AUTHOR(S): NEAU, C., REGNIER-COUDERT, O. and MCCALL, J.

TITLE: An analysis of indirect optimisation strategies for scheduling.

YEAR: 2018

Publisher citation: NEAU, C., REGNIER-COUDERT, O. and MCCALL, J. 2018. An analysis of indirect optimisation strategies for scheduling. In Proceedings of Institute of Electrical and Electronics Engineers (IEEE) congress on evolutionary computation (IEEE CEC 2018), 8-13 July 2018, Rio de Janeiro, Brazil. Piscataway: IEEE [online], article ID 8477967. Available from: https://doi.org/10.1109/CEC.2018.8477967

OpenAIR citation: NEAU, C., REGNIER-COUDERT, O. and MCCALL, J. 2018. An analysis of indirect optimisation strategies for scheduling. In Proceedings of Institute of Electrical and Electronics Engineers (IEEE) congress on evolutionary computation (IEEE CEC 2018), 8-13 July 2018, Rio de Janeiro, Brazil. Piscataway: IEEE, article ID 8477967. Held on OpenAIR [online]. Available from: https://openair.rgu.ac.uk/

An Analysis of Indirect Optimisation Strategies for Scheduling

Charles Neau, Olivier Regnier-Coudert and John McCall
School of Computing Science and Digital Media
Robert Gordon University
Aberdeen, Scotland
{c.neau, o.regnier-coudert, j.mccall}@rgu.ac.uk

***Abstract*—By incorporating domain knowledge, simple greedy procedures can be defined to generate reasonably good solutions to many optimisation problems. However, such solutions are unlikely to be optimal and their quality often depends on the way the decision variables are input to the greedy method. Indirect optimisation uses meta-heuristics to optimise the input of the greedy decoders. As the performance and the runtime differ across greedy methods and meta-heuristics, deciding how to split the computational effort between the two sides of the optimisation is not trivial and can significantly impact the search.**

In this paper, an artificial scheduling problem is presented along with five greedy procedures, using varying levels of domain information. A methodology to compare different indirect optimisation strategies is presented using a simple Hill Climber, a Genetic Algorithm and a population-based Local Search.

By assessing all combinations of meta-heuristics and greedy procedures on a range of problem instances with different properties, experiments show that encapsulating problem knowledge within greedy decoders may not always prove successful and that simpler methods can lead to comparable results as advanced ones when combined with meta-heuristics that are adapted to the problem. However, the use of efficient greedy procedures reduces the relative difference between meta-heuristics.

I. INTRODUCTION

In EAs, the term genotype describes the domain searched by an algorithm, that is the search space on which operators are applied. In order to assess solutions, a phenotype is required. The phenotype represents the domain in which a solution can be evaluated, or in other words, a domain that can be read by the fitness function. Not only does using alternative genotypes allow some problems to be modelled efficiently by EAs, but it may also map a problem to a domain which is more adapted to these algorithms. Most scheduling problem uses genotypes that differ from phenotypes. For example, most state of the art methods to the well-studied Permutation Flowshop Scheduling Problem (PFSP) [1] generate permutations that are translated into a schedule that can be evaluated, for example by computing its makespan. The function translating the genotype into a phenotype is often referred to as a decoder.

In the case of PFSP, all studies use the same decoder and there are no known alternative to it. This may not be the case for all problems for which several decoders can be defined, resulting in different phenotypes for a given genotype. Consequently, decoders to a same problem can result in solutions of different quality. In addition, decoders can be of different complexity or incorporate different quantity of domain-specific knowledge, impacting the runtime of the optimisation process or the fitness landscape.

The introduction of a decoder in the optimisation process may also be motivated by the need to help the optimisation algorithm to produce good solutions from the start of the search, to satisfy constraints or by the need to produce solutions of reasonable quality with a reduced number of fitness evaluations.

There is a large body of literature on the topic of indirect optimisation, revealing that it has been applied to many areas of computational intelligence including set covering problem [2], constraint satisfaction and graph colouring problems [3], routing and sequencing [4], project scheduling [5] or Bayesian network structure learning [6], [7], [8].

Using decoders enables the applications of well-researched algorithms that rely on standard representations to real-world problems in a wide range of domains such as telecommunications [9], nurse scheduling [10], web service composition [11] or transportation [12].

This paper studies how the use of different decoders, combined with different meta-heuristics impacts the search and proposes an analysis of the tradeoff between spending computational efforts on decoding or on optimisation. In order to propose several decoders for the same problem, a new artificial scheduling problem is proposed and instances with distinct characteristics generated. All encoders take a permutation as input.

The paper is organised as follows. First, the artificial scheduling problem and the instance generator used for experiments are described in the next section. In Section III, the different greedy procedures considered for the problem are detailed. Section IV describes the meta-heuristics selected for the study. The experimental procedure and associated results are presented in Section V before concluding in Section VI

II. THE SCHEDULING PROBLEM

For the purpose of this study, an artificial permutation problem is created. The motivation behind the creation of an additional scheduling problem lies in the fact that most existing scheduling problems do not leave room for many greedy procedures to be implemented and are therefore not suitable for the present study. The proposed problem is simple and enables the definition of many greedy algorithms, with

different level of domain knowledge. In this section, the new artificial problem is presented along with an instance generator.

A. The Problem

A set X of n jobs needs to be scheduled on a set M of m machines. In this problem, all machines handle the same number of jobs each and so $k \in \left[1, \frac{n}{m}\right]$. The objective of the problem is to find an allocation $x_{i,j,k}$, where $x_{i,j,k} = 1$ if job X_i is allocated to machine M_j in position k, 0 otherwise, such that $\max_i C_i$ is minimised. C_i is defined as:

$$C_i = \begin{cases} \sum_{j,k} P_{i,j} * (1 + \frac{1}{9+k}) * x_{i,j,k} & k = 1 \\ \sum_{j,k} (P_{i,j} * (1 + \frac{1}{9+k}) + \sum_{i'} C_{i'} * x_{i',j,k-1}) * x_{i,j,k} & k > 1 \end{cases} \quad (1)$$

and subject to:

$$\sum_{j,k} x_{i,j,k} = 1, \forall i \in [1, n] \quad (2)$$

The completion time C_i of each job X_i depends on its runtime on the machine $P_{i,j}$ and on its position k on this machine. A fraction of $P_{i,j}$ that depends on its position on the machines is added to the runtime. This fraction is set so that it increases the completion time of jobs with low values of k more significantly. The proportion of $P_{i,j}$ has been set in order to account for the relative positions of the jobs on a specific machine and for the purpose of the present study has been set to $\frac{1}{9+k}$ in (1).

B. Problem instances

A problem instance is defined by the cost matrix $P_{i,j}$. In order to generate instances with different properties, the stochastic process described in Algorithm 1 is used. The process first generates a mean runtime μ_i for a job by sampling a Gaussian distribution of mean μ and standard deviation α. In order to generate a runtime for each machine Pi, j, another Gaussian distribution with mean μ_i is sampled with standard deviation β. For this paper, μ was set to 100 and kept unchanged across instances. We refer to instances using the notation *n-m-α-β-id*.

Algorithm 1 Instance generator

```
Input: n, m, μ, α, β,
for i ← 1, n do
    μ_i = N(μ, α)
    for j ← 1, m do
        P_{i,j} = N(μ_i, β)
    end for
end for
Output P_{i,j}
```

III. GREEDY PROCEDURES

All greedy procedures designed for the problem follow a generic overall method in which items of an input permutation π are allocated to a machine at a specific position k iteratively. The overall process is described in Algorithm 2. Although this process is common to all methods, the allocate function that determine the machine and the position at which the job should be performed differs. In this section, five greedy decoders are described.

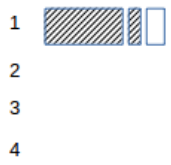

Fig. 1. Allocation of a job using LG. Jobs in grey are already allocated while the position chosen by LG for the next job is shown in white.

Algorithm 2 Overall greedy procedure to construct a schedule

```
Input permutation π
Initialise x = 0_{n×m×n/m}
for i ← 1, |π| do
    j, k ← allocate(π, i, x)
    x_{π_i,j,k} = 1
end for
```

A. Length Greedy Procedure (LG)

The Length Greedy procedure (LG) allocates the next job X_{π_i} to the first machine that has not reached its maximum number of jobs n/m, at the last position available as illustrated in Figure 1. Outline of the procedure is provided in Algorithm 3.

Algorithm 3 LG allocation

```
function allocate(π, i, x)
    machine = ⌈ i / (n/m) ⌉
    position = Σ_k x_{π_i,j,k} + 1
    return machine, position
end function
```

B. Width Greedy Procedure (WG)

The Width Greedy procedure (WG) allocates the next job X_{π_i} to the machine that can process it at the smallest position. The procedure is illustrated in Figure 2 and Algorithm 4.

LG and WG do not incorporate much problem information and so they both can be described as naive. One of their common characteristics is that each input permutation π results in a distinct schedule and so all possible schedules can be explored using these methods.

C. Finish Greedy Procedure (FG)

As shown in Figure 3 and Algorithm 5, the Finish Greedy procedure (FG) loops through all machines and allocates the job X_{π_i} to the machine on which it can complete at the earliest time.

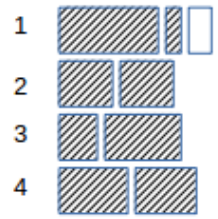

Fig. 2. Allocation of a job using WG. Jobs in grey are already allocated while the position chosen by WG for the next job is shown in white.

Algorithm 4 WG allocation

```
1: function allocate(π, i, x)
2:     machine = i mod m
3:     position = Σ_k x_{π_i,j,k} + 1
4:     return machine, position
5: end function
```

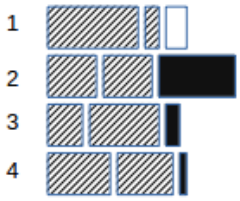

Fig. 3. Allocation of a job using FG. Jobs in grey are already allocated while the position chosen by FG for the next job is shown in white. Positions shown in black are the other potential positions for the jobs that are not chosen.

Algorithm 5 FG allocation

```
1: function allocate(π, i, x)
2:     bestMachine = 0, bestPosition = 0, bestScore = −1
3:     for j ← 1, m do
4:         position = Σ_k x_{π_i,j,k} + 1
5:         if position ≤ n/m then
6:             if position = 1 then
7:                 score = P_{π_i,j} ∗ (1 + 1/(9+position))
8:             else
9:                 score = P_{π_i,j} ∗ (1 + 1/(9+position)) + Σ_{i'} C_{i'} ∗ x_{i',j,position−1}
10:            end if
11:            if bestScore = −1 | bestScore > score then
12:                bestScore = score
13:                bestMachine = j
14:                bestPosition = position
15:            end if
16:        end if
17:    end for
18:    return bestMachine, bestPosition
19: end function
```

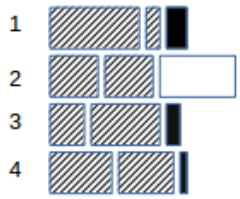

Fig. 4. Allocation of a job using SG. Jobs in grey are already allocated while the position chosen by SG for the next job is shown in white. Positions shown in black are the other potential positions for the jobs that are not chosen.

D. Start Greedy Procedure (SG)

Similar to FG, the Start Greedy procedure (SG) loops through all machines and allocates the job X_{π_i} to the machine on which it can start at the earliest time. It is detailed in Figure 4 and Algorithm 6.

Algorithm 6 SG allocation

```
1: function allocate(π, i, x)
2:     bestMachine = 0, bestPosition = 0, bestScore = −1
3:     for j ← 1, m do
4:         position = Σ_k x_{π_i,j,k} + 1
5:         if position ≤ n/m then
6:             if position = 1 then
7:                 score = 0
8:             else
9:                 score = Σ_{i'} C_{i'} ∗ x_{i',j,position−1}
10:            end if
11:            if bestScore = −1 | bestScore > score then
12:                bestScore = score
13:                bestMachine = j
14:                bestPosition = position
15:            end if
16:        end if
17:    end for
18:    return bestMachine, bestPosition
19: end function
```

E. Execution Greedy Procedure (EG)

The final decoder is the Execution Greedy procedure (EG). Similarly to FG and SG, EG loops through all machines and allocates the job X_{π_i} to the machine on which its processing time is the shortest. It is detailed in Figure 5 and Algorithm 7.

Unlike WG and LG, FG, SG and EG all have additional problem knowledge in the sense that some optimisation is done within these algorithms. The three methods generate schedules while trying to minimise a time objective, whether this is finish, start or execution time. While WG and LG are by definition able to cover the whole space of possible schedules, FG, SG and EG may not as many input permutation may result in similar schedules.

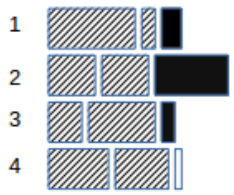

Fig. 5. Allocation of a job using EG. Jobs in grey are already allocated while the position chosen by EG for the next job is shown in white. Positions shown in black are the other potential positions for the jobs that are not chosen.

Algorithm 7 EG allocation

```
1: function allocate(π, i, x)
2:    bestMachine = 0, bestPosition = 0, bestScore = −1
3:    for j ← 1, m do
4:       position = Σ_k x_{π_i,j,k} + 1
5:       if position ≤ n/m then
6:          score = P_{π_i,j} ∗ (1 + 1/(9+position))
7:          if bestScore = −1 | bestScore > score then
8:             bestScore = score
9:             bestMachine = j
10:            bestPosition = position
11:         end if
12:      end if
13:   end for
14:   return bestMachine, bestPosition
15: end function
```

F. Performance and runtime of the greedy procedures

In order to describe the different greedy procedures introduced in this paper in terms of performance, runtime and landscapes, preliminary experiments were run on selected instances. For this experiments, 100 instances of each problem characteristics were generated. For each instance, 100 solutions were randomly generated, each used as input to all greedy procedures. The resulting fitness values were collected and plotted.

Three pairs of $\{\alpha, \beta\}$ were selected. These are $\{0.9, 0.1\}$, $\{0.9, 0.3\}$ and $\{0.9, 0.9\}$. As illustrated in Figures 6, 7, 8 for dimensions 50x5, these give different profiles with respect to the performance of the greedy procedures. When $\alpha = 0.9$ and $\beta = 0.1$, the difference between the best greedy approach (EG) and the worst ones (LG, WG and SG) is 7%. When $\alpha = 0.9$ and $\beta = 0.9$, this difference reaches 83%, showing that instances generated with the latter values of α and β are more challenging for LG, WG and SG.

It is also interesting to observe the worst solutions obtained by the greedy methods. Across all instances, LG and WG appear to produce the worst solutions in comparison with other approaches. This is in line with their principle that allows both LG and WG to generate every possible phenotype that exists to a problem. This is not the case with the three other algorithms that incorporate extra problem knowledge which will naturally

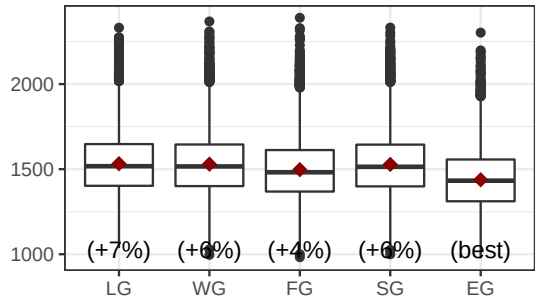

Fig. 6. Schedule quality using the five greedy search on instances generated using $\alpha = 0.9$ and $\beta = 0.1$

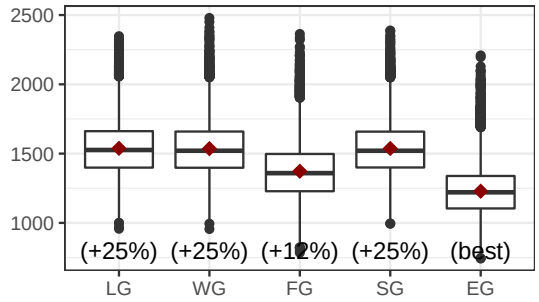

Fig. 7. Schedule quality using the five greedy search on instances generated using $\alpha = 0.9$ and $\beta = 0.3$

prevent very poor phenotypes to be evaluated. While this seems a good feature, especially when little computation is available, it may not always support search algorithms and introduce plateaux, where several solutions produced, have similar fitness values. Plateaux are often a challenge for algorithms such as GAs because they prevent discrimination between solutions and so affect selection.

Additional experiments were performed to gain insight on the difference between the greedy methods in terms of runtime. 1000 solutions were generated for three types of instances for problems of dimensions 50x5 and evaluated using each greedy procedure. The average runtimes associated with each and associated standard deviations are provided in Table I. Experiments were run on a standard PC (2.5 GHz CPU, 8GB

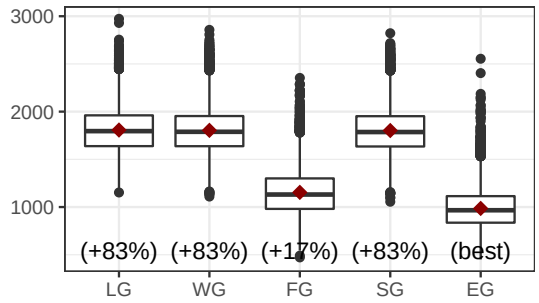

Fig. 8. Schedule quality using the five greedy search on instances generated using $\alpha = 0.9$ and $\beta = 0.9$

RAM) using R.

As expected, the runtimes correlate with the amount of problem knowledge incorporated into each greedy procedure. The two most naive methods, LG and WG, exhibit the fastest profiles and the difference between their runtime and those of the three remaining techniques is large and statistically significant (based on student t-tests).

FG and EG are the two methods taking the longest time to translate a genotype into a phenotype and there is no significant difference between the two. SG's runtime is between those of LG and WG and those of FG and EG. Thus, in terms of runtime, the 5 greedy approaches exhibit 3 distinct profiles.

TABLE I
AVERAGE RUNTIME OF THE GREEDY PROCEDURES OVER 1000 RUNS ON 50X5 INSTANCES, SHOWING THE TIME (IN μS) REQUIRED TO PERFORM A SINGLE EVALUATION

Greedy Proc.	$\{\alpha, \beta\}$		
	$\{0.9, 0.1\}$	$\{0.9, 0.3\}$	$\{0.9, 0.9\}$
LG	320 (143)	308 (322)	310 (107)
WG	284 (108)	281 (77)	282 (119)
FG	1670 (335)	1678 (257)	1763 (294)
SG	1263 (232)	1241 (288)	1249 (242)
EG	1625 (247)	1501 (227)	1503 (283)

IV. META-HEURISTICS

In order to optimise the input to the greedy procedures described above, three meta-heuristics were selected: a Hill Climber (HC) local search algorithm, a Genetic Algorithm (GA) and a new algorithm which evolves a population of solution by generating multiple neighbours for selected solutions. In this section, we describe each of the three methods.

A. Hill Climber

The pseudo-code of the standard HC used in the study is presented in Algorithm 8. The mutation operator chosen to define the neighbourhood is the swap operator which swaps two randomly selected elements from a permutation. Note that the initial solution is chosen as the best among $2n$ randomly generated solutions denoted as pop, to reduce the impact of poor initial solutions on the search. Preliminary experiments using alternative operators (insert and adjacent swap) were also investigated but did not yield better results.

Algorithm 8 Hill Climber

```
Generate pop uniformly at random
Evaluate and rank pop
x = pop_1
xbest = x
fbest = f(x)
for i ← 1, maxFEs do
    Generate x by applying mutation to xbest
    if f(x) < fbest then
        xbest = x
        fbest = f(x)
    end if
end for
return xbest
```

Algorithm 9 Genetic Algorithm

```
Generate pop uniformly at random
Evaluate pop
for i ← 1, maxGen do
    newpop = ∅
    Add xbest to newpop
    while |newpop| < |pop| do
        Select p_1 and p_2
        Generate offsprings c_1 and c_2 by applying crossover
        to p_1 and p_2 with probability cRate
        Mutate offsprings c_1 and c_2 by applying mutation
        with probability mRate
        Evaluate c_1 and c_2
        Add c_1 and c_2 to newpop
    end while
end for
return xbest
```

B. Genetic Algorithm

The GA used in this study is a standard generational GA. Its outline is provided in Algorithm 9. It uses elitism of size 1, a population size of $2n$, a tournament selection of size 3, partially matched crossover with crossover rate $cRate = 0.8$ and swap mutation with mutation rate $mRate = 0.1$.

C. Mutations on Selection Algorithm (MOSA)

A third algorithm called Mutations on Selection Algorithm (MOSA) is introduced in order to provide an alternative to the simple naive HC and the GA. Similarly to the GA, this algorithm evolves a population of solutions. However, it does not use any crossover operator and relies on a mutation operator to generate new solutions as does HC. The outline of MOSA is provided in Algorithm 10. The algorithm starts by generating an initial population at random. The population is then evaluated and ranked. At each generation, truncation selection is used to select the best k% of pop. Selected solutions are then mutated cyclically one after the other and inserted into the new population until $newpop$ is of the same size as pop. This means that each selected solution will generate multiple offsprings in the next population. The overall best solution $xbest$ seen during the search is returned.

For the experiments presented in this paper, for consistency between algorithms and based on preliminary results, the population size is set to $2n$ and the truncation size is set to 5% of the population size.

MOSA essentially makes use of the ranks of solutions to direct the search and favours small genotype changes rather than larger ones such as those resulting from the use of crossover operators. This matches the principles of other algorithms coupling either mutation or probabilistic model sampling and selection mechanisms to move in the search space [7], [13].

Algorithm 10 MOSA

Generate pop uniformly at random
Evaluate and rank pop
$xbest \leftarrow pop_1$
for $i \leftarrow 1, maxGen$ **do**
  $newpop = \varnothing$
  Select subset $selection$ of k solutions from pop by truncation
  $j = 0$
  while $|newpop| < |pop|$ **do**
    Mutate $selection_j$ and insert into $newpop$
    if $j = |selection|$ **then**
      $j = 0$
    else
      $j = j + 1$
    end if
  end while
  Evaluate and rank $newpop$
  if $newpop_1$ is better than $xbest$ **then**
    $xbest \leftarrow pop_1$
  end if
end for
return $xbest$

V. EXPERIMENTS AND RESULTS

A. Experimental setup

The main experimental analysis aims to provide insights on how the pairing of meta-heuristics and greedy procedures perform on instances with different characteristics (different n, m, α and β). For this purpose, all combinations were evaluated on each problem family. To reduce the bias that could potentially be introduced when randomly generating an instance, 5 instances were created for each $\{n, m, \alpha, \beta\}$ set. The maximum number of fitness evaluations used as stopping criteria for all algorithms is provided in Table II. These were chosen based on preliminary experiments on sample instances in order for all algorithms to have enough time to converge. Problems of size 20, 50 and 100 are considered. For each, two values of m are considered, $m = n/2$ and $m = n/10$, that is problems where a solution will consist in 2 and 10 jobs per machine respectively. A total of 50 runs were performed for each set of algorithms on each instance. Student t-test with Bonferroni correction was performed to test statistical differences in results between strategies for each problem family. Combinations that are statistically better than others are highlighted in bold in the result tables.

TABLE II
MAXIMUM NUMBER OF FITNESS EVALUATIONS

n	Fitness Evaluations
20	1200
50	30000
100	60000

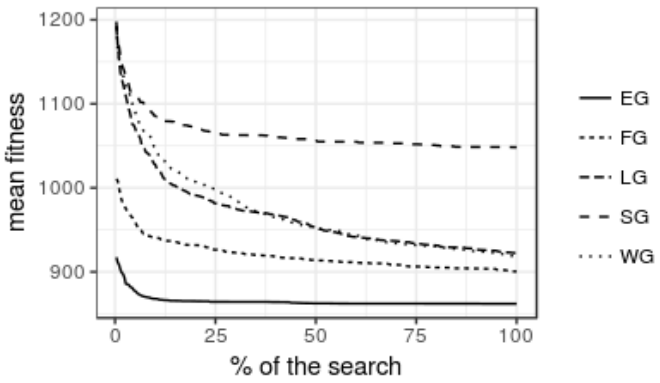

Fig. 9. Fitness of the best solution obtained by the GA over time, averaged over 20 runs on *50-5-0.9-0.3-1*

B. Results and discussion

TABLE III
RESULTS ON INSTANCES WITH $n = 20$

	GA	HC	MOSA
20-2-0.9-0.1-LG	1214.61	1240.62	1195.79
20-2-0.9-0.1-WG	1215.19	1241.20	**1191.31**
20-2-0.9-0.1-FG	1214.52	1229.22	**1191.24**
20-2-0.9-0.1-SG	1221.49	1238.42	**1191.24**
20-2-0.9-0.1-EG	1374.28	1378.90	**1191.24**
20-2-0.9-0.3-LG	1063.01	1139.31	1034.47
20-2-0.9-0.3-WG	1060.78	1134.85	**1032.45**
20-2-0.9-0.3-FG	1056.46	1080.99	**1032.45**
20-2-0.9-0.3-SG	1090.25	1127.78	**1032.45**
20-2-0.9-0.3-EG	1085.96	1095.11	**1032.45**
20-2-0.9-0.9-LG	1004.99	1217.20	**980.82**
20-2-0.9-0.9-WG	1004.15	1209.14	**980.44**
20-2-0.9-0.9-FG	988.42	1008.92	**980.44**
20-2-0.9-0.9-SG	1070.72	1191.26	**980.44**
20-2-0.9-0.9-EG	1036.64	1049.05	**980.44**
20-10-0.9-0.1-LG	300.82	320.82	282.13
20-10-0.9-0.1-WG	304.11	320.32	**279.70**
20-10-0.9-0.1-FG	288.68	309.26	**279.47**
20-10-0.9-0.1-SG	294.79	320.96	**279.47**
20-10-0.9-0.1-EG	295.09	308.46	**279.47**
20-10-0.9-0.3-LG	263.34	306.77	228.88
20-10-0.9-0.3-WG	268.20	306.49	228.11
20-10-0.9-0.3-FG	234.04	252.97	**226.33**
20-10-0.9-0.3-SG	264.58	304.10	227.85
20-10-0.9-0.3-EG	242.02	256.12	**225.92**
20-10-0.9-0.9-LG	219.37	326.46	129.83
20-10-0.9-0.9-WG	219.00	332.86	**126.19**
20-10-0.9-0.9-FG	132.74	145.78	**126.19**
20-10-0.9-0.9-SG	250.99	323.22	**126.19**
20-10-0.9-0.9-EG	132.87	153.23	**126.19**

Tables III, IV and V show the average fitness obtained by each combination of meta-heuristics and greedy procedure for each family of problem. Note that the results provided are averaged over 50 runs on each of the 5 instances produced for a given problem characteristic and so are overall averaged over 250 runs.

Experiments first highlight differences in terms of quality of solutions produced by the different meta-heuristics without particular consideration for the greedy procedure chosen. Generally speaking and as expected, HC is the least performing

TABLE IV
RESULTS ON INSTANCES WITH $n = 50$

	GA	HC	MOSA
50-5-0.9-0.1-LG	1229.50	1354.18	1193.07
50-5-0.9-0.1-WG	1231.71	1356.23	1187.84
50-5-0.9-0.1-FG	1234.16	1307.51	1186.88
50-5-0.9-0.1-SG	1259.92	1338.93	1186.88
50-5-0.9-0.1-EG	1196.14	1243.53	**1178.44**
50-5-0.9-0.3-LG	1003.92	1286.86	931.72
50-5-0.9-0.3-WG	1006.28	1282.26	**927.83**
50-5-0.9-0.3-FG	1000.12	1097.42	**927.29**
50-5-0.9-0.3-SG	1135.23	1257.80	**927.29**
50-5-0.9-0.3-EG	951.33	1032.39	**927.00**
50-5-0.9-0.9-LG	650.14	1295.53	562.57
50-5-0.9-0.9-WG	648.18	1293.98	**557.78**
50-5-0.9-0.9-FG	599.41	694.28	**556.61**
50-5-0.9-0.9-SG	1005.87	1246.13	**557.65**
50-5-0.9-0.9-EG	607.36	649.81	**556.37**
50-25-0.9-0.1-LG	356.62	419.99	**343.23**
50-25-0.9-0.1-WG	356.79	420.93	**343.04**
50-25-0.9-0.1-FG	347.05	399.01	**343.04**
50-25-0.9-0.1-SG	353.56	418.51	**343.04**
50-25-0.9-0.1-EG	357.59	397.11	**343.04**
50-25-0.9-0.3-LG	278.00	386.04	**245.33**
50-25-0.9-0.3-WG	279.85	386.55	**245.05**
50-25-0.9-0.3-FG	249.58	304.11	**245.05**
50-25-0.9-0.3-SG	284.63	386.29	**245.05**
50-25-0.9-0.3-EG	251.92	288.35	**245.05**
50-25-0.9-0.9-LG	237.00	439.00	**204.28**
50-25-0.9-0.9-WG	233.52	441.50	**204.25**
50-25-0.9-0.9-FG	**204.44**	209.27	**204.25**
50-25-0.9-0.9-SG	295.07	434.40	**204.25**
50-25-0.9-0.9-EG	**204.49**	208.86	**204.25**

TABLE V
RESULTS ON INSTANCES WITH $n = 100$

	GA	HC	MOSA
100-10-0.9-0.1-LG	1120.27	1305.59	1056.85
100-10-0.9-0.1-WG	1123.58	1305.80	1053.83
100-10-0.9-0.1-FG	1107.68	1228.34	1052.38
100-10-0.9-0.1-SG	1166.01	1284.96	1052.38
100-10-0.9-0.1-EG	1079.45	1194.90	**1036.69**
100-10-0.9-0.3-LG	963.99	1316.90	833.26
100-10-0.9-0.3-WG	971.99	1318.22	825.90
100-10-0.9-0.3-FG	884.45	1026.83	823.71
100-10-0.9-0.3-SG	1139.47	1284.78	825.21
100-10-0.9-0.3-EG	822.05	964.48	**789.34**
100-10-0.9-0.9-LG	747.47	1555.54	538.70
100-10-0.9-0.9-WG	755.05	1564.57	529.12
100-10-0.9-0.9-FG	547.37	699.35	497.05
100-10-0.9-0.9-SG	1309.65	1488.05	528.05
100-10-0.9-0.9-EG	522.48	605.02	**486.82**
100-50-0.9-0.1-LG	376.07	434.45	**358.19**
100-50-0.9-0.1-WG	378.49	434.91	**358.05**
100-50-0.9-0.1-FG	**358.62**	406.73	**358.05**
100-50-0.9-0.1-SG	366.42	435.37	**358.05**
100-50-0.9-0.1-EG	360.04	402.10	**358.05**
100-50-0.9-0.3-LG	347.68	429.19	**321.55**
100-50-0.9-0.3-WG	350.07	426.70	**321.19**
100-50-0.9-0.3-FG	**321.38**	341.20	**321.19**
100-50-0.9-0.3-SG	341.96	426.42	**321.19**
100-50-0.9-0.3-EG	**321.45**	332.74	**321.19**
100-50-0.9-0.9-LG	269.05	477.13	147.23
100-50-0.9-0.9-WG	264.82	478.36	140.50
100-50-0.9-0.9-FG	**137.03**	152.58	**136.59**
100-50-0.9-0.9-SG	320.00	475.25	140.25
100-50-0.9-0.9-EG	**137.52**	147.06	**136.59**

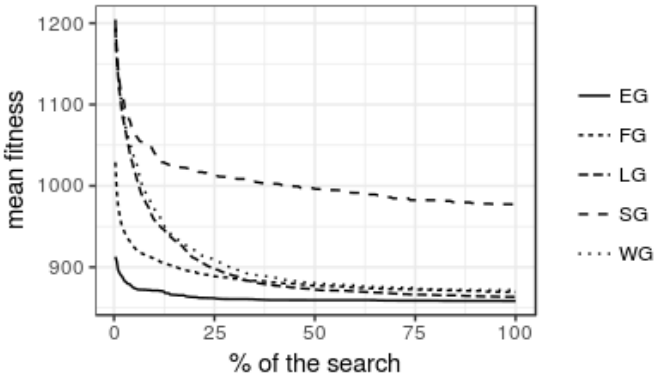

Fig. 10. Fitness of the best solution obtained by MOSA over time, averaged over 20 runs on *50-5-0.9-0.3-1*

method. Although the GA used in this study is better than HC, the overall best meta-heuristics is the proposed MOSA, which makes use of both local move operators and of a population.

Figures 9 and 10 show the fitness of the best solution found over time for GA and MOSA respectively and so confirm that both algorithms do not suffer from early convergence issues. While both methods continuously produce solutions for improving fitness, MOSA does so in a faster manner. It is particularly apparent in the early stages of the search, where the mutation operator used in MOSA leads to faster improvements than the combination of mutation and crossover used by the GA.

The extent of the difference between the algorithms is also correlated with the instance characteristics as illustrated in Figure 11, where the relative percentage deviation between MOSA and GA is shown for dimension 50. This shows how much better the results of MOSA are than those of GA. Because the value is positive on all instances, MOSA always outperforms GA. In general, the deviation increases as β increases. This reflects higher differences between the job costs in the $P_{i,j}$ matrix.

By incorporating a different amount of problem information in the different greedy procedures, bias was introduced towards some of them, expecting to perform better on random solutions. This was confirmed by the earlier experiments for which results were provided in Figures 6, 7 and 8. When considering the use of a meta-heuristics that may not be particularly adapted to the problem, such as the GA used in the present study, the importance of the greedy procedure is key to obtaining good final solutions. When considering GA, the use of FG and EG guarantees a better final solution that other greedy approaches. However, when considering MOSA, overall better adapted to this problem, the choice of the greedy procedure becomes less important and on many instances, leads to the same final results.

This is without considering the runtime associated with each of the greedy methods. Reported results are based on a fixed number of fitness evaluations, similar to all runs, regardless of the runtime. As seen in Table I, the runtime can vary significantly across greedy methods with FG at least 5 times

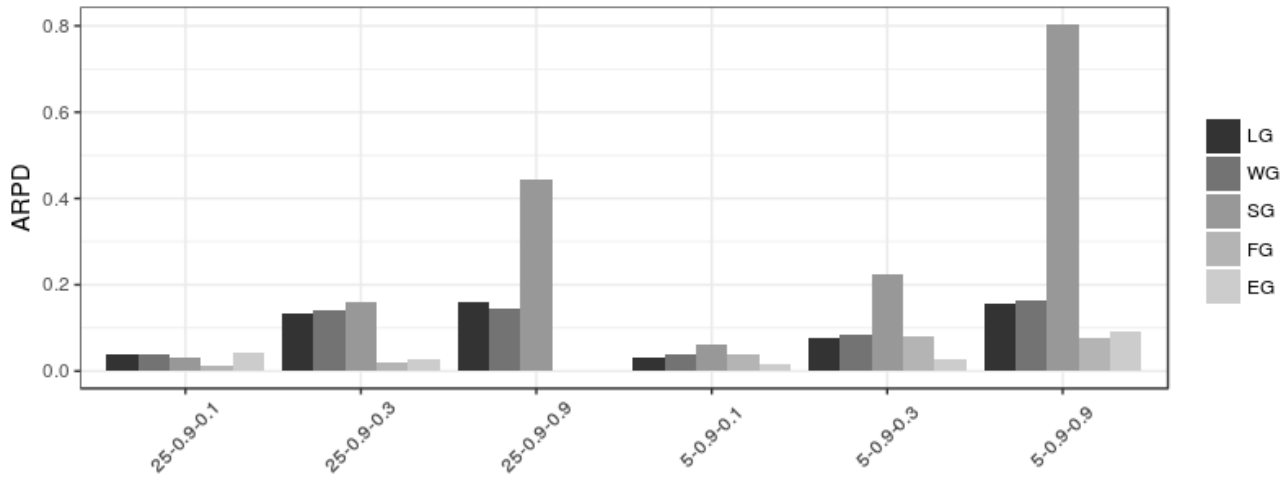

Fig. 11. ARPD of GA vs MOSA for combined with all greedy procedures on all instances of size n=50

more computationally expensive than WG for example and thus the choice of the greedy method should be recommended based on the computational time available.

Finally, experiments also highlighted the importance of carefully designing greedy procedures. For example, regardless of the meta-heuristic chosen and despite having some problem information embedded, SG produces solutions of particularly bad quality. Despite Figures 6, 7 and 8 showing that its performance is similar to LG and WG based on random solutions, it actually proves worst than LG and WG when used as part of an indirect optimisation strategy. In this particular case, the knowledge embedded into the greedy procedure acts as a constraint that prevents some phenotypes to be produced, some of them likely to be of high quality.

VI. CONCLUSIONS

In this paper, a methodology to compare greedy procedures and the optimisation of their input has been introduced. A novel artificial problem, an instance generator and five greedy methods presented. Three optimisation algorithms were used to perform the indirect optimisation, including a novel algorithm, MOSA, which exhibits better performance than a GA and a HC.

Beyond the simple comparison of meta-heuristics, the analysis shows the influence of the greedy methods within an indirect search framework. While greedy procedures that contains a lot of problem knowledge (e.g. EG) are essential with some meta-heuristics like HC and GA, they can be substituted by more naive approaches (e.g. LG, WG) when combined with better meta-heuristics like MOSA. This can prove particularly useful when considering the runtime of individual greedy procedures, where naive approaches take considerable less time to decode a solution.

The analysis performed in this paper provided good insights on how meta-heuristics and greedy decoders can interact. However, more elements should be considered in order to get a more complete picture. Future work should incorporate fitness landscape analysis and highlight how landscapes associated with each decoder may support different optimisation techniques. Finally, the impact of the problem characteristics α and β should be further studied.

REFERENCES

[1] M. Ayodele, J. Mccall, and O. Regnier-coudert, "RK-EDA : A Novel Random Key Based Estimation of Distribution Algorithm," in *International Conference on Parallel Problem Solving from Nature*, 2016, pp. 849–858.

[2] U. Aickelin, "An Indirect Genetic Algorithm for Set Covering Problems," *Journal of the Operational Research Society*, vol. 53, no. 10, pp. 1118–1126, 2002.

[3] X. Hao and J. Liu, "A multiagent evolutionary algorithm with direct and indirect combined representation for constraint satisfaction problems," *Soft Computing*, 2017.

[4] J. Kubalik and M. Snizek, "A Novel Evolutionary Algorithm with Indirect Representation and Extended Nearest Neighbor Constructive Procedure for Solving Routing Problems," in *International Conference on Intelligent Systems Design and Applications*, 2014.

[5] M. Ayodele, J. McCall, and O. Regnier-Coudert, "Estimation of distribution algorithms for the Multi-Mode Resource Constrained Project scheduling problem," in *2017 IEEE Congress on Evolutionary Computation, CEC 2017 - Proceedings*, 2017.

[6] R. Kabli, F. Herrmann, and J. Mccall, "A Chain-Model Genetic Algorithm for Bayesian Network Structure Learning," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, 2007.

[7] O. Regnier-Coudert and J. McCall, "Competing mutating agents for Bayesian network structure learning," in *International Conference on Parallel Problem Solving from Nature*, 2012.

[8] ——, "An island model genetic algorithm for Bayesian network structure learning," *IEEE Congress on Evolutionary Computation, CEC 2012*, pp. 1–8, 2012.

[9] S. Shakya, B. S. Lee, C. Di Cairano-Gilfedder, and G. Owusu, "Spares parts optimization for legacy telecom networks using a permutation-based evolutionary algorithm," in *2017 IEEE Congress on Evolutionary Computation, CEC 2017 - Proceedings*, 2017.

[10] U. Aickelin and K. A. Dowsland, "An indirect genetic algorithm for a nurse-scheduling problem," *Computers and Operations Research*, 2004.

[11] A. Sawczuk da Silva, Y. Mei, H. Ma, and M. Zhang, "A memetic algorithm-based indirect approach to web service composition," in *2016 IEEE Congress on Evolutionary Computation, CEC 2016*, 2016.

[12] U. Derigs and T. Döhmer, "Indirect search for the vehicle routing problem with pickup and delivery and time windows," *OR Spectrum*, 2008.

[13] J. Ceberio, A. Mendiburu, and J. A. Lozano, "Kernels of Mallows Models for Solving Permutation-based Problems," pp. 505–512, 2015.