

The Performance of Frequency Fitness Assignment on JSSP for Different Problem Instance Sizes

Iris Pijning¹^a, Levi Koppenhol¹^b, Danny Dijkzeul⁴^c,
Nielis Brouwer⁵^d, Sarah L. Thomson³^e and Daan van den Berg²^f

¹Master Information Studies, UvA Amsterdam

²VU Amsterdam, UvA Amsterdam

³Edinburgh Napier University, United Kingdom

⁴Cover Genius

⁵Rabobank

daan@yamasan.nl

Keywords: Optimization, Frequency Fitness Assignment, HillClimber, Job Shop Scheduling Problem, Neutrality

Abstract: This study compares the performance of the hillClimber algorithm to that of the hillClimber with a plugged in Frequency Fitness Assignment (FFA) method on the optimization of 240 Job Shop Scheduling Problem (JSSP) instances. The JSSP instances have been systematically generated in gridwise sizes to investigate the performance of the algorithms on problem instances with steadily increasing numbers of jobs and machines. The comparison of the FFA-hillClimber and the default hillClimber is done in both EQ setting, accepting equally good (or fitness-frequent) solutions, and NO setting, that only accepts improvement. FFA-hillClimbers are more successful than default hillClimbers on smaller problem instances, but not on larger ones. Results also suggest a function between the ratio between jobs and machines, number of evaluations, and the success of the respective algorithms.


1 The Job Shop Scheduling Problem


The Job Shop Scheduling Problem (JSSP) is a constrained optimization problem which entails minimizing the length, or *makespan* of a schedule with j jobs on m machines (Błażewicz et al., 1996; Weise et al., 2021). In JSSP, each job needs to be processed once on each machine exactly once, but what makes the problem hard is that each process has a predetermined processing time and that the processes have precedence constraints. This means, for example, that Job 1 must *first* be processed on Machine 0 for exactly 2 minutes, then on Machine 1 for exactly 5 minutes, and finally on Machine 2 for 9 minutes (see Fig. 1). No longer, no shorter, and in exactly that order. Pro-


cessing times are continuous and not divisible, but idle time on a machine between jobs is possible. Furthermore, a machine can only process one job at a time, and job can only be processed by one machine at a time (Weise et al., 2021; Jain and Meeran, 1999; de Bruin, 2022).


Practical applications do not require much imagination, as efficient scheduling of manufacturing processes is a way for businesses to reduce costs (Jain and Meeran, 1999). But also less intuitive and more mission-critical applications such as surgery scheduling in hospitals and clinics can be modeled as JSSP (Pham and Klinkert, 2008). Not only do surgical procedures make up a significant source of revenue (in some countries¹), but scheduling resources like personnel (surgeons, anaesthetists, nurses) and facilities (operating rooms, intensive care beds) make up a significant chunk of its costs.


Academic interest in the objective of schedule


^a <https://orcid.org/0000-0002-7551-1679>

^b <https://orcid.org/0000-0003-2356-184X>

^c <https://orcid.org/0000-0001-8185-1293>

^d <https://orcid.org/0000-0001-6269-1722>

^e <https://orcid.org/0000-0001-6971-7817>

^f <https://orcid.org/0000-0001-5060-3342>

¹Obviously, the objective of ‘revenue’ depends on a country’s health care system.

Job 0	Machine 2 (4 min)	Machine 0 (7 min)	Machine 1 (2 min)
Job 1	Machine 0 (2 min)	Machine 1 (5 min)	Machine 2 (9 min)
Job 2	Machine 0 (6 min)	Machine 2 (1 min)	Machine 1 (1 min)

Job 1	Job 1	Job 0	Job 2	Job 0	Job 0	Job 2	Job 1	Job 2
-------	-------	-------	-------	-------	-------	-------	-------	-------

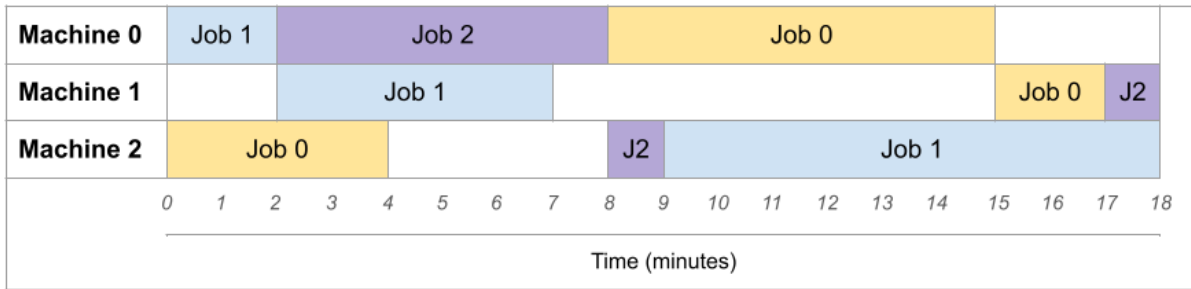


Figure 1: An example of a randomly generated problem instance (top), a random solution in permutation representation (middle), and the corresponding schedule for that solution permutation, determining its makespan (bottom).

makespan minimization stems back to at least the 1950s, when it was remarkably enough considered an easy problem. But since then, as along its factorial search space increase, (Eq. 1), JSSP has been proven to be an NP-Hard problem (Lawler et al., 1993; Lenstra and Kan, 1979). It is true that a superpolynomial search space increase in itself does not mean a problem is NP-hard, as Leonard Euler demonstrated in 1736 when solving the Bridges of Königsberg problem with a polynomial-time method. For NP-hard problems however, such an algorithm is not known, making these problems not solvable (meaning: finding the optimal solution) in any stretch of reasonable time for realistic instance sizes.

But even if a problem is NP-hard, it is not a final verdict on the hardness of an individual instance. Many nuances exist, like the solvability phase transition of a problem, along which the hardest instances can be found, while vast majority of the instances is relatively easy (Sleegers et al., 2022; Braam and van den Berg, 2022). For the number partition problem, which is classified as ‘weakly NP-hard’, the number of informational bits per integer, and even the *distribution* of informational bits over the integers exert influence on the instances’ computational hardness (Sazhinov et al., 2023).

For the JSSP, something similar is at play, as the ratio between the number of jobs j and the number of machines m in an instance also appear to play a

part in the difficulty of finding optimal or reasonable solutions for JSSP instances. In the extremes, when j/m is either really high or really low, “simple priority rules almost surely generate an optimal schedule” (Streeter and Smith, 2006). These results are very strong, and possibly related to Ruben Horn’s work on the number partition problem (Horn et al., 2024b; Horn et al., 2024a), but the converse is also true: when j/m is close to 1, the problem is likely hard (Streeter and Smith, 2006). Randomly generated schedules for these instances are likely further away from known optimal solutions than for instances with smaller and larger j/m ratios, and local optima in the solution landscaped are also known to be further from global optima (Streeter and Smith, 2006).

[randomness streeter]

However hard it may be to find an optimal solution for a JSSP instance, it is a problem that allows for easy generation of random initial solutions from which to start the heuristic optimization process, unlike problems like the traveling tournament problem where creating a random initial schedule that satisfies all constraints to be considered a valid schedule has even been called impossible (Verduin et al., 2023b; Verduin et al., 2023a; Verduin et al.,) and HP protein folding that suffers from the same problem (Jansen et al., 2023; Koutstaal et al., 2024; Kommandeur et al., 2024). For the JSSP, things are a lot easier as an initial random valid solution (a JSSP

schedule) can be created in linear time. Furthermore, there is also a deterministic constant time connective mutation type available, which is also not guaranteed (e.g. the traveling tournament problem and HP protein folding don't have one). Both the initialization and mutation procedures will be further explained in Sections 4 and 5.

When it comes to the question of data sets, a collected set of 242 benchmark instances is commonly used in JSSP research literature, courtesy of Jelke J. van Hoorn (Van Hoorn, 2018). In this set the number of jobs range between 6 and 100, and the number of machines between 5 and 20, with the smallest individual instance existing of 6 jobs on 6 machines and the largest 100 jobs on 20 machines (van Hoorn, 2015). The distribution of j and m is somewhat haphazard over the set, but this is understandable as the set is comprised of 8 earlier JSSP benchmark sets. The advantage is of course the reachable generality of comparisons across earlier studies. In another more recent study, custom problem instances are generated drawing jobs' processing times from different probability distributions, to more fully understand the landscape of possible JSSP instances (Strassl and Musliu, 2022). For our experiments, a set of JSSP problem instances with gradually increasing job and machine numbers is created for a more granular look into the effect of instance size, but also the aforementioned job/machine ratios, on hillClimber and FFA-hillClimber performance. Using newly created JSSP instances rather than a known benchmark set does mean that there are no known optimal makespans for the generated instances, but we won't need those, to compare the performance of the FFA-hillClimber to the hillClimber algorithm rather than to find optimal solutions. These instances and algorithms' source code is publicly available (Anonymous, 2024).

2 HillClimbers

Possibly the most elementary evolutionary algorithm is the $(1+1)$ EA, shorthand for "the new generation is chosen as the best individual of one parent and one child" (Droste et al., 2002), but colloquially known as the 'hillClimber' algorithm. HillClimbers exist in many variants, with best-first moves, proportional probability moves, variable mutability, random restarts and all sorts of other bells and whistles, but we will restrict this study to the *stochastic hillClimber*. In order to optimize a given problem, the stochastic hillClimber starts off with a initial valid random solution and tries to optimize the quality by making one mutation in each generation

and accepting the mutated solution if better. Moving through the solution space like this is also called the "Choose First Positive" (MacFarlane et al., 2010) or "first improvement" (Basseur and Goëffon, 2013) strategy, which means that an encountered new solution that changes the objective value positively, (or at least non-negatively). These algorithms usually perform well, but have the risk of getting stuck in a local optimum rather than moving towards the global optimum in the solution space (Dijkzeul et al., 2022; Russell and Norvig, 2010).

One decision that needs to be made when implementing a hillClimber algorithm is whether to accept only better solutions, or equally good solutions as well.

[BEGIN SARAH:

The choice of a neutral moves policy should depend on the extent of neutrality in the landscape. Existing literature indicates a non-trivial proportion of neutrality in the landscapes of JSSP (Weise et al., 2021; Tsogbetse et al., 2022) and, indeed, in scheduling problems more generally (Sutton, 2007). The presence of neutrality in fitness landscapes can be helpful (Yu and Miller, 2002) or unhelpful (Collins, 2005) to search; it is likely that this depends on the type of neutrality (Vanneschi et al., 2007) and design of the algorithm.

Conceptually, also accepting solutions with equal objective values might alleviate some of the risk of the hillClimber getting stuck in a local optimum. Indeed, this has been ratified in the literature: a study which systematically compared hill climbers with different pivot rules and neutral moves policies found accepting neutral moves to be advantageous to search (Basseur and Goëffon, 2015).

As it relates to design of the FFA hill climber, both choices for neutral moves policy seem to be applicable on JSSP (Weise et al., 2021; de Bruin, 2022; de Bruin et al., 2023). Weise et al. state that "A plateau of the objective function is also a plateau under FFA" (Weise et al., 2022). Indeed, because all members of a plateau will share the same frequency fitness value it seems that acceptance of solutions with equally-rare fitness may be necessary for escape from neutral regions. Nevertheless, we would like to study both neutral move policies in the interest of rigour.

:END SARAH]

Until now, these considerations have not been discussed in earlier studies of FFA for JSSP optimization. In this study we will, and we will label the settings as EQ for hillClimbers that DO accept equally good solutions, and NOEQ for hillClimbers that DO NOT accept equally good solutions. Turns out it makes quite a difference.

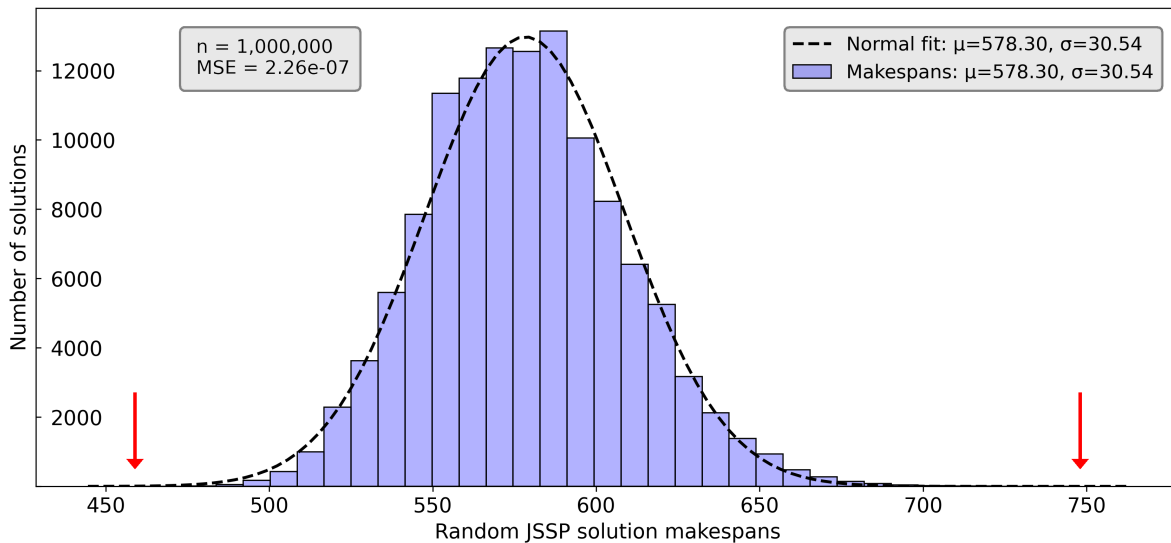


Figure 2: The distribution of makespans for 10^6 randomly generated unoptimized schedules for a JSSP instance with 50 jobs and 16 machines.

3 Frequency Fitness Assignment

“Frequency Fitness Assignment” is the brainchild of Thomas Weise, really. A publication in 2013 from his lab in HeFei University, China, introduced a new way to steer evolutionary algorithms through the search space of combinatorial optimization problems. The new ‘plugin’, Frequency Fitness Assignment (FFA), biases an evolutionary algorithm towards *rarer* objective values, rather than towards *better* objective values per se (Weise et al., 2013). The (a posteriori?) rationale behind this method is that “good solutions are indeed rare and the better the solutions get, the rarer they are” (Weise et al., 2021). Yet, surprisingly enough, the application of FFA in evolutionary algorithms has already shown good results on several optimization problems such as the traveling salesman problem (Liang et al., 2022; Liang et al., 2024) and HP protein folding (Koutstaal et al., 2024). The results on the job shop scheduling problem have been independently replicated by Ege de Bruin from Amsterdam’s VU university, and later published at EvoSTAR 2023 (de Bruin, 2022; de Bruin et al., 2023). Recently, more in-depth studies on the efficiency, algorithmic invariance under objective function transformations and explainability of its performance through entropy and search space trajectories have appeared, showing that the concept is slowly maturing from a wild proposal to a well-understood principle (Weise et al., 2020; Weise et al., 2022; Thomson et al., 2024).

When optimizing instances of the JSSP with a hillClimber algorithm (HC) and a hillClimber with

Frequency Fitness Assignment plugged into it (HC-FFA), FFA-hillClimber mainly shows good but not better results on most problem instances studied in both (Weise et al., 2021) and (de Bruin et al., 2023). However, the FFA-hillClimber does manage to outperform the classic hillClimber on some instances in these studies. One possible reason for the FFA-hillClimber outperforming the hillClimber is that the FFA-hillClimber does not get stuck in a local minimum like the hillClimber does (de Bruin et al., 2023). This work does however point out that, based on their results, the FFA-hillClimber seems more likely to outperform the hillClimber for JSSP on *smaller* problem instances, and less likely on larger instance sizes. This observation, and the validation or falsification of it, are the core issues of the paper you are currently reading. We aim to follow up on this reasoning by specifically comparing the performance of the hillClimber versus the FFA-hillClimber on JSSP instances, both when accepting equal solutions (EQ setting), or when only accepting better solutions (NOEQ setting), on a set of *systematically sized* JSSP instances.

4 Problem (instance) representation

Following the method used in previous studies of FFA on JSSP (Weise et al., 2021; de Bruin, 2022; de Bruin et al., 2023) to transform a problem instance into a valid schedule, the following constraints must be met:

- In each instance there are j jobs that must be processed on all m machines exactly once.

- Each job has to be processed on each machines in the order specified in the problem instance.
- Each job has a processing time on each machine specified in the problem instance.
- A job can be processed on only one machine at a time.
- A machine can process only one job at a time.

A single problem instance consists of a table of $m \times j$ entries holding two integers in each cell: the machineID and the processing time on that machine (Figure 1, top table). A solution to the problem instance can be given by a single permutation of $m \times j$ integers, all corresponding to a jobID (Figure 1, central array). Each jobID appears in the permutation m times, as is done in papers by Weise et al., De Bruin, and De Bruin, Thomson, and Van den Berg (Weise et al., 2021; de Bruin, 2022; de Bruin et al., 2023).

This representation can be seen as a list in which each jobID occurs m times. But since the processing order on the machines is a hard requirement for a job, the m entries in the permutation is identical. Switching two identical jobIDs from different places in the permutation therefore does not lead to a new solution (e.g. in Figure 1: Job 2 in 4th position and Job 2 in the 9th position). This reflects in the number of possible ‘reasonable’ schedules (meaning: without trivially unnecessary machine idle time), which is

$$\frac{(jm)!}{j(m)!} \quad (1)$$

The list of jobIDs, can give rise to a valid schedule in all of its permutations without further amends. Furthermore, all reasonable schedules can be represented by such a permutation. Finally, and this is neither trivial nor unimportant, a mutation exists that connects all representations into one connected combinatorial state space, making sure that at least principally, every solution is reachable from every other solution. That mutation is the swap mutation, which swaps two elements in the permutational solution representation. Other mutations, such as double swaps or triple shuffles, can also connect the entire combinatorial state space and might function better or worse, depending on the algorithmic deployment (e.g. simulated annealing or genetic algorithms might better use different mutation types). Finally, we should understand that having such a mutation is a luxury. Many constraint optimization problems, even from the same class, appear *not* to have such a mutation, making them practically much harder (cite TTP, protein folding)

Constructing the corresponding schedule from a permutational solution is relatively straightforward

and can be done in $O(n)$ time. Taking the example in Figure 1, Job 1 is the first job in the permutation and it can start right away at time 0 on Machine 0, occupying it for 2 minutes. Next to be placed in the schedule is again job 1, this time on Machine 1, and occupying it for 5 minutes. It can start at the first available minute on Machine 1 after both Job 1’s previous process and Machine 1’s previous job are finished. The latter of these is the strongest constraint in this case, and Job 1 can start immediately on Machine1 after it finishes its process on Machine 0. Note that if either Job 0 or Job 2 would have been wedged in between, we would have gotten the same final schedule. In other words: the permutation representation is somewhat redundant. This might cause some neutrality in the search space, although the effect might depend on the instance size.

After all jobs from the permutation are placed, the time it takes for all jobs to complete all their processes is called the *makespan* of the schedule. In the example in figure 1 the makespan is 18 minutes. Minimizing the makespan is the objective of a JSSP instance, and the makespan’s length is therefore its objective value.

5 Experiment

For our experiment, we generated JSSP instances with jobs $j \in \{5, 10, 15, \dots, 90, 95, 100\}$ jobs, and machines $m \in \{5, 6, 7, \dots, 23, 24, 25\}$, totalling $20 \times 21 = 420$ instances. These ranges of j and m completely envelop Weisal’s original study and De Bruin et al.’s replication, which both use Van Hoorn’s benchmark set (Weise et al., 2021; van Hoorn, 2015; Van Hoorn, 2018). For each job j , a random permutation of the m machines is assigned, after which each job process, for every machine, gets assigned a random duration of $1 \leq dur \leq 10$ minutes.

After creation, all 420 problem instances are solved with two algorithms, a default hillClimber and an FFA-hillClimber, both of which have two settings, eq/noeq. When switched to eq, the algorithm will accept equally good mutated solutions (or equally fitness-infrequent in case of FFA), but when switched to noeq it will only accept better solutions (or more fitness-infrequent in case of FFA). Our experimental setup is thereby slightly wider than earlier studies which only studied these algorithms in eq setting (Weise et al., 2021; de Bruin, 2022; de Bruin et al., 2023).

The default hillClimber (“the simplest local search possible” (Weise et al., 2021)) starts off with a single random but valid solution, and every iteration performs one mutation, implemented as a ‘job swap’.

The job swap operation randomly selects two different job indices in the permutational representation, and subsequently swaps these iff the jobIDs are different – else a new random index is selected for the second job. Iff the makespan of the newly mutated schedule is shorter than the incumbent schedule, the new schedule is accepted and replaces the incumbent schedule. If the hillClimber is in `eq` setting for this run, it will not only accept a better schedule (with shorter makespans), but also an equally good schedule.

The FFA-hillClimber also has an `eq - noeq` setting. As explained earlier, its FFA-plugin entails keeping a frequency log with every possible makespan value and how often it was encountered. It starts off with all log entries set to zero, after which it initializes a random solution, calculates its makespan value, and increases that value’s entry in the frequency log by 1. Each iteration, it mutates the incumbent schedule similar to the hillclimber: it randomly selects two different job indices in the permutation, and subsequently swaps these two jobs iff they have different jobIDs (or draws two new values otherwise). It then calculates the makespan of the new schedule, increases that makespan’s observed frequency in the log, looks whether this value was less encountered than the incumbent objective value and if so, accepts the mutated schedule as the new incumbent schedule. Different from the default hillClimber, the FFA-hillClimber also separately retains the best-so-far solution, which often is different from the incumbent solution. Finally, an FFA-hillClimber run can also be set to `eq`, thereby also accepting mutated solutions with makespans that are equally frequently encountered contrary to the `noeq` setting, which ensures accepting only less frequently encountered makespan values’ schedules.

All four algorithmic settings performed 3 runs of 10^6 function evaluations for each of the 420 problem instances. The runtime of each algorithm on all 420 problem instances is about 3.5 days on a local machine, meaning that the entire experiment took approximately 6 weeks. This is much fewer than previous studies, that usually deploy 10^9 function evaluations per problem instance (Weise et al., 2021; de Bruin, 2022; de Bruin et al., 2023). It has been pointed out that this high number of evaluations may turn the FFA-hillClimber algorithm into an almost “stochastically exhaustive search” (de Bruin et al., 2023). However, even on the scale of 10^6 function evaluations we do get some very interesting

	NOEQ	EQ
HC-FFA	161,028	158,420
HC	164,420	142,027

Table 1: Sums of the best makespans found for all 420 JSSP instances after one million evaluations for each of the four algorithmic settings versions. The default hillClimber showed both the worst and the best performance, and the `EQ` setting outperformed the `NOEQ` setting on these instances.

Results.

When comparing the absolute performance of all four algorithmic settings, the default hillClimber performs both best and worst. Summing up² all 420 average makespans gives 164,420 for the hillClimber that accepts only mutations that lead to better makespans (the `NOEQ` setting), which is the worst performing algorithmic setting. When the same hillClimber *does* accept equal-makespan-mutations however (the `EQ` setting), it becomes the best performing algorithmic setting with a total makespan of 142,027.

[**BEGIN SARAH:** This ratifies findings from the literature on other problems where acceptance of neutral moves has been found to be advantageous to search efficiency (Basseur and Goëffon, 2015).

:END SARAH]

When it comes to the FFA-HillClimber, it is again the `EQ` setting that outperforms the `NOEQ` setting, at 158,420 total makespan against 161,028 total makespan.

[**BEGIN SARAH:** This finding matches with the axiom mentioned in Section 2 that a plateau in objective function space is also a plateau with relation to FFA. It appears that the freedom of movement afforded by allowing moves to solutions with equally-rare fitness may be necessary to escape the plateaus.

:END SARAH]

On the larger scale of things, these differences can be regarded as quite small. If we would normalize the makespan of best algorithmic setting (HillClimber with `EQ`) to 1, the setting FFA-HillClimber with `EQ` would have 1.12, the setting FFA-HillClimber with `NOEQ` would have 1.13 and the worst setting, HillClimber with `NOEQ`, would have 1.16. These values are small in the context of optimization, where improvement in a run can easily lead up to 50% better objective values, even in the experiments from our study (Figures 5 and 6). The objective of this study however, was to gather insight on the dominance of the FFA-hillClimber over the default hillClimber (and vice versa) relative to the computational budget. In

²For suggestive reasons, we do not average these values; a makespan of 5 machines with 10 jobs is obviously lower than 5 machines with 100 jobs. The summed end result however, would not differ.

Number of evaluations	NOEQ			EQ		
	HC-FFA win	Tie	HC win	HC-FFA win	Tie	HC win
10,000	6.905%	0.238%	92.857%	0.476%	2.857%	96.667%
100,000	23.571%	4.286%	72.143%	0%	8.81%	91.19%
250,000	44.286%	4.286%	51.429%	0.476%	13.095%	86.429%
500,000	61.667%	4.048%	34.286%	0.952%	16.905%	82.143%
750,000	71.19%	4.286%	24.524%	0.952%	17.857%	81.19%
1,000,000	81.19%	4.524%	14.286%	1.19%	20.476%	78.333%

Table 2: The percentage of wins (or tie) per algorithm variant on increasing number of evaluations over all 420 JSSP instances. In the uphill variants, the FFA-hillClimber increasingly finds shorter makespans than the hillClimber as the number of evaluations increases. For the sideways variants, hillClimber wins decrease over evaluations, and the percentage of ties increases.

other words: these values could be strongly related to the exact budget of 10^6 evaluations. For 10^4 , 10^5 or 10^{10} evaluations, things could be quite different.

The advantage of creating our own benchmark set of 420 instances with the regularity $j \in \{5, 10, 15, \dots, 90, 95, 100\}$ jobs, and machines $m \in \{5, 6, 7, \dots, 23, 24, 25\}$ is that it allows for a comparison of the algorithmic settings’ performance in a grid view, with m on the horizontal axis, and j on the vertical axis (Figures 3 and 4). Coloring cell (20,55) red means that for the instance with 20 machines and 55 jobs, the FFA-hillClimber reached the best average performance after 3 runs of 10^6 generations. Coloring it blue means the default hillClimber delivered the best average performance for the same instance.

Taking this idea one step further, we also froze the runs after 10,000, 100,000, 250,000, 500,000 and 750,000 generations, creating an exact same grid view for different points in the run. When these intermediate grids are then placed in order from 10,000 generations to 1,000,000 generations, a clear picture emerges (Figures 3 and 4).

When in EQ-mode, the default hillClimber is the dominant algorithm throughout the runs; just a few red cells for very low numbers of jobs, mostly emerging later in the run (Fig. 3, percentages can be found in Table 2). The number of ties though, when the default hillClimber and the FFA-hillClimber increases slightly for lower numbers of machines – almost irrespective of the number of jobs. This might be taken as a suggestion that in the *very* long run, the dominance of the default hillClimber recedes.

When in NOEQ-mode, the picture is quite different. For low numbers of generations, the default hillClimber still dominates the grid, but throughout the evolutionary process, the FFA-hillClimber wins more and more terrain, starting with the lower numbers of jobs and machines but eventually taking over almost the entire grid at 1,000,000 evaluations (Fig. 3, percentages can be found in Table 2). It might therefore seem that the FFA-hillClimber is favourable in the

long run, but this is clearly not the case, because the *absolute* results still favour the default hillClimber, in EQ mode, over any other algorithmic setting (see Table 1 again). On the other hand again, it must be noted that these results only pertain to our experiment, and different numbers of generations might give different outcomes.

The convergence in Figures 5 and 5 illustrate the relative differences between hillClimber and FFA-hillClimber of either setting EQ or NOEQ. It becomes apparent that when hillClimber outperforms FFA-hillClimber, the instances are usually quite large. When FFA-hillClimber outperforms the default hillClimber, the instances are usually on the smaller side.

6 Conclusion and Discussion

For this benchmark set, using 1 million evaluations (or: generations), the performance ranking for the four algorithmic settings is clear (makespan lengths are normalized to facilitate comparison):

- (makespan = 1.00): HillClimber with EQ
- (makespan = 1.12): FFA-HillClimber with EQ
- (makespan = 1.13): FFA-HillClimber with NOEQ
- (makespan = 1.16): HillClimber with NOEQ

The mandatory nuance though, is that this indeed pertains 1 million evaluations. It is very likely that for other numbers of evaluations, the ranking might look quite different, possibly stronger in favour of FFA in both settings. Generally speaking, the more function evaluations, the better FFA performs.

These findings closely relate to De Bruin et al’s earlier observation and hypothesis, that plugging the FFA method into a hillClimber algorithm may result in a “stochastically exhaustive algorithm” (de Bruin et al., 2023). If this qualification is indeed truthful, FFA is expected to perform better on smaller JSSP instances, as these have smaller combinatorial

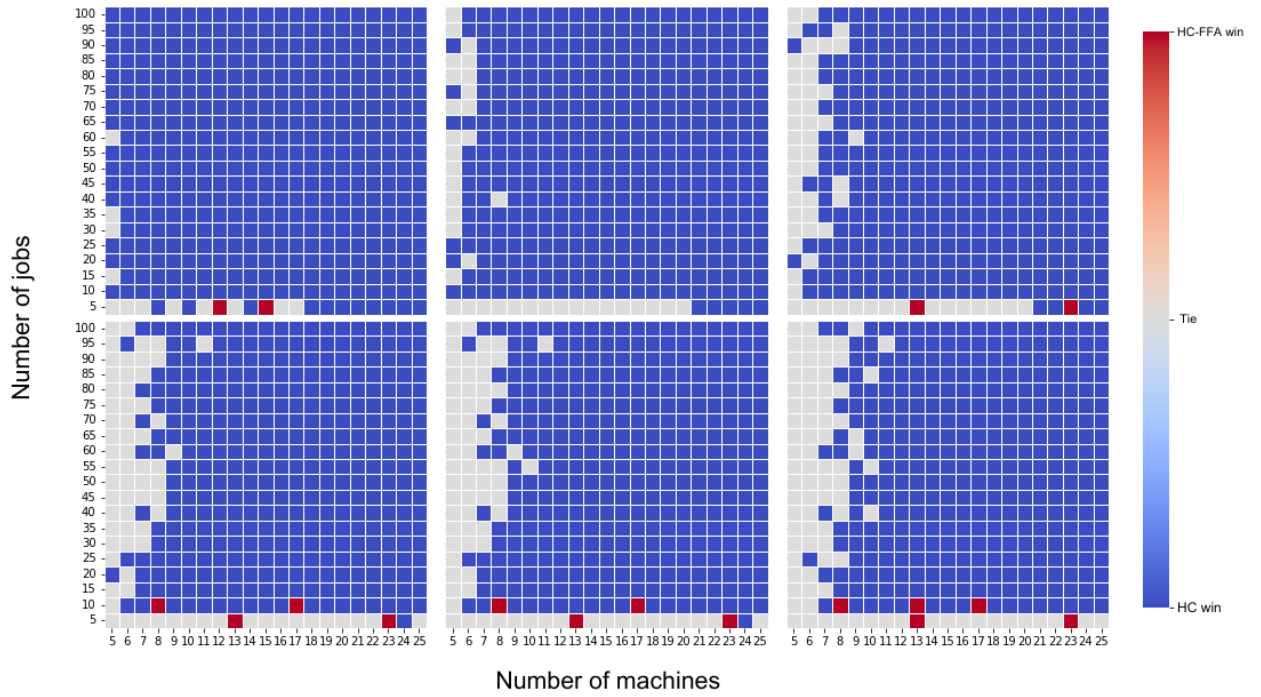


Figure 3: Best performance (or tie) per JSSP instance size for EQ settings. On the top row: the best found makespans at 10,000, 100,000, and 250,000 function evaluations. On the bottom row: those at 500,000, 750,000, and 1,000,000 evaluations.

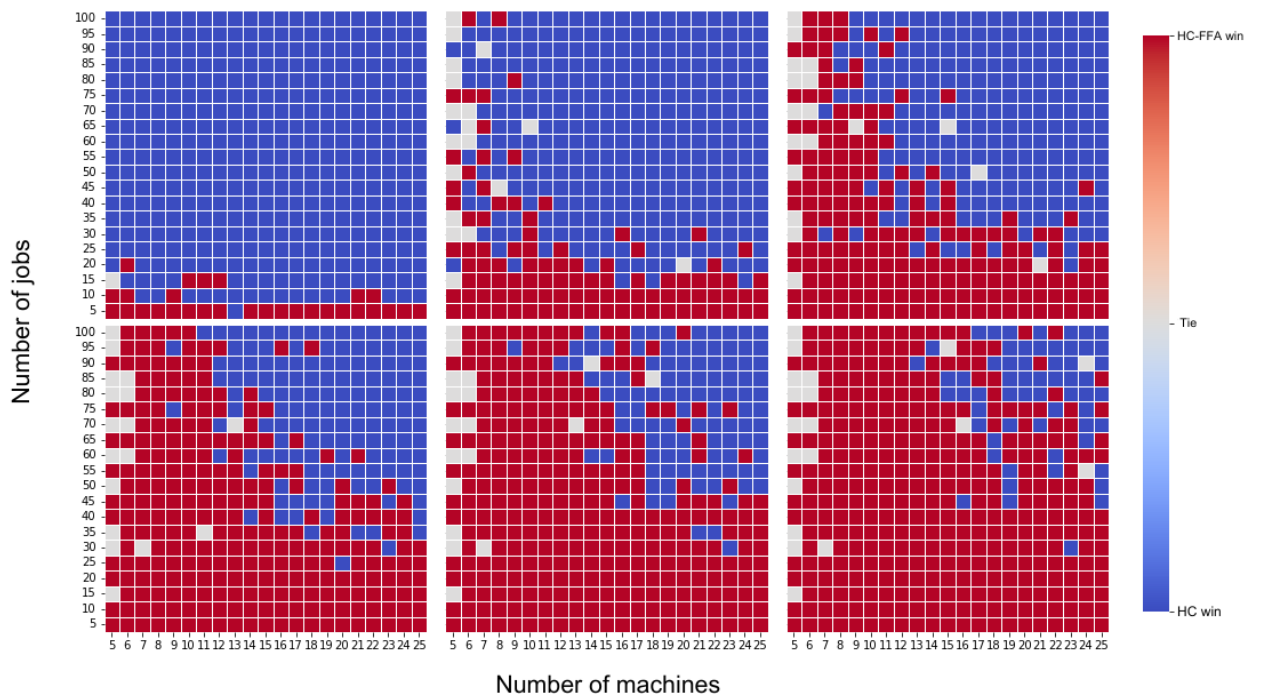


Figure 4: Best performance (or tie) per JSSP instance size for NOEQ settings. On the top row: the best found makespans at 10,000, 100,000, and 250,000 function evaluations. On the bottom row: those at 500,000, 750,000, and 1,000,000 evaluations.

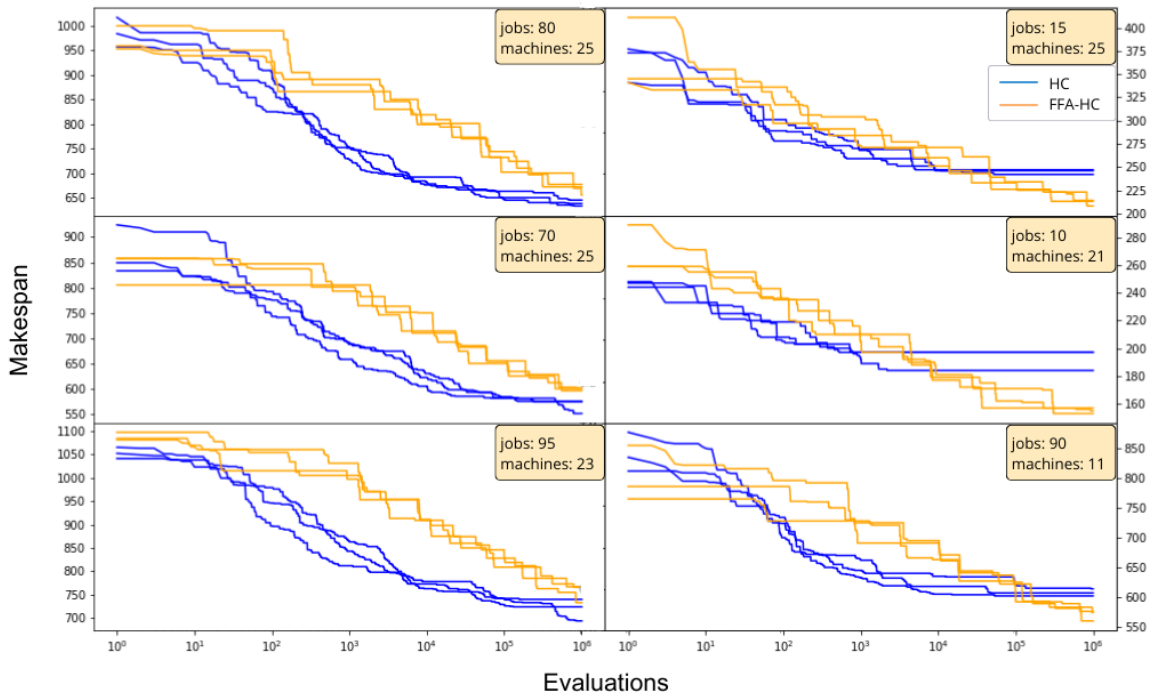


Figure 5: Some typical runs for the $NOEQ$ setting, with on the left hand side are the three instances where the default hillClimber outperformed the FFA-hillClimber with the biggest absolute difference. On the right hand side are the three instances where the FFA-hillClimber found the biggest improvements in makespan over the uphill hillClimber. The number of evaluations are on a logarithmic scale.

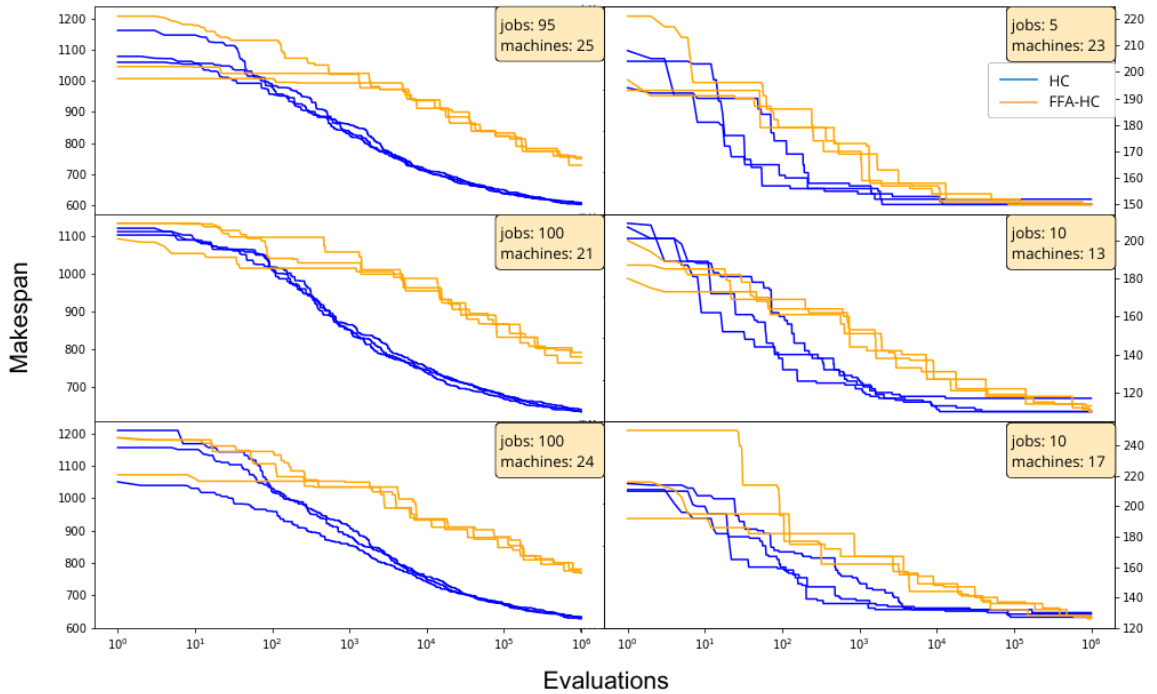


Figure 6: Some typical runs for the EQ setting, with on the left hand side are the three instances where the default hillClimber outperformed the FFA-hillClimber with the biggest absolute difference. On the right hand side are the three instances where the FFA-hillClimber found the biggest improvements in makespan over the uphill hillClimber. The number of evaluations are on a logarithmic scale.

search spaces, requiring fewer evaluations to exhaustively explore (either deterministically or stochastically). Our results appear that confirm this; the results in Figure 4 show that the FFA-hillClimber in NOEQ mode indeed overtakes the hillClimber in NOEQ mode for increasing numbers of evaluations, but more importantly: this process starts at the *smaller* instances, visualized by the red area progressively expanding from the bottom left.

This is also true for both algorithms in EQ mode, but the effect is much less pronounced, showing just a slight expansion of the grey area, signifying more ties, but no convincing dominance of FFA. Quite the contrary, actually. We are unsure why this happens EQ mode, but it might have to do with the smallness of the mutation, the neutrality of the landscape, or the relatively small number of evaluations (carefully denoting that a budget of 10^6 evaluations is only small compared to the regular budgets of FFA, which range to 10^9). We think it is well possible that for these algorithmic settings, FFA will also overtake at a budget of 10^9 evaluations, and have to rethink our current experimental setup then, because an estimated runtime of 12,000 weeks (231 years) on the current configuration might be a little over the top.

Another detail to note is the increase of *ties* instead of FFA-runs. Considering that these too are the smaller instances, it is possible that ties mean that in both settings the global optimum was found. Therefore, it is possible that we accidentally discovered that the hillClimber in EQ mode actually reaches a lot of global minima on these problem instances. There is no way to check this hypothesis (as this problem is still NP-hard), but the relatively low number of possible makespan values might justify an attempt with an exact algorithm to rigorously evaluate these problem instances.

Taking this thought one step further, it is quite possible, that for these low numbers of job duration (1 to 10 minutes), many global minima exist, similar to the number partition problem with many low integers (van den Berg and Adriaans, 2021; Sazhinov et al., 2023). So even if the combinatorial state space is sizeable, the number of global optima might be high too. In fact, if the partition problem is any measure to go by, it is possible that the number of global optima *increases* for larger instances if the range of processing times remains the same (Horn et al., 2024b; Horn et al., 2024a). This number might in fact be exponential, but that might still not be enough given the factorial nature of JSSP.

REFERENCES

- Anonymous (2024). Repository containing source material: https://github.com/thirteen-beans/JSSP_FFA.
- Basseur, M. and Goëffon, A. (2013). Hill-climbing strategies on various landscapes: an empirical comparison. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 479–486.
- Basseur, M. and Goëffon, A. (2015). Climbing combinatorial fitness landscapes. *Applied Soft Computing*, 30:688–704.
- Błażewicz, J., Domschke, W., and Pesch, E. (1996). The job shop scheduling problem: Conventional and new solution techniques. *European journal of operational research*, 93(1):1–33.
- Braam, F. and van den Berg, D. (2022). Which rectangle sets have perfect packings? *Operations Research Perspectives*, 9:100211.
- Collins, M. (2005). Finding needles in haystacks is harder with neutrality. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1613–1618.
- de Bruin, E. (2022). Escaping local optima by preferring rarity with the frequency fitness assignment. Master’s thesis, Vrije Universiteit Amsterdam.
- de Bruin, E., Thomson, S. L., and Berg, D. v. d. (2023). Frequency fitness assignment on jssp: A critical review. In *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*, pages 351–363. Springer.
- Dijkzeul, D., Brouwer, N., Pijning, I., Koppenhol, L., and Van den Berg, D. (2022). Painting with evolutionary algorithms. In *International Conference on Computational Intelligence in Music, Sound, Art and Design (Part of EvoStar)*, pages 52–67. Springer.
- Droste, S., Jansen, T., and Wegener, I. (2002). On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science*, 276(1-2):51–81.
- Horn, R., Jansen, R., van Eck, O., and van den Berg, D. (2024a). When being fair is hard: Predictability of yes- or no-instance type for number partitioning problems. (submitted).
- Horn, R., Thomson, S. L., van den Berg, D., and Adriaans, P. (2024b). The easiest hard problem: Now even easier the easiest hard problem, number partitioning problem, combinatorial optimization.
- Jain, A. S. and Meeran, S. (1999). Deterministic job-shop scheduling: Past, present and future. *European journal of operational research*, 113(2):390–434.
- Jansen, R., Horn, R., van Eck, O., Verduin, K., Thomson, S. L., and van den Berg, D. (2023). Can hp-protein folding be solved with genetic algorithms? maybe not.
- Kommandeur, J., Koutstaal, J., Timmer, R., Jansen, R., and Weise, T. (2024). Two fast but unsuccessful algorithms for generating randomly folded proteins in hp. *EvoStar LBAs*.
- Koutstaal, J., Kommandeur, J., Timmer, R., Horn, R., Thomson, S. L., and van den Berg, D. (2024). Frequency fitness assignment for untangling proteins in 2d. *EvoStar LBAs*.

- Lawler, E. L., Lenstra, J. K., Kan, A. H. R., and Shmoys, D. B. (1993). Sequencing and scheduling: Algorithms and complexity. *Handbooks in operations research and management science*, 4:445–522.
- Lenstra, J. K. and Kan, A. R. (1979). Computational complexity of discrete optimization problems. In *Annals of discrete mathematics*, volume 4, pages 121–140. Elsevier.
- Liang, T., Wu, Z., Lässig, J., van den Berg, D., Thomson, S. L., and Weise, T. (2024). Addressing the traveling salesperson problem with frequency fitness assignment and hybrid algorithms.
- Liang, T., Wu, Z., Lässig, J., van den Berg, D., and Weise, T. (2022). Solving the traveling salesperson problem using frequency fitness assignment. In *2022 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 360–367. IEEE.
- MacFarlane, A., Secker, A., May, P., and Timmis, J. (2010). An experimental comparison of a genetic algorithm and a hill-climber for term selection. *Journal of documentation*, 66(4):513–531.
- Pham, D.-N. and Klinkert, A. (2008). Surgical case scheduling as a generalized job shop scheduling problem. *European Journal of Operational Research*, 185(3):1011–1025.
- Russell, S. J. and Norvig, P. (2010). *Artificial intelligence a modern approach*. London.
- Sazhinov, N., Horn, R., Adriaans, P., and van den Berg, D. (2023). The partition problem, and how the distribution of input bits affects the solving process (submitted).
- Slegers, J., Thomson, S. L., and van Den Berg, D. (2022). Universally hard hamiltonian cycle problem instances.
- Strassl, S. and Musliu, N. (2022). Instance space analysis and algorithm selection for the job shop scheduling problem. *Computers & Operations Research*, 141:105661.
- Streeter, M. J. and Smith, S. F. (2006). How the landscape of random job shop scheduling instances depends on the ratio of jobs to machines. *Journal of Artificial Intelligence Research*, 26:247–287.
- Sutton, A. M. (2007). An analysis of search landscape neutrality in scheduling problems. In *Proceedings of the ICAPS*, page 79.
- Thomson, S. L., Ochoa, G., van den Berg, D., Liang, T., and Weise, T. (2024). Entropy, search trajectories, and explainability for frequency fitness assignment. to appear).
- Tsogbetse, I., Bernard, J., Manier, H., and Manier, M.-A. (2022). Impact of encoding and neighborhood on landscape analysis for the job shop scheduling problem. *IFAC-PapersOnLine*, 55(10):1237–1242.
- van den Berg, D. and Adriaans, P. (2021). Subset sum and the distribution of information. In *IJCCI*, pages 134–140.
- van Hoorn, J. J. (2015). Jobshop instances and solutions.
- Van Hoorn, J. J. (2018). The current state of bounds on benchmark instances of the job-shop scheduling problem. *Journal of Scheduling*, 21(1):127–128.
- Vanneschi, L., Tomassini, M., Collard, P., Vérel, S., Pirola, Y., and Mauri, G. (2007). A comprehensive view of fitness landscapes with neutrality and fitness clouds. In *Genetic Programming: 10th European Conference, EuroGP 2007, Valencia, Spain, April 11-13, 2007. Proceedings 10*, pages 241–250. Springer.
- Verduin, K., Horn, R., van Eck, O., Jansen, R., Weise, T., and van den Berg, D. The traveling tournament problem: Rows-first versus columns-first. *ICEIS 2024*, pages 447–455.
- Verduin, K., Thomson, S. L., and van den Berg, D. (2023a). Too constrained for genetic algorithms. too hard for evolutionary computing. the traveling tournament problem.
- Verduin, K., Weise, T., and van den Berg, D. (2023b). Why is the traveling tournament problem not solved with genetic algorithms?
- Weise, T., Li, X., Chen, Y., and Wu, Z. (2021). Solving job shop scheduling problems without using a bias for good solutions. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1459–1466.
- Weise, T., Wan, M., Wang, P., Tang, K., Devert, A., and Yao, X. (2013). Frequency fitness assignment. volume 18, pages 226–243. IEEE.
- Weise, T., Wu, Z., Li, X., and Chen, Y. (2020). Frequency fitness assignment: Making optimization algorithms invariant under bijective transformations of the objective function value. *IEEE Transactions on Evolutionary Computation*, 25(2):307–319.
- Weise, T., Wu, Z., Li, X., Chen, Y., and Lässig, J. (2022). Frequency fitness assignment: optimization without bias for good solutions can be efficient. *IEEE Transactions on Evolutionary Computation*.
- Yu, T. and Miller, J. (2002). Finding needles in haystacks is not hard with neutrality. In *Genetic Programming: 5th European Conference, EuroGP 2002 Kinsale, Ireland, April 3–5, 2002 Proceedings 5*, pages 13–25. Springer.