.15 | 0.27 | 0.21 | 453227 | 0.10 | **0.06** | 0.21 | 480739 | 0.02 | 0.02 | 0.02 |

Table 10.5: GOMEA, VNS4 and GOMEA with random dependencies: Structured TFT instances.

| | | Job correlated | | | | Machine Correlated | | | | Mixed correlated | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Best | pGOMEA | IG | RAND | Best | pGOMEA | IG | RAND | Best | pGOMEA | IG | RAND |
| $50 \times 20$ | $\alpha = 0.2$ | 3680 | 0.90 | **0.31** | 0.95 | 3751 | 0.17 | **0.00** | 0.21 | 3877 | 0.05 | **0.00** | 0.05 |
| | | 3636 | 1.13 | **0.52** | 1.39 | 3816 | 0.63 | **0.42** | 0.63 | 3956 | 0.00 | 0.00 | 0.00 |
| | | 3741 | 0.87 | **0.37** | 1.04 | 3810 | 0.75 | **0.00** | 0.81 | 3731 | 0.52 | **0.00** | 0.64 |
| | | 3725 | 0.99 | **0.34** | 1.37 | 3897 | 0.00 | 0.00 | 0.00 | 3989 | 0.45 | **0.00** | 0.45 |
| | | 3761 | 0.72 | **0.40** | 0.94 | 3843 | 1.41 | **0.08** | **0.78** | 3980 | 0.43 | **0.00** | 0.43 |
| | | 3700 | 0.74 | **0.20** | 1.00 | 3720 | 0.00 | 0.00 | 0.09 | 4072 | 0.05 | **0.00** | 0.05 |
| | | 3750 | 0.77 | **0.28** | 0.99 | 3780 | 0.50 | **0.13** | 0.62 | 3970 | 0.05 | **0.00** | 0.05 |
| | | 3778 | 0.73 | **0.25** | 1.10 | 3795 | 0.29 | **0.03** | 0.45 | 3908 | 0.54 | **0.00** | 0.51 |
| | | 3602 | 0.78 | **0.36** | 1.19 | 3968 | 0.68 | **0.00** | 0.68 | 3973 | 0.91 | **0.30** | 1.06 |
| | | 3611 | 1.19 | **0.22** | 1.40 | 3960 | 0.48 | **0.23** | 0.48 | 3881 | 1.16 | **0.30** | 1.16 |
| | $\alpha = 0.4$ | 3639 | 0.58 | **0.30** | 0.74 | 4346 | 0.00 | 0.00 | 0.00 | 4350 | 0.45 | **0.10** | 0.46 |
| | | 3481 | 0.82 | **0.16** | 1.03 | 4087 | 0.00 | 0.00 | 0.00 | 4303 | 0.00 | 0.00 | 0.00 |
| | | 3590 | 0.43 | **0.06** | 0.52 | 4155 | 0.00 | 0.00 | 0.00 | 4139 | 0.17 | **0.00** | 0.19 |
| | | 3565 | 0.50 | **0.15** | 0.67 | 4343 | 0.00 | 0.00 | 0.00 | 4243 | 0.05 | **0.00** | 0.26 |
| | | 3770 | 0.82 | **0.32** | 0.93 | 4080 | 0.42 | **0.00** | **0.23** | 4135 | 0.00 | 0.00 | 0.00 |
| | | 3719 | 0.42 | **0.16** | 0.62 | 4269 | 0.00 | 0.00 | 0.00 | 4149 | 0.00 | 0.00 | 0.00 |
| | | 3734 | 0.48 | **0.13** | 0.55 | 4353 | 0.00 | 0.00 | 0.00 | 4308 | 0.07 | **0.00** | **0.00** |
| | | 3626 | 0.65 | **0.32** | 0.85 | 4312 | 0.00 | 0.00 | 0.00 | 4070 | 0.12 | **0.00** | **0.05** |
| | | 3719 | 0.51 | **0.17** | 0.66 | 4459 | 0.06 | **0.00** | **0.00** | 4389 | 0.05 | **0.00** | 0.05 |
| | | 3761 | 0.56 | **0.19** | 0.82 | 4050 | 0.00 | 0.00 | 0.00 | 4228 | 0.05 | **0.00** | 0.05 |
| | $\alpha = 0.6$ | 3806 | 0.17 | **0.03** | 0.16 | 4335 | 0.00 | 0.00 | 0.00 | 4710 | 0.00 | 0.00 | 0.00 |
| | | 3785 | 0.29 | **0.11** | 0.40 | 4664 | 0.00 | 0.00 | 0.00 | 4645 | 0.00 | 0.00 | 0.00 |
| | | 3907 | 0.10 | **0.03** | 0.13 | 4867 | 0.00 | 0.00 | 0.00 | 4999 | 0.00 | 0.00 | 0.00 |
| | | 3575 | 0.14 | **0.08** | 0.14 | 4814 | 0.00 | 0.00 | 0.00 | 4706 | 0.02 | **0.00** | 0.02 |
| | | 3764 | 0.15 | **0.00** | 0.13 | 4399 | 0.00 | 0.00 | 0.00 | 4563 | 0.00 | 0.00 | 0.00 |
| | | 3925 | 0.19 | **0.08** | 0.25 | 4812 | 0.00 | 0.00 | 0.00 | 4637 | 0.00 | 0.00 | 0.00 |
| | | 3583 | 0.39 | **0.11** | 0.31 | 4789 | 0.00 | 0.00 | 0.00 | 4897 | 0.10 | **0.00** | **0.08** |
| | | 3976 | 0.25 | **0.13** | 0.26 | 4725 | 0.00 | 0.00 | 0.00 | 4481 | 0.00 | 0.00 | 0.00 |
| | | 3940 | 0.30 | **0.13** | 0.28 | 4112 | 0.00 | 0.00 | 0.00 | 4610 | 0.00 | 0.00 | 0.02 |
| | | 3691 | 0.24 | **0.11** | 0.23 | 4858 | 0.00 | 0.00 | 0.00 | 4244 | 0.00 | 0.00 | 0.00 |
| $100 \times 20$ | $\alpha = 0.2$ | 6090 | 1.08 | **0.39** | 1.35 | 6793 | 0.00 | 0.00 | 0.00 | 7003 | 0.00 | 0.00 | 0.00 |
| | | 6068 | 0.87 | **0.12** | 1.16 | 6891 | 0.09 | **0.00** | 0.09 | 6656 | 0.32 | **0.14** | 0.31 |
| | | 6060 | 1.26 | **0.26** | 1.42 | 6909 | 0.39 | **0.14** | 0.39 | 7166 | 0.31 | **0.08** | 0.31 |
| | | 5879 | 1.05 | **0.48** | 1.22 | 6849 | 0.66 | **0.09** | 0.66 | 6824 | 0.10 | **0.00** | 0.10 |
| | | 6178 | 0.80 | **0.35** | 1.06 | 6665 | 0.70 | **0.41** | 0.74 | 6596 | 0.15 | **0.00** | 0.11 |
| | | 6330 | 0.98 | **0.40** | 1.25 | 6297 | 0.30 | 0.30 | 0.30 | 6839 | 0.12 | **0.00** | 0.19 |
| | | 6072 | 1.00 | **0.14** | 1.22 | 7215 | 0.00 | 0.00 | 0.00 | 6593 | 0.46 | **0.06** | 0.48 |
| | | 6334 | 0.99 | **0.24** | 1.04 | 6766 | 0.09 | **0.00** | 0.09 | 6731 | 0.34 | 0.34 | 0.34 |
| | | 6155 | 0.87 | **0.20** | 1.06 | 7038 | 0.09 | **0.00** | 0.09 | 6368 | 0.56 | **0.19** | 0.73 |
| | | 6214 | 1.09 | **0.14** | 1.29 | 6999 | 0.00 | 0.00 | 0.00 | 6507 | 0.41 | 0.41 | 0.41 |
| | $\alpha = 0.4$ | 6120 | 0.83 | **0.38** | 0.96 | 7786 | 0.01 | **0.00** | **0.00** | 7761 | 0.12 | **0.00** | 0.12 |
| | | 6038 | 0.68 | **0.20** | 0.82 | 7266 | 0.00 | 0.00 | 0.00 | 7627 | 0.09 | **0.07** | 0.09 |
| | | 6118 | 0.42 | **0.26** | 0.56 | 7739 | 0.00 | 0.00 | 0.08 | 7413 | 0.00 | 0.00 | 0.00 |
| | | 6361 | 0.75 | **0.20** | 0.83 | 7723 | 0.00 | 0.00 | 0.00 | 6865 | 0.20 | **0.13** | 0.20 |
| | | 5898 | 0.46 | **0.35** | 0.68 | 7912 | 0.00 | 0.00 | 0.00 | 7387 | 0.09 | **0.09** | **0.09** |
| | | 6296 | 0.79 | **0.25** | 0.85 | 7842 | 0.00 | 0.00 | 0.00 | 8111 | 0.05 | **0.00** | 0.05 |
| | | 6360 | 0.85 | **0.31** | 1.00 | 7754 | 0.00 | 0.00 | 0.00 | 7381 | 0.09 | **0.00** | 0.09 |
| | | 6173 | 0.65 | **0.18** | 0.87 | 7848 | 0.00 | 0.00 | 0.00 | 7302 | 0.08 | **0.00** | 0.08 |
| | | 6146 | 0.82 | **0.28** | 0.97 | 8145 | 0.00 | 0.00 | 0.00 | 7723 | 0.04 | **0.00** | 0.04 |
| | | 6329 | 0.80 | **0.38** | 0.93 | 7524 | 0.00 | 0.00 | 0.03 | 7373 | 0.03 | **0.00** | 0.03 |
| | $\alpha = 0.6$ | 6175 | 0.59 | **0.25** | 0.57 | 9027 | 0.00 | 0.00 | 0.00 | 7723 | 0.00 | 0.00 | 0.00 |
| | | 6611 | 0.60 | **0.20** | 0.57 | 7507 | 0.00 | 0.00 | 0.00 | 8528 | 0.06 | **0.01** | 0.04 |
| | | 6237 | 0.65 | **0.22** | 0.71 | 8741 | 0.00 | 0.00 | 0.00 | 7117 | 0.00 | 0.00 | 0.00 |
| | | 6032 | 0.46 | **0.14** | 0.46 | 7369 | 0.00 | 0.00 | 0.00 | 7872 | 0.00 | 0.00 | 0.00 |
| | | 6120 | 0.40 | **0.15** | 0.49 | 8145 | 0.00 | 0.00 | 0.00 | 8384 | 0.00 | 0.00 | 0.00 |
| | | 6047 | 0.49 | **0.17** | 0.51 | 8772 | 0.00 | 0.00 | 0.00 | 7904 | 0.00 | 0.00 | 0.00 |
| | | 6381 | 0.35 | **0.16** | 0.48 | 8842 | 0.00 | 0.00 | 0.00 | 8486 | 0.00 | **0.00** | 0.00 |
| | | 5789 | 0.57 | **0.17** | 0.58 | 8820 | 0.00 | 0.00 | 0.00 | 8835 | 0.08 | **0.02** | **0.07** |
| | | 6187 | 0.34 | **0.09** | 0.44 | 8519 | 0.01 | **0.00** | **0.00** | 7927 | 0.00 | 0.00 | 0.00 |
| | | 6286 | 0.56 | **0.10** | 0.64 | 8303 | 0.00 | 0.00 | 0.00 | 8825 | 0.08 | 0.08 | 0.08 |

Table 10.6: GOMEA, IG and GOMEA with random dependencies: Structured $C_{max}$ instances.

# Chapter 11

# Conclusions

This research has shown the interaction of pGOMEA with domain knowledge and vice versa. The research question *How can domain-knowlege of permutation problems be exploited (using heuristics) in the permutation-based Gene-pool Optimal Mixing Evolutionary Algorithm?* has been extensively answered. Different forms of seeding and hybridization have been discussed. In this chapter we give a global overview of the results. In our discussion we will interpret these results of pGOMEA on the PFSP for the broader perspective of permutation problems and GOMEA in general. Using the results of our experiments, we will give some practical recommendations for using pGOMEA in combination with domain knowledge in Section 11.2. We finally conclude this thesis with some recommendations for future work concerning pGOMEA and using domain knowledge in Section 11.3.

## 11.1 Summary

In recent research, GOMEA has been extended to work for permutation problems. This new pGOMEA algorithm uses a random-key encoding to ensure correct crossovers. A linkage tree is build using dependencies-values based on random-key order and proximity. This new pGOMEA algorithm has shown to be effective on the PFSP with the TFT criterion. In this problem, the optimal processing order of $J$ jobs processed on $M$ machines should be found. In experiments it outperforms an other recently proposed permutation estimation of distribution algorithm.
The good performance of pGOMEA is surprising as it does not incorporate any form of domain knowledge. Therefore pGOMEA might be able to compete with state-of-the-art algorithms when incorporating domain knowledge by itself. Domain knowledge comes in many forms. Mostly, domain knowledge is incorporated using constructive or improvement heuristics. Multiple heuristics exist for the two main objectives of the PFSP: the TFT and $C_{max}$ criterion. The RZ constructive heuristic can be used to generate one good solution for the PFSP with the TFT criterion, while the Palmer heuristic uses a similar approach to generate a good solution for the PFSP with the $C_{max}$ criterion. For this criterion the most used constructive heuristic is the NEH-heuristic, which builds a solution by repeatedly inserting a job in the best suitable place in the permutation. These single-solution constructive heuristics can be used to seed one good solution to the populations or forced improvement phase of pGOMEA. Unfortunately this approach steers pGOMEA too much towards this one solution, which leads to a bias that can possibly decrease the quality of pGOMEA.

Using multi-solution constructive heuristics we can seed more solutions in the populations of pGOMEA. For the PFSP with the TFT criterion we can use the LR heuristic, which can generate up to $J$ different good solutions. For the $C_{max}$ criterion the CDS heuristic which generates up to $M-1$ solutions can be used. For both criteria multi-solution seeding improves pGOMEA, the more good solutions are added to the population, the better pGOMEA performs. The costs of generating the seeds should however be taken into account. Seeding with heuristically improved random solutions is therefore too expensive.

The improvement heuristics can however be used for hybridizing pGOMEA. Experiments show that the improvement heuristics are best applied when pGOMEA has changed a solution. This gives the best effect on both fitness and convergence. Most improvement heuristics for permuta-

tion problems are based on simple swap and insertion heuristics. These heuristics can be applied with a limit on the improvement depth or a limit on the probability of applying the improvement heuristic. These forms of hybridization however do not give better results. Only for the TFT criterion and instances with a few machines, the swap heuristic can be useful.
Though the simple insertion and swap heuristic cannot easily improve pGOMEA, they do not use any domain knowledge. More advanced improvement heuristics have therefore been tested. The advanced Cut-and-Repair heuristic was clearly not improving pGOMEA for the TFT criterion, as it still uses little domain knowledge and expensive fitness evaluations. The NEH improvement heuristic for the $C_{max}$ criterion can however improve solutions much faster than the regular insertion heuristic. Using this heuristic in pGOMEA gives better results as less time is needed for a population to convergence.

When no improvement or constructive heuristics are known for a permutation problem, pGOMEA can still profit from domain knowledge. In Chapter 8 we have introduced dependency seeding. Dependency seeding enhances the proximity and relative ordering dependency with a third factor derived from domain knowledge. In our experiments we have firstly seen that the influence of domain knowledge should be carefully determined. Using a constant factor determining the domain knowledge influence is not sufficient. A better approach is to use a cooling scheme for the amount of domain knowledge influence. This leads to a lot of domain knowledge in early generations and a lot of model knowledge in later generations. The use of a cooling scheme needs parameters for the initial influence and the cooling factor. For such a scheme it is not very hard to find values for the parameters such that pGOMEA is improved. Finding the optimal parameters can however be more difficult.
Secondly, our experiments show that dependency seeding needs the right type of seeded dependency values. The RZ and Palmer heuristic slope indices were able to improve pGOMEA. However pGOMEA could not be improved by adding dependency-values from a constructive population.

As a last form of improving pGOMEA, we have experimented with substructural neighborhoods. pGOMEA cannot be improved by hybridization with a substructural neighborhood searcher. Substructural neighborhood searchers also do not outperform simple improvement heuristics, as the local optima of simple improvement heuristics seem equally good as the local optima of the substructural neighborhood searcher.

Finally, we have researched the quality of pGOMEA on instances with varying sizes and structure. For the $C_{max}$ criterion pGOMEA was outperformed by the IG algorithm on the well-known Taillard benchmark set. For the same instances, pGOMEA performs better than the VNS4 algorithm for the TFT criterion. As structured instances by Watson et al. were too easily solved, we experimented on newly generated structured instances. For these instances we observed similar results as for Taillard instances, since other algorithms also profit from structure in the instances. A difference can however be found in mixed-correlated instances, where VNS4 suddenly outperformed pGOMEA on large structured instances. This indicates that the dependendency values of pGOMEA model job-correlation the best. In our tests on structured instances we have also examined the difference between pGOMEA and pGOMEA using random dependency values. Here, we have seen that the choice of dependency value does certainly matter, as standard pGOMEA performs better than pGOMEA with random dependencies.

## 11.2 Recommendation

Though we have found some useful ways of enhancing pGOMEA, the question remains whether our observations apply for each type of permutation problem. For example: is population seeding still effective when pGOMEA is used to solve the traveling salesman problem? And how should we apply dependency seeding for this problem if multi-solution seeding cannot be achieved? Though some of these questions need further research, we can give some guidelines from the results we have achieved on the PFSP.

For population seeding we have seen that the more seeds are given, the better pGOMEA performs. Seeding a single solution can however misguide pGOMEA. When finding a multi-solution constructive heuristic one should keep in mind that this can also misguide pGOMEA. When the

generated permutations only differ slightly, pGOMEA will find a group of strong dependencies that might limit the explorative behaviour of pGOMEA. One should therefore make sure that the used constructive heuristic does not result in strongly dependent permutations. If this is not possible one might prove or show that the structure from constructive heuristic never misguides the search, this also enables one to use single-solution seeding. A third option is to use larger population sizes, but this comes at the cost of more computation time and slower convergence (and pGOMEA uses a population sizing scheme). When using population seeding one should always balance the time spend by the constructive heuristic with the amount of time spend in pGOMEA. At some point one can better explore using the existing seeds, than create more seeds that do not really add much information.

If population seeding is not possible, one might consider using dependency seeding. When using dependency seeding, one should make sure that the dependency values are in balance. Heuristic dependency seeding for instance has failed because the dependency values had some extreme dependency values that do not reflect the real dependency between two variables. One is advised to use dependency values that closely reflect the actual dependency between variables e.g. the distance between two cities in the traveling salesman problem.

The use of improvement heuristics in pGOMEA is harder to give any recommendations for. The quality of hybridization depends on the quality of the available improvement heuristics and the properties of the instance. We have for instance seen that the quality of the swap heuristic is very dependent on the amount of machines and jobs in the problem. For the TFT criterion the swap heuristic could improve pGOMEA for some instances, while it did not improve pGOMEA for the $C_{max}$ criterion. One is advised to look both at heuristics with a high probability of improving a solution and to heuristics with a large average improvement. Though this second type intuitively sounds the best, the first type performs better for the PFSP with the TFT criterion. A second advise is to use local search only if a solution has changed. This gives the best results with respect to the convergence of pGOMEA, even if the local search is limited by a probability of improvement.

The type of problems that are best solved using pGOMEA, are problems or instances with a fitness-landscape containing a lot of global optima. This can be seen by the results on PFSP-$C_{max}$ instances and by the results of pGOMEA on the simpler Watson-instances. These problems however seem easier in general. As pGOMEA is not outperformed on problems without domain-specific local search, we recommend using pGOMEA for problems without such local searchers. When domain-specific local search is present, one can either use this local searcher in a metaheuristic or in pGOMEA. For the $C_{max}$ criterion we have however seen that this pGOMEA is not always better, though it is a parameter-free algorithm finding solutions with a decent quality.

## 11.3 Future work

This research has mainly focussed on enhancing pGOMEA with domain knowledge. As little research has been done on the convergence behavior and dependency-influence on pGOMEA, most conclusions are limited to pGOMEA for the PFSP criterion. The conclusions can only be used as a rule of thumb when applying pGOMEA to other permutation problems. The (aggregated) influence of both individual dependency measures should be researched on benchmark problems with different types of structure in the fitness function (relative order/proximity). On top of this research, the (parameter) influence and types of dependency-seeds can be analyzed to answer questions like: *What is the effect of over/under-estimating dependency-seeds on the quality of the linkage tree?* and *What effect has noise in dependency-values on the explorative behavior of pGOMEA?*. These questions are also not answered for standard GOMEA yet, while this might give new insights in the behavior of pGOMEA.

One can also further research the effect of insertion-behavior of the rescaling operator. Can this be completely replaced with an insertion local searcher, or is the scaling effect also necessary? Such fundamental questions about pGOMEA might best be answered by analyzing the performance on permutation problems with a known optimum (e.g. sorting or polynomial solvable machine scheduling problems).

Besides a fundamental research, more practical research can be performed. For instance the quality of pGOMEA on the TSP can be analyzed. Here, dependency seeding might be applied with the distance between cities as dependency values. Secondly one can try to solve real-world problems or instances using pGOMEA, this will also give more insight about pGOMEA's quality than artificially structured problems.

# Appendices

# Appendix A

# New Structured PFSP benchmark

In this appendix, we introduce a new way of generating structured PFSP instances. First we define the requirements of our PFSP generator in Section A.1. In Section A.2 we introduce an algorithm generating PFSP instances meeting the specified requirements. Finally, we review the properties of the generated instances from a theoretical and practical viewpoint in Sections A.3 and A.4

## A.1 Requirements

For the generated instances, we have multiple strong requirements:

- Job and machine correlated instances can be generated,
- The amount of correlation should be tunable using a parameter $\alpha \in [0, 1]$,
- Instances with any amount of jobs and machines can be generated,
- Instances can be generated with processing times between two specified bounds: $p_{lb}$ and $p_{ub}$,
- Instances with $\alpha = 0.0, p_{lb} = 1$ and $p_{ub} = 99$ should be generated in the way Taillard instances are generated,
- For instances with $\alpha = 1.0$, processing times for the same machine/job should be equal.

Also we have some secondary requirements, which we would like to meet:

- For every $\alpha \in [0, 1]$ any value between $p_{lb}$ and $p_{ub}$ should be equally likely selected for every processing time,
- Mixed correlated instances can be generated.

## A.2 Details

Our algorithm for generating structured instances contains the following input parameters:

- The instance size: $n$ jobs and $m$ machines,
- The bounds of processing times: $[p_{lb}, p_{ub}]$,
- The correlation type: job-correlated (JC) machine-correlated (MC) or mixed-correlated (MXC)
- The correlation factor: $\alpha \in [0, 1]$.

Our algorithm for generating job-correlated instances follows the same idea as Watson et al. For every job $j$, we define a distribution $D_j$ from which its processing times are sampled. Here, every job has the same size of its distribution. The processing times are now sampled in three steps using the uniform distribution $U(min, max)$:

1. **Determine distributions width:**
   First, we define the distribution half-widths of all these distributions as: $HW = \frac{(1-\alpha)\cdot(p_{ub}-p_{lb})}{2}$. This results in a value in $[0, \frac{(p_{ub}-p_{lb})}{2}]$.

2. **Determine distributions means:**
   Secondly, we sample a mean $\mu_j$ for every distribution $D_j$. This mean is sampled from $U(p_{lb} + HW, p_{ub} - HW])$.

3. **Determine processing times:**
   For any processing time of job $j$, a random value is sampled from $U(\mu_j - HW, \mu_j + HW)$.

For machine-correlated instances, we use the same process with one distribution per machine instead of one distribution per job.

For mixed-correlated instances, we first sample processing times using machine-correlated sampling. Then, the processing times $p(i,j)$ within each machine $i$ are re-assigned to the jobs. We do this using the processing times $p'(i,j)$ of the job-dependent sampling method. The largest processing time $p(i,j)$ is assigned to the job $k$ with the largest processing time $p'(i,k)$. This way of sampling can again be described in three steps:

1. **Machine processing times sampling:**
   Sample the processing times for all machines using machine-correlated sampling

2. **Job order-sampling:**
   Sample processing times using job-correlated sampling

3. **Re-assigning machine processing times:**
   Re-assign the the processing times for each machine based on the ranks of the processing times found in step 2).

In step 1) the processing times are sampled using the same parameters as given for the mixed-correlated sampling. In step 2) sampling is done with the same correlation factor and with processing times in $[0, 1]$, while allowing non-discrete processing times and distribution widths.

## A.3 Properties of the generated instances: theory

### A.3.1 Job- and machine-correlated instances

For job- and machine-correlated instances, we can easily prove the following statements:

- **Instances with $\alpha = 0.0, p_{lb} = 1$ and $p_{ub} = 99$ should be generated in the way Taillard instances are generated:**
  Filling in the values gives $HW = 49$ and $\mu_j \in [50, 50] = 50$. Therefore every processing time is sampled from the distribution $U(1, 99)$, like for Taillards instances.

- **For instances with $\alpha = 1.0$, processing times for the same machine/job should be equal:**
  As $HW = 0$, the means are sampled from a uniform distribution of all possible processing times. The processing times do not differ from this mean as we sample the processing times from $U(\mu_j, \mu_j)$.

- **Processing times are exactly within the bounds:**
  The means are sampled from $U(p_{lb} + HW, p_{ub} - HW)$. In the ultimate case, we sample a mean from the end of this distribution. Then, we can get a value of at most $HW$ higher or lower, resulting in the value $p_{lb}$ or $p_{hb}$.

### A.3.2 Mixed correlated instances

For mixed-correlated instances we can also prove these statements:

- **Instances with $\alpha = 0.0, p_{lb} = 1$ and $p_{ub} = 99$ should be generated in the way Taillard instances are generated:**
  Filling in the values gives $HW = 49$ and $\mu_j \in [50, 50] = 50$. Therefore every initial processing time is sampled from the distribution $[1, 99]$, like for Taillards instances. Re-assignment within machines is not biased, as the ranks are based on random processing times generated using Taillard method (job-correlation with $\alpha = 0.0$).

- **For instances with $\alpha = 1.0$, processing times for the same machine should be equal:**
  This argument is equal to that for machine-correlated instances. Re-assignment does not change any value.

- **Processing times are exactly within the bounds:**
  This argument is equal to that for machine-correlated instances. Re-assignment does not change any value.

### A.3.3 Processing time distribution

For $\alpha = 0.0$ and $\alpha = 1.0$ we have seen that a processing time is in fact the result of one uniform distribution. For correlation-values in between, we do not have this property. For these values we observe that low and high processing times are less likely to occur. Figure A.1 shows how these probabilities are distributed for the standard bounds $p_{lb} = 1$ and $p_{ub} = 99$, the distribution for $\alpha$ is equal to the distribution for $1 - \alpha$. This effect of the $\alpha$ parameter should be kept in mind when using this benchmark.

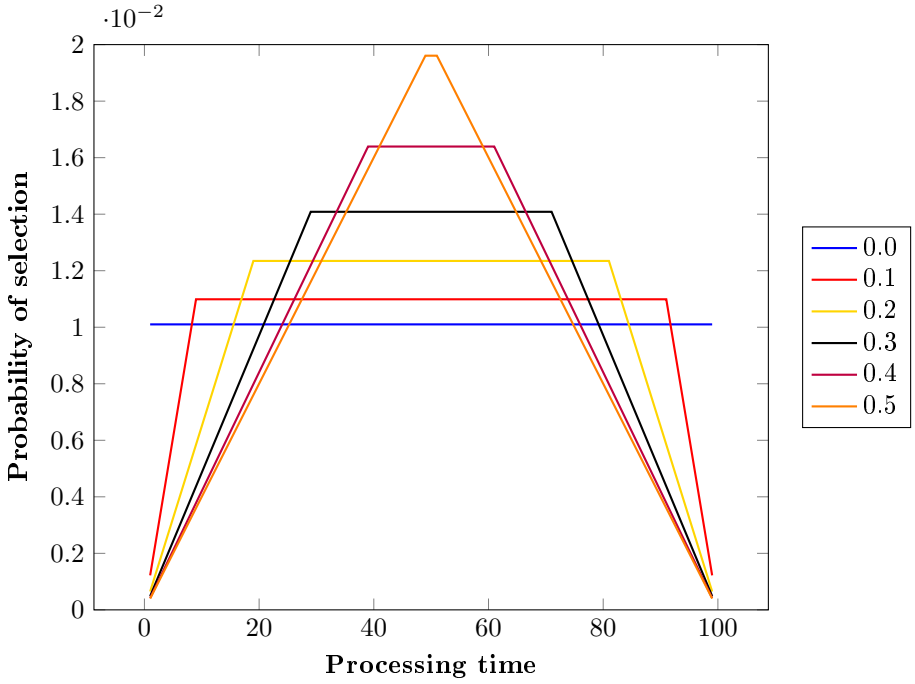



Figure A.1: Probability of selecting a processing time for different $\alpha$.

## A.4 Properties of the generated instances: practice

### A.4.1 Lower bounds and Upper bounds

In order to get insight in the difficulty of structured problems, we generated multiple instances with sizes $(50 \times 20)$, $(100 \times 20)$ and $(200 \times 20)$, using different alpha-values. For each of these instances we calculate two lower bounds for the $C_{max}$ citerion (using Taillard method [35] and a Proportionate method [42]). We also run IG and pGOMEA once on each instance for only $J \cdot 100$ milliseconds to get an upper bound. Using the distance between the upper and lower bounds we can

| | $\alpha$ | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Correlation type | 0.0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 |
| $50 \times 20$ problems | | | | | | | | | | | |
| Job | 318 | 319 | 355 | 362 | 286 | 183 | 98 | 44 | 10 | 0 | 0 |
| Machine | 316 | 225 | 122 | 82 | 43 | 29 | 22 | 13 | 3 | 1 | 0 |
| Mixed | 307 | 246 | 168 | 131 | 109 | 84 | 95 | 73 | 26 | 10 | 0 |
| $100 \times 20$ problems | | | | | | | | | | | |
| Job | 270 | 280 | 322 | 320 | 421 | 449 | 295 | 142 | 49 | 1 | 0 |
| Machine | 279 | 151 | 91 | 60 | 31 | 22 | 11 | 9 | 2 | 0 | 0 |
| Mixed | 285 | 172 | 118 | 79 | 42 | 28 | 16 | 22 | 10 | 2 | 0 |
| $200 \times 20$ problems | | | | | | | | | | | |
| Job | 214 | 217 | 248 | 275 | 324 | 384 | 502 | 448 | 190 | 24 | 0 |
| Machine | 208 | 116 | 54 | 30 | 27 | 17 | 12 | 7 | 3 | 0 | 0 |
| Mixed | 227 | 134 | 78 | 52 | 26 | 15 | 5 | 1 | 2 | 2 | 0 |

Table A.1: Average distance between lower and upper bound.

get an intuition about the difficulty of the problem instances. Table A.1 shows the average distance between the upper and lower bound over fifty runs, using different instances and correlation-factors.

For machine-correlated instances, we see a rapid non-linear decrease in the distance between the two bounds. We can therefore conclude that machine-correlated instances get easier when more structure is added. The difference in distance might however also be caused by the improved quality of the lower bound (here, mostly the Taillard-bound).

For the job-correlated instances, we see how the difference between the bounds initially increases as structure is added. The moment of decrease depends on the size of the problem. As already mentioned by Watson et al., the Taillard-bound does not give a good approximation for job-correlated instances. The difference in upper and lower bound can possibly be caused by the Taillard-bound performing badly on these instances. One should take this into account before labeling job-correlated instances as harder than machine-correlated instances.

# Appendix B

# Paper draft

## Heuristics in permutation GOMEA for solving the permutation flowshop scheduling problem with the total flowtime criterion.

Gerben Aalvanger

July 7, 2017

**Abstract**

The recently introduced permutation Gene-pool Optimal Mixing Evolutionary Algorithm (pGOMEA) has shown to be an effective Model Based Evolutionary Algorithm (MBEA) for permutation problems. So far, pGOMEA has only been used in the context of Black-Box Optimization (BBO). This paper firstly shows that pGOMEA can be greatly improved by incorporating constructive heuristics that seed the population. Secondly the paper shows whether pGOMEA can be improved using hybridization with improvement heuristics. The improved pGOMEA is compared to state-of-the-art algorithms solving the PFSP. Both unstructured and structured instances are used in the benchmarks. The results show that pGOMEA outperforms the VNS4 algorithm for the PFSP with the Total Flowtime criterion.

### 1 Introduction

Recently, Bosman et al. [2] introduced permutation GOMEA (pGOMEA), a Model Based Evolutionary Algorithm (MBEA) which is able to solve permutation problems from a Black-Box Optimization (BBO) perspective. pGOMEA has been tested on the Permutation Flowshop Scheduling Problem (PFSP) with the total flowtime (TFT) criterion. In these tests, pGOMEA outperformed GM-EDA [3] an other permutation MBEA. In order to improve pGOMEA further, we should shift from a BBO perspective to a White-Box perspective. In this paper we show the effect of adding constructive and improvement heuristics to pGOMEA. The experiments are an selection and extension of the experiments in the thesis of Aalvanger [1].

In Section 2 we shortly introduce pGOMEA. After this we explain the PFSP and benchmark instances and performance measures in Section 3. Constructive heuristics for the PFSP are given in Paragraph 4.1, along with some experiments on the effectiveness of these heuristics. In Paragraph 4.3 we do the same for improvement heuristics for the PFSP. Finally, we compare the enhanced pGOMEA with VNS4, a state-of-the-art algorithm solving the PFSP in Section 5. Section 6 concludes this paper with some conclusions and final remarks.

### 2 Permutation GOMEA

#### 2.1 Solution and Model Encoding

pGOMEA encodes solutions using a random-key encoding. An $n-$variable permutation is encoded as $r = (r_1, \cdots r_n)$, where each random key $r_i \in [0, 1]$. The position of variable $i$ in the permutation is equal to the position of $r_i$ when $r$ is sorted in ascending order. Multiple random key encodings can encode the same permutation. For example, $r_1 = (0.34, 0.56, 0.21)$ and $r_2 = (0.72, 0.93, 0.12)$ both encode $x = (1, 2, 0)$.

#### 2.2 Model building

The model used in pGOMEA is a linkage tree. The root of the linkage tree is a set with all variables. Each node is recursively split up, ending in leaves containing only a single variable. Variables grouped in a node are supposed to be dependent, so optimal mixing can improve solutions

effectively.
In pGOMEA, the linkage tree is build for each generation, by merging nodes starting at the bottom of the tree. The two sets $i$ and $j$ are merged which have the strongest dependency $\delta(I, J)$. For two variables $i$ and $j$, the dependency is composed of two factors: $\delta(i, j) = \delta_1(i, j) \cdot \delta_2(i, j)$. The first dependency factor is based on relative-ordering information in the population and is calculated using the entropy of the probability that variable $i$ is before variable $j$ in the population:

$$\delta_1(i, j) = 1 - Entropy(p_{i,j}). \tag{1}$$

The second dependency factor uses the average squared distance in random key values of variable $i$ and $j$:

$$\delta_2(i, j) = 1 - \frac{1}{n} \sum_{k=0}^{n-1} (r_i^k - r_j^k)^2. \tag{2}$$

This results in a symmetric dependency measure between two variables, where high values indicate a high dependency. We can extend the dependency measure to calculate the dependency between two sets, by taking the average pairwise dependency of the variables in the sets:

$$\delta(I, J) = \frac{1}{|I| \cdot |J|} \sum_{i \in I} \sum_{j \in J} \delta(i, j). \tag{3}$$

### 2.3 Optimal Mixing

Using the sets in the linkage tree pGOMEA improves the solutions in a population, therefore pGOMEA uses Gene-pool Optimal Mixing (GOM). For each solution, pGOMEA takes every set in the linkage tree as a crossover mask. The values of the masked variables are then substituted by values from a random donor solution. For example, solution $r_1 = (0.2, 0.3, 0.6, 0.5)$ is changed using crossover mask $(x_1, x_2, x_4)$ and donor $r_2 = (0.9, 0.5, 0.1, 0.7)$ to $r_1' = (\mathbf{0.9}, \mathbf{0.5}, 0.6, \mathbf{0.7})$. If such a change is not strictly improving a solution, the substitution is reverted. Thanks to the random keys encoding, optimal mixing always results in a feasible permutation.

If a solution is not improved using any crossover mask, pGOMEA will 'force' improvements using the best known solution so far. In this Forced Improvement (FI) phase, pGOMEA repeats optimal mixing but the best known solution is used as donor, instead of a random one. In order to improve convergence changes are accepted when they do not decrease the quality of the solution. For the same reason, Forced improvement is also entered if the best overall solution has not changed for $NIS = 10 + 10 \cdot \log n$ generations.

With a probability of 0.1, pGOMEA will 'scale' the random keys before substitution. Here, the values to substitute are scaled to a new interval. For example, scaling random keys $(0.9, 0.5, 0.7)$ to the interval $[0.3, 0.5]$ results in $(0.50.3, 0.4$. Scaling allows pGOMEA to move a group of variables closer together in the permutation. Also, the random key diversity is improved in the population. Random key diversity is also ensured by re-encoding. After the GOM phase of pGOMEA, each random key gets a new values, while retaining the order of the random keys.

### 2.4 Population Sizing Scheme

When implemented, pGOMEA would look like the pseudocode in Algorithm 1. However, one needs to specify the population size before running the algorithm. Therefore, pGOMEA incorporates an exponential population sizing scheme. In this scheme, a population is started with size $n_{base}$. Every four times this population is evaluated, a population with size $2 \cdot n_{base}$ is evaluated once. This pattern recurses, so population $i$ is evaluated four times as often as population $i + 1$. Using such a scheme, no population size has to be estimated. When a population is converged, no evaluations are performed anymore for that population, allowing pGOMEA to evaluate more in the other populations.

## 3 Benchmark problem: Permutation Flowshop Scheduling

The PFSP is concerned with finding the optimal solution for scheduling $J$ jobs on $M$ *machines*. Each job requires $M$ operations, which should be performed sequentially, starting on machine 1

**Result:** A good/optimal solution with respect to fitness function $f$
$Pop \leftarrow rand\_Pop(n)$ ;
**while** $\neg termination\_criterion$ **do**
  $LT \leftarrow build\_LT(Pop)$ ; // Model-building
  **foreach** $receiver \in Pop$ **do**
    $receiver^* \leftarrow receiver$;
    $improved \leftarrow False$;
    **foreach** $set \in FOS$ **do** // Gene-pool Optimal Mixing
      $donor \leftarrow Random(Pop)$;
      $child \leftarrow Donate_{rescale}(receiver^*, set, donor, Rand(0,1) < 0.1)$;
      **if** $f(child) > f(receiver^*)$ **then**
        $receiver^* \leftarrow child$;
        $improved \leftarrow True$;
    **if** $\neg improved \vee NIS$ **then** // Forced Improvement
      **foreach** $set \in FOS$ **do**
        $child \leftarrow Donate_{rescale}(receiver^*, set, best\_solution, Rand(0,1) < 0.1)$;
        **if** $f(child) \geq f(receiver^*)$ **then**
          $receiver^* \leftarrow child$;
          break
    $receiver = Reencode(receiver^*)$ // Re-encoding
**return** best solution from $Pop$

**Algorithm 1:** GOMEA outline

and finishing on machine $M$ (the *Flowshop* property). Operations cannot be interrupted, but a job can be delayed when its operations are not performed immediately after each other. Any solution can be seen as a permutation of jobs, since each machine has to process the jobs in the same order (the *Permutation* property). In three field notation, the PFSP is denoted by $F|prmu|\gamma$, where $\gamma$ refers to the objective function that is used for optimizing the schedule. Here, we consider the total flowtime (TFT) criterion, which is defined as the sum of completion times of all jobs:

$$TFT(\pi) = \sum_{i=1}^{J} c(\pi_i, M). \tag{4}$$

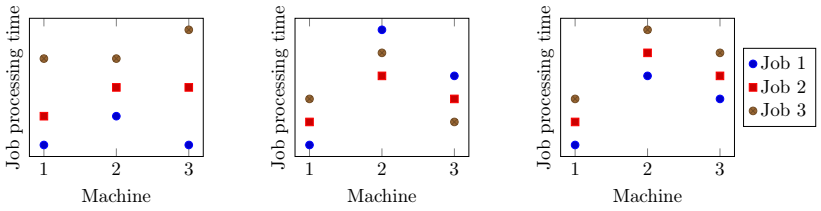
The completion times of all jobs can be calculated using the equations in (5) in $\mathcal{O}(J \cdot M)$ time. For the TFT criterion, the PFSP is NP-hard when $M > 1$.

$$\begin{aligned} &c(\pi_1, 1) = p(\pi_1, 1) \\ &c(\pi_1, j) = c(\pi_1, j-1) + p(\pi_1, j) \quad \text{for } j = 2 \cdots M \\ &c(\pi_i, 1) = c(\pi_{i-1}, 1) + p(\pi_i, 1) \quad \text{for } i = 2 \cdots J \\ &c(\pi_1, 1) = max\{c(\pi_{i-1}, j), c(\pi_i, j-1)\} + p(\pi_,j), \\ &\qquad \text{for } i = 2 \cdots J; \text{for } j = 2 \cdots M. \end{aligned} \tag{5}$$

### 3.1 Problem instances

**Taillard Instances**
For the PFSP, the most often used benchmark set is developed by Taillard [6]. The benchmark set can be divided in 12 ($J \times M$) sets with 10 instances each (See Table 1). The instances are a selection of the hardest randomly generated instances. Instances are considered to be difficuld if a simple metaheuristic does not often find the same makespan or if the found makespans are far from a lower bound on the makespan.
**Structured Instances**
Aalvanger [1] introduced a new set of benchmarks for testing algorithms on structured instances. The benchmark set contains the three types of structured instances as described by Watson et al. [7]: Job-correlated (JC), Machine-correlated (MC) and Mixed-correlated (MXC) instances (see Figure 1). In job-correlated instances, processing times are dependent on the job and not on the machines. Therefore the processing times of operations in one job are related. In machine-correlated instances the structure goes the other way around. Here, processing times on one machine are

related, while processing times within one job are unrelated. Mixed-correlated instances are equal to Machine-correlated instances, but here the relative ranks of job processing times are largely independent of the machine.




(a) Job-correlated (b) Machine correlation (c) Mixed correlation

Figure 1: Types of structure in PFSP instances.

For each of the three correlation types, four $(J \times 20)$ sets are generated (See underlined in Table 1). For each instance size, 1100 instances are generated, with varying values for correlation: $\alpha \in \{0.0, 0.1 \cdots 1.0\}$. For $\alpha = 0.0$, instances reflect the way Taillard instances are generated, higher values introduce more correlation. For $\alpha = 1.0$, every task in a job/machine has the same processing time.

| | $J = 20$ | $J = 50$ | $J = 100$ | $J = 200$ | $J = 500$ |
|---|---|---|---|---|---|
| $M = 5$ | $20 \times 5$ | $50 \times 5$ | $100 \times 5$ | | |
| $M = 10$ | $20 \times 10$ | $50 \times 10$ | $100 \times 10$ | $200 \times 10$ | |
| $M = 20$ | $\underline{20 \times 20}$ | $\underline{50 \times 20}$ | $\underline{100 \times 20}$ | $\underline{200 \times 20}$ | $500 \times 20$ |

Table 1: Sizes of the Taillard PFSP instances, for underlined sizes structured instances are available.

### 3.2 Comparing results

To compare algorithms for PFSP, the Relative Percentage Deviation (RPD) is often used. The RPD describes the relative distance to the best known upper bound $(UB)$ of an instance and the result of the algorithm $RES$. The RPD is calculated by

$$RPD(RES) = \frac{100 \cdot (RES - UB)}{UB}. \tag{6}$$

RPD values are best used when the upper bound is very close to the optimal solution. An RPD value of 0.0 then means that the optimal solution has been found. Over a set of runs, the average or median RPD is often reported (ARPD/MRPD). In our results, we also report the average over the MRPDs of multiple instances (AMRPD).
In order to calculate the significance in difference between two algorithms, we use the Mann-Whitney-U test. Unless reported otherwise, we use sample sizes of 20 per instance to find MRPD values. AMRPD values are found over 10 instances with the same size. For significance tests we use a significance level of $p < 0.05$.

## 4 Heuristics for the PFSP

### 4.1 Constructive heuristics

For the TFT criterion, Liu and Reeves have introduced the LR(x) heuristic [5], which can generate up to $J$ schedules, depending on the parameter $x$. LR(x) builds a schedule from the front to the back, using the following three steps:

1. Sort all jobs according to the index function.

2. Create $x$ partial schedules with the top-x jobs scheduled first. Extend the partial schedules by iteratively adding the best job according to the re-evaluated index function.

3. Select the best schedule generated in step 2).

The index function for adding job $i$ after the last job $k$ in the partial schedule consists of two components:

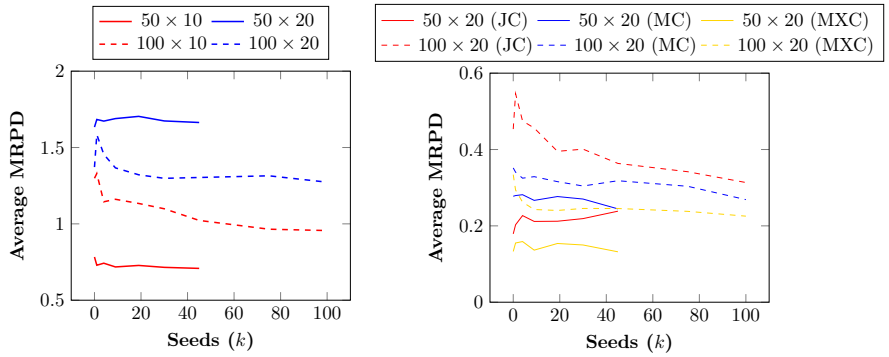
1. A *weighted total machine idle time*, punishing the time the machines wait between job $k$ and job $i$. Idle time on the first machines is punished more than idle time on the last machines.

2. The *artificial total flow time*, is the sum of the completion time of job $i$ plus the completion time of an artificial job representing the unscheduled jobs.

For the LR(x) heuristic, the last generated schedule is expected to be worse than the first. For the CDS heuristic, this is not the case.

### 4.2 Constructive heuristics seeding: results

For the LR heuristic we have tested the effect of seeding solutions in the initial populations of pGOMEA. Figure 2 shows that for most instances, more seeds result in better solutions. This holds for both structured and unstructured instances. An interesting fact is the effect of single-solution seeding. Here, the dominant new solution can misguide optimal mixing, leading to worse solutions. The effect of multi-solution seeding (maximal amount of seeds) is visible in Figure 3, where the behavior of (seeded) pGOMEA is shown over time. Here, one can see how the effect of seeding is the biggest when pGOMEA only uses a few fitness evaluations.




(a) Taillard instances

(b) Structured instances

Figure 2: Seeding with the LR heuristics: amount of seeds vs. solution quality after $50,000,000$ fitness evaluations.

### 4.3 Improvement heuristics

For the PFSP with the TFT criterion, various improvement heuristics exist. Each of these improvement heuristics are based on two fundamental permutation-heuristics: the insertion and swap heuristic. The swap heuristic takes two jobs and swaps them in a permutation. The insertion heuristic takes one job and puts it in another place in the permutation. Both heuristics have a neighbor-space that is quadratic in the amount of jobs and take $\mathcal{O}(J \cdot M)$ time to compute the fitness of a neighbor. In pGOMEA an improvement heuristic is best applied when a solution has changed in the GOM phase. For pGOMEA solving the PFSP with the TFT criterion, the swap heuristic has the most potential, especially on instances with a few machines [1]. In Figure 4 we show for unstructured instances how pGOMEA performs when this improvement heuristic is

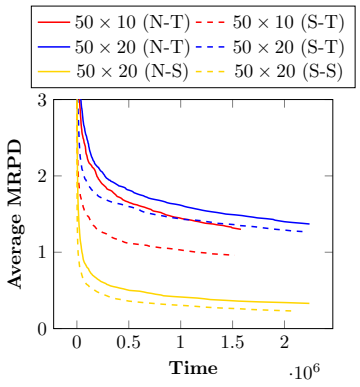



Figure 3: Maximal seeding (S) vs. Non-seeding (N): behavior over time for Taillard (T) instances and structured instances (S) (MXC, $\alpha = 0.3$).

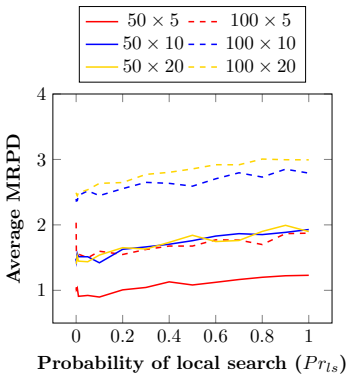



Figure 4: Hybrid pGOMEA quality on with respect to the probability of local search for Taillard instances

applied with some probability $Pr_{ls}$. The shown AMRPD values are a result of 10 medians over 5 runs.

## 5 Benchmark

As pGOMEA can best be enhanced with constructive heuristics generating as much schedules as possible, we add this feature to pGOMEA. Improvement heuristics are not used in pGOMEA, as they do not show consistent improvements on all instances. We compare pGOMEA with VNS4, a variable neighborhood searcher which uses an optimal form of combining the insertion heuristic and swap heuristic in order to solve the PFSP with the TFT criterion [4]. Both algorithms have been optimized to the level of Big-O notation and have been run for $400 \cdot J \cdot M$ milliseconds. Table 2 shows the MRPD values for VNS4 and pGOMEA. The best solution is marked bold and if the other solution performs significantly worse, its cell is marked grey.

The results show that pGOMEA often outperforms VNS4 significantly in most cases. In a few cases, pGOMEA is outperformed by VNS4, though this is often not significant. Unfortunately, both algorithms find far from optimal solutions within this time period. For better results, the algorithms should run a longer time.

Secondly, we have tested pGOMEA and VNS4 on multiple structured instances with size $100 \times 20$. For these problems we have run the algorithms for $400 \cdot (1 - \alpha) \cdot J \cdot M$ seconds, as structure makes the problems easier. Table 3 shows the results for three types of structured instances and three alpha values.

The results show for job-correlated instances that pGOMEA always outperforms the VNS4 algorithm. The type of structure apparently suits pGOMEA best, while VNS4 cannot benefit from an easier fitness landscape. The machine correlated instances with a high amount of structure ($\alpha \geq 0.4$) are however easier for VNS4. When machine and job correlation are mixed, the PFSP is best solved using pGOMEA. pGOMEA finds solutions with MRPD values lower than 0.5, showing that structured instances are much easier than the standard Taillard instances.

| | Best | pGOMEA | VNS4 | | Best | pGOMEA | VNS4 | | Best | pGOMEA | VNS4 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $50 \times 5$ | 64803 | **0.47** | 0.54 | $50 \times 10$ | 87207 | 1.26 | **1.23** | $50 \times 20$ | 125831 | **0.78** | 1.06 |
| | 68062 | **0.63** | 0.63 | | 82820 | **0.79** | 1.42 | | 119259 | **0.56** | 0.94 |
| | 63162 | **0.92** | 1.02 | | 79987 | **0.84** | 1.07 | | 116459 | **0.71** | 1.11 |
| | 68226 | 0.85 | **0.84** | | 86581 | **0.71** | 0.99 | | 120712 | **0.92** | 0.96 |
| | 69392 | **0.65** | **0.65** | | 86450 | **0.48** | 1.15 | | 118184 | 1.25 | **1.16** |
| | 66841 | **0.65** | 0.71 | | 86637 | **0.97** | 1.00 | | 120703 | **0.82** | 1.04 |
| | 66253 | **0.60** | 0.69 | | 88866 | **0.63** | 1.07 | | 122962 | **1.01** | 1.03 |
| | 64359 | **0.52** | 0.77 | | 86824 | 1.18 | **1.09** | | 122489 | **1.02** | 1.03 |
| | 62981 | 0.69 | **0.62** | | 85526 | **1.07** | 1.24 | | 121872 | **0.99** | 1.06 |
| | 68853 | 0.93 | **0.84** | | 88077 | **0.62** | 1.11 | | 124064 | **0.90** | 1.06 |
| $100 \times 5$ | 253713 | **0.80** | 0.94 | $100 \times 10$ | 299431 | **1.02** | 1.43 | $100 \times 20$ | 367267 | 1.63 | **1.57** |
| | 242777 | **1.04** | 1.12 | | 274593 | **1.55** | 1.77 | | 374032 | **1.41** | 1.50 |
| | 238180 | **0.57** | 0.94 | | 288630 | **1.15** | 1.54 | | 371417 | **1.47** | 1.56 |
| | 227889 | **0.82** | 1.01 | | 302105 | **1.55** | 1.59 | | 373822 | **1.54** | 1.75 |
| | 240589 | **0.77** | 0.95 | | 285340 | **1.09** | 1.19 | | 370459 | 1.58 | **1.50** |
| | 232936 | **0.82** | 1.12 | | 270817 | **1.14** | 1.56 | | 372768 | **1.60** | 1.66 |
| | 240669 | **0.85** | 0.90 | | 280649 | **1.21** | 1.30 | | 374483 | **1.30** | 1.71 |
| | 231428 | **1.00** | 1.06 | | 291665 | **0.81** | 1.52 | | 385456 | **1.39** | 1.56 |
| | 248481 | 1.00 | **0.93** | | 302624 | **1.20** | 1.44 | | 376063 | **1.43** | 1.58 |
| | 243360 | **0.87** | 0.88 | | 292230 | **1.02** | 1.58 | | 379899 | **1.45** | 1.68 |

Table 2: Quality of pGOMEA and VNS4 on Taillard instances.

| | | Job correlated | | | Machine Correlated | | | Mixed correlated | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Best | pGOMEA | VNS4 | Best | pGOMEA | VNS4 | Best | pGOMEA | VNS4 |
| $100 \times 20$ | $\alpha = 0.2$ | 354483 | **0.42** | 0.46 | 376876 | 0.28 | **0.18** | 392925 | **0.35** | 0.48 |
| | | 352436 | **0.38** | 0.58 | 379781 | **0.31** | 0.42 | 375810 | **0.26** | 0.66 |
| | | 354530 | **0.34** | 0.43 | 395682 | **0.18** | 0.42 | 400652 | **0.14** | 0.36 |
| | | 341043 | **0.26** | 0.52 | 389305 | **0.36** | 0.58 | 376562 | **0.31** | 0.67 |
| | | 354807 | **0.31** | 0.69 | 389688 | **0.25** | 0.59 | 368258 | **0.29** | **0.29** |
| | | 370144 | **0.33** | 0.54 | 360776 | **0.15** | 0.54 | 388375 | **0.33** | 0.63 |
| | | 351676 | **0.32** | 0.62 | 389651 | **0.33** | 0.59 | 379751 | 0.62 | **0.38** |
| | | 372308 | **0.31** | 0.39 | 377689 | 0.38 | **0.35** | 372021 | **0.18** | 0.29 |
| | | 362577 | **0.43** | 0.48 | 385017 | **0.29** | 0.40 | 364454 | **0.29** | 0.29 |
| | | 360703 | **0.39** | 0.57 | 389271 | **0.25** | 0.35 | 349791 | 0.28 | 0.52 |
| | $\alpha = 0.4$ | 338303 | **0.27** | 0.39 | 432943 | 0.25 | **0.20** | 409404 | **0.15** | 0.23 |
| | | 331602 | **0.23** | 0.51 | 404417 | 0.20 | **0.19** | 408221 | **0.11** | **0.11** |
| | | 337449 | **0.14** | 0.23 | 411893 | 0.10 | **0.09** | 381029 | **0.07** | 0.15 |
| | | 343512 | **0.15** | 0.49 | 410707 | 0.26 | **0.16** | 367183 | **0.18** | 0.30 |
| | | 321656 | **0.28** | 0.41 | 437327 | 0.20 | **0.12** | 392645 | **0.12** | 0.33 |
| | | 348395 | **0.23** | 0.53 | 418578 | 0.23 | **0.16** | 428664 | **0.06** | 0.10 |
| | | 350807 | **0.26** | 0.28 | 402707 | 0.14 | **0.13** | 400714 | 0.15 | **0.12** |
| | | 336867 | **0.11** | 0.47 | 419103 | **0.14** | **0.14** | 378965 | **0.16** | 0.28 |
| | | 334469 | **0.16** | 0.44 | 440173 | 0.21 | **0.15** | 411678 | **0.06** | 0.15 |
| | | 341423 | **0.35** | 0.78 | 417045 | 0.13 | **0.08** | 392508 | **0.10** | 0.22 |
| | $\alpha = 0.6$ | 319476 | **0.17** | 0.32 | 491858 | **0.11** | 0.12 | 395660 | **0.03** | 0.06 |
| | | 335018 | **0.13** | 0.30 | 407094 | 0.14 | **0.05** | 454738 | **0.02** | 0.02 |
| | | 311021 | **0.06** | 0.22 | 450581 | **0.04** | 0.02 | 365546 | **0.02** | **0.02** |
| | | 301670 | **0.16** | 0.37 | 391007 | 0.14 | **0.06** | 400760 | **0.00** | 0.00 |
| | | 303487 | **0.10** | 0.21 | 449246 | 0.13 | **0.03** | 447317 | **0.01** | 0.03 |
| | | 299529 | **0.12** | 0.22 | 479164 | **0.12** | 0.14 | 412359 | **0.02** | 0.05 |
| | | 320155 | **0.14** | 0.17 | 469897 | 0.41 | **0.10** | 450060 | **0.02** | **0.02** |
| | | 290573 | **0.15** | 0.27 | 471761 | 0.18 | **0.06** | 484204 | **0.01** | 0.02 |
| | | 305220 | **0.12** | 0.26 | 463159 | 0.14 | **0.10** | 414879 | **0.02** | **0.02** |
| | | 328872 | **0.15** | 0.27 | 453227 | 0.10 | **0.06** | 480739 | **0.02** | **0.02** |

Table 3: Quality of GOMEA and VNS4 on structured instances.

## 6 Conclusions

In this paper, we have shown the effect of incorporating constructive and improvement heuristics in permutation Gene-pool Optimal Mixing Evolutionary Algorithm (pGOMEA) for the permutation flowshop scheduling problem (PFSP) with the total flowtime criterion. We have shown that pGOMEA can effectively be improved by using a constructive heuristic for the initial schedules. In general, the more schedules are generated, the better pGOMEA performs. For improvement heuristics we have seen that the swap-heuristic can sometimes improve pGOMEA, though this is very dependent on the nature of the instances. Instances with a few machines are easier solved when using the swap-heuristic. Other instances become harder to solve for pGOMEA.

Using the effective combination of a constructive heuristic and pGOMEA, we have compared pGOMEA with VNS4. On structured and unstructured instances, pGOMEA outperforms VNS4. Only for machine correlated structured instances, VNS4 outperforms pGOMEA, though both algorithms find near-optimal solutions for these instances. Altogether, this research shows that pGOMEA is an effective model-based evolutionary algorithm that can easily and effectively be extended with domain knowledge.

## References

[1] Gerben Aalvanger. Incorporating domain knowledge in Permuation Gene-pool Optimal Mixing Evolutionary Algorithms. Master's thesis, Utrecht University, the Netherlands, 2017.

[2] Peter AN Bosman, Ngoc Hoang Luong, and Dirk Thierens. Expanding from discrete cartesian to permutation gene-pool optimal mixing evolutionary algorithms. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*, pages 637–644. ACM, 2016.

[3] J. Ceberio, E. Irurozki, A. Mendiburu, and J. A. Lozano. Extending distance-based ranking models in estimation of distribution algorithms. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 2459–2466, July 2014.

[4] Wagner Emanoel Costa, Marco César Goldbarg, and Elizabeth G Goldbarg. New VNS heuristic for total flowtime flowshop scheduling problem. *Expert Systems with Applications*, 39(9):8149–8161, 2012.

[5] Jiyin Liu and Colin R Reeves. Constructive and composite heuristic solutions to the p$||\sum C_i$ scheduling problem. *European Journal of Operational Research*, 132(2):439–452, 2001.

[6] E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278 – 285, 1993.

[7] Jean-Paul Watson, Laura Barbulescu, L Darrell Whitley, and Adele E Howe. Contrasting structured and random permutation flow-shop scheduling problems: search-space topology and algorithm performance. *INFORMS Journal on Computing*, 14(2):98–123, 2002.

# Appendix C

# List of abbreviations

**ARPD** . . . . . *Average Relative Pairwise Deviation*: A performance measure for the solution quality of an optimization algorithm. This measure uses the average distance to the best known solution for a problem instance.

**BBO** . . . . . *Black-Box Optimization*: Solving a combinatorial optimization problem, without knowing or assuming anything about the objective function (except the input).

**BOA** . . . . . *Bayesian Optimization Algorithm*: An estimation-of-distribution algorithm using a Bayesian network.

**CDS** . . . . . *Campbell, Dudek and Smith*: A constructive heuristic designed to find a good solution for the permutation flowshop scheduling problem.

**eCGA** . . . . . *extended Compact Genetic Algorithm*: An estimation-of-distribution algorithm using a marginal product model.

**EA** . . . . . *Evolutionary Algorithm*: A sub-group of combinatorial optimization algorithms that use a population-based approach, populations are incrementally updated using selection, recombination and mutation.

**EDA** . . . . . *Estimation-of-Distribution Algorithm*: A subset of model-based evolutionary algorithms that models the parameter distribution of a population in order to sample the offspring for the next population.

**FI** . . . . . *Forced Improvement*: The second phase in the gene-pool optimal mixing step of the gene-pool optimal mixing evolutionary algorithm, where optimal mixing is performed using the best-known solution in the population. The FI phase is only entered under certain conditions.

**FOS** . . . . . *Family-Of-Subsets*: A set of subsets of the problem variables. A FOS can be used to model dependencies between problem variables.

**FPE** . . . . . *Forward Pairwise Exchange*: An improvement heuristic designed to find a good solution for the permutation flowshop scheduling problem.

**GA** . . . . . *Genetic Algorithm*: A subset of evolutionary algorithms using a natural-evolution approach. Crossovers are used in the recombination phase.

**GLS** . . . . . *Genetic Local Search*: A genetic algorithm using a local search procedure to improve individuals.

**GOM** . . . . . *Gene-pool Optimal Mixing*: The recombination phase in the gene-pool optimal mixing evolutionary algorithm.

**GOMEA** . . . . . *Gene-pool Optimal Mixing Evolutionary Algorithm*: A model-based evolutionary algorithm using the contents of a family-of-subsets structure as crossover masks. Optimal mixing is performed with one randomly selected donor per subset in the family-of-subsets structure.

**pGOMEA** . . . . *permutation Gene-pool Optimal Mixing Evolutionary Algorithm*: The gene-pool optimal mixing evolutionary algorithm using a random-keys encoding for the permutation and using proximity and relative ordering information for building the linkage tree.

**LR** . . . . . *Liu and Reeves*: A constructive heuristic designed to find one or more good solutions for the permutation flowshop scheduling problem.

**LS** . . . . . *Local Search(er)*: An algorithm that optimizes a solution for a combinatorial optimization problem by walking through the search space from one solution towards better neighboring solutions.

**LTGA** . . . . . *Linkage Tree Genetic Algorithm*: A gene-pool optimal mixing evolutionary algorithm incorporating the linkage tree as family-of-subsets structure.

**LT-GOMEA** . . . *see LTGA*

**MBEA** . . . . . *Model-Based Evolutionary Algorithm*: A subset of evolutionary algorithms that uses a learned model over the population to improve recombination.

**MRPD** . . . . . *Median Relative Pairwise Deviation*: A performance measure for the solution quality of an optimization algorithm. This measure uses the median distance to the best known solution for a problem instance.

**MPM** . . . . . *Marginal Product Model:* A model capturing variable dependencies; this model creates sets of dependent variables that are independent of all other sets.

**NEH** . . . . . *Nawaz, Enscore and Ham*: A constructive heuristic designed to find a good solution for the permutation flowshop scheduling problem.

**NIS** . . . . . *No-Improvement Stretch*: The amount of consecutive generations without improvement of the best solution in the population after which the forced improvement phase is entered.

**OM** . . . . . *Optimal Mixing*: Crossover in optimal-mixing evolutionary algorithms using sets of dependent variables as crossover masks. OM substitutes the masked variables of a receiver solution by the variables of a donor solution if this does not decrease the solution quality of the receiver.

**PFSP** . . . . . *Permutation Flowshop Scheduling Problem*: A machine scheduling problem where $J$ jobs should be scheduled on $M$ machines. All job should be processed on each machines in the same given order. The order in which jobs are scheduled is the same for each machine.

**RA** . . . . . *Rapid Access*: A constructive heuristic designed to find a good solution for the permutation flowshop scheduling problem.

**RACS** . . . . . *Rapid Access with Closed order Search*: The rapid access heuristic with a limited improvement heuristic used afterwards.

**RAES** . . . . . *Rapid Access with Extensive Search*: The rapid access heuristic with an exhaustive improvement heuristic used afterwards.

**ROM** . . . . . *Recombinative Optimal Mixing*: The recombination phase in the recombinative optimal mixing evolutionary algorithm.

**ROMEA** . . . . . *Recombinative Optimal Mixing Evolutionary Algorithm*: A model-based evolutionary algorithm using the contents of a family-of-subsets structure as crossover masks in optimal mixing. Optimal mixing is performed with one randomly selected donor per receiver.

**RZ** . . . . . *Rajendran and Ziegler*: An index-based constructive heuristic designed to find a good solution for the permutation flowshop scheduling problem.

**RZ-LS** . . . . . *Rajendran and Ziegler Local Search*: Improvement heuristic in the Rajendran and Ziegler constructive heuristic for permutation flowshop scheduling problems.

**TFT** . . . . . *Total Flowtime*: An objective function for the permutation flowshop scheduling problem that minimizes the sum of completion times of the jobs.

**TSP** . . . . . *Traveling Salesman Problem*: A permutation problem where $n$ cities should be visited with a route as short as possible.

**UPGMA** . . . . . *Unweighted Pairwise Group Method with Arithmetic Mean*: A method for calculating dependencies between two sets of variables. UPGMA only considers the pairwise dependencies between individual variables in the two sets.

**VNS** . . . . . *Virtual Neighborhood Search*: A meta-heuristic combinatorial optimization algorithm using alternating search-neighborhoods.

# Bibliography


[1] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies–a comprehensive introduction. *Natural computing*, 1(1):3–52, 2002.

[2] Peter AN Bosman, Ngoc Hoang Luong, and Dirk Thierens. Expanding from discrete cartesian to permutation gene-pool optimal mixing evolutionary algorithms. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*, pages 637–644. ACM, 2016.

[3] Peter AN Bosman and Dirk Thierens. Permutation optimization by iterated estimation of random keys marginal product factorizations. In *International Conference on Parallel Problem Solving from Nature*, pages 331–340. Springer, 2002.

[4] Peter AN Bosman and Dirk Thierens. The roles of local search, model building and optimal mixing in evolutionary algorithms from a BBO perspective. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, pages 663–670. ACM, 2011.

[5] Peter AN Bosman and Dirk Thierens. Linkage neighbors, optimal mixing and forced improvements in genetic algorithms. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 585–592. ACM, 2012.

[6] Herbert G Campbell, Richard A Dudek, and Milton L Smith. A heuristic algorithm for the n job, m machine sequencing problem. *Management science*, 16(10):B–630, 1970.

[7] J. Ceberio, E. Irurozki, A. Mendiburu, and J. A. Lozano. Extending distance-based ranking models in estimation of distribution algorithms. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 2459–2466, July 2014.

[8] Wei-Ming Chen, Chu-Yu Hsu, Tian-Li Yu, and Wei-Che Chien. Effects of discrete hill climbing on model building for estimation of distribution algorithms. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 367–374. ACM, 2013.

[9] Wagner Emanoel Costa, Marco César Goldbarg, and Elizabeth G Goldbarg. New VNS heuristic for total flowtime flowshop scheduling problem. *Expert Systems with Applications*, 39(9):8149–8161, 2012.

[10] David G Dannenbring. An evaluation of flow shop sequencing heuristics. *Management science*, 23(11):1174–1182, 1977.

[11] Xingye Dong, Ping Chen, Houkuan Huang, and Maciek Nowak. A multi-restart iterated local search algorithm for the permutation flow shop problem minimizing total flow time. *Computers & Operations Research*, 40(2):627–632, 2013.

[12] Thyago SPC Duque, David E Goldberg, and Kumara Sastry. Enhancing the efficiency of the ECGA. In *International Conference on Parallel Problem Solving from Nature*, pages 165–174. Springer, 2008.

[13] Philippe Galinier and Jin-Kao Hao. Hybrid evolutionary algorithms for graph coloring. *Journal of combinatorial optimization*, 3(4):379–397, 1999.

[14] Michael R Garey, David S Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of operations research*, 1(2):117–129, 1976.

[15] Teofilo Gonzalez and Sartaj Sahni. Flowshop and jobshop schedules: complexity and approximation. *Operations research*, 26(1):36–52, 1978.

[16] Georges Harik. Linkage learning via probabilistic modeling in the ECGA. *Urbana*, 51(61):801, 1999.

[17] Mark Hauschild and Martin Pelikan. An introduction and survey of estimation of distribution algorithms. *Swarm and Evolutionary Computation*, 1(3):111–128, 2011.

[18] Johnny C Ho and Yih-Long Chang. A new heuristic for the n-job, m-machine flow-shop problem. *European Journal of Operational Research*, 52(2):194–202, 1991.

[19] Shih-Huan Hsu and Tian-Li Yu. Optimization by pairwise linkage detection, incremental linkage set, and restricted/back mixing: Dsmga-ii. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 519–526. ACM, 2015.

[20] David Iclanzan and Dan Dumitrescu. Overcoming hierarchical difficulty by hill-climbing the building block structure. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1256–1263. ACM, 2007.

[21] Edward Ignall and Linus Schrage. Application of the branch and bound technique to some flow-shop scheduling problems. *Operations research*, 13(3):400–412, 1965.

[22] Bryant A Julstrom. Comparing Darwinian, Baldwinian, and Lamarckian search in a genetic algorithm for the 4-cycle problem. In *Late Breaking Papers at the 1999 Genetic and Evolutionary Computation Conference*, pages 134–138. Citeseer, 1999.

[23] Claudio F Lima, Martin Pelikan, Kumara Sastry, Martin Butz, David E Goldberg, and Fernando G Lobo. Substructural neighborhoods for local search in the Bayesian optimization algorithm. In *Parallel Problem Solving from Nature-PPSN IX*, pages 232–241. Springer, 2006.

[24] Jiyin Liu and Colin R Reeves. Constructive and composite heuristic solutions to the $p||\sum C_i$ scheduling problem. *European Journal of Operational Research*, 132(2):439–452, 2001.

[25] DS Palmer. Sequencing jobs through a multi-stage process in the minimum total time–a quick method of obtaining a near optimum. *OR*, pages 101–107, 1965.

[26] Quan-Ke Pan and Rubén Ruiz. A comprehensive review and evaluation of permutation flowshop heuristics to minimize flowtime. *Computers & Operations Research*, 40(1):117–128, 2013.

[27] Quan-Ke Pan, Mehmet Fatih Tasgetiren, and Yun-Chia Liang. A discrete differential evolution algorithm for the permutation flowshop scheduling problem. *Computers & Industrial Engineering*, 55(4):795–816, 2008.

[28] Martin Pelikan, David E Goldberg, and Erick Cantu-Paz. Linkage problem, distribution estimation, and Bayesian networks. *Evolutionary computation*, 8(3):311–340, 2000.

[29] Chandrasekharan Rajendran and Hans Ziegler. An efficient heuristic for scheduling in a flowshop to minimize total weighted flowtime of jobs. *European Journal of Operational Research*, 103(1):129–138, 1997.

[30] Rubén Ruiz and Concepción Maroto. A comprehensive review and evaluation of permutation flowshop heuristics. *European Journal of Operational Research*, 165(2):479–494, 2005.

[31] Rubén Ruiz, Concepción Maroto, and Javier Alcaraz. Two new robust genetic algorithms for the flowshop scheduling problem. *Omega*, 34(5):461–476, 2006.

[32] Rubén Ruiz and Thomas Stützle. A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, 177(3):2033–2049, 2007.

[33] Kumara Sastry and David E Goldberg. Designing competent mutation operators via probabilistic model building of neighborhoods. In *Genetic and Evolutionary Computation Conference*, pages 114–125. Springer, 2004.

[34] SMA Suliman. A two-phase heuristic approach to the permutation flow-shop scheduling problem. *International Journal of production economics*, 64(1):143–152, 2000.

[35] E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278 – 285, 1993.

[36] Eric Taillard. Some efficient heuristic methods for the flow shop sequencing problem. *European journal of Operational research*, 47(1):65–74, 1990.

[37] Dirk Thierens. Scalability problems of simple genetic algorithms. *Evolutionary computation*, 7(4):331–352, 1999.

[38] Dirk Thierens. The linkage tree genetic algorithm. In *Proceedings of the 11th International Conference on Parallel Problem Solving from Nature: Part I*, PPSN'10, pages 264–273, Berlin, Heidelberg, 2010. Springer-Verlag.

[39] Dirk Thierens and Peter AN Bosman. Optimal mixing evolutionary algorithms. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 617–624. ACM, 2011.

[40] Lin-Yu Tseng and Ya-Tai Lin. A hybrid genetic local search algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, 198(1):84–92, 2009.

[41] AJ Umbarkar and PD Sheth. Crossover operators in genetic algorithms: a review. *ICTACT Journal on Soft Computing*, 6(1):1083–1092, 2015.

[42] Jean-Paul Watson, Laura Barbulescu, L Darrell Whitley, and Adele E Howe. Contrasting structured and random permutation flow-shop scheduling problems: search-space topology and algorithm performance. *INFORMS Journal on Computing*, 14(2):98–123, 2002.

[43] Yi Zhang and Xiaoping Li. Estimation of distribution algorithm for permutation flow shops with total flowtime minimization. *Computers & Industrial Engineering*, 60(4):706–718, 2011.

[44] Mark Zlochin and Marco Dorigo. Model-based search for combinatorial optimization: A comparative study. In *International Conference on Parallel Problem Solving from Nature*, pages 651–661. Springer, 2002.