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

Iris Pijning[1][a], Levi Koppenhol[2][b], Danny Dijkzeul[4][c],
Nielis Brouwer[5][d], Sarah L. Thomson[3][e] and Daan van den Berg[2][f]

[1] *Master Information Studies, UvA Amsterdam, The Netherlands*
[2] *VU Amsterdam, The Netherlands*
[3] *Edinburgh Napier University, U.K.*
[4] *Cover Genius, The Netherlands*
[5] *Rabobank, The Netherlands*

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

Abstract: The Frequency Fitness Assignment (FFA) method steers evolutionary algorithms by objective *rareness* instead of objective *goodness*. Does this mean the size of the combinatorial search space influences its performance when compared to more traditional evolutionary algorithms? Our results suggest it does. To address to which extent the search space size matters for the effectiveness of the FFA-principle, we compare the algorithms on 420 Job Shop Scheduling Problem (JSSP) instances systematically generated in gridwise sizes. The comparison of the FFA-hillclimber and the standard hillclimber is done in both `EQ` setting, accepting equally good (or fitness-frequent) solutions, and `NOEQ` setting, only accepting improvement. FFA-hillclimbers are more successful than standard hillclimbers on smaller problem instances, but not on larger ones. It seems that the ratio between jobs and machines, influences the success of the respective algorithms for fixed computational budgets.

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 the JSSP, each job needs to be processed once on each machine exactly once, but what makes the problem hard is that a job's m processes have predetermined processing times and 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. A process is completed in one continuum and cannot be divided in separate parts, but idle time on a machine between jobs is possible. Furthermore, a machine can only process one job at a time, and a job can only be processed by one machine at a time (Weise et al., 2021; Jain and Meeran, 1999; de Bruin, 2022).

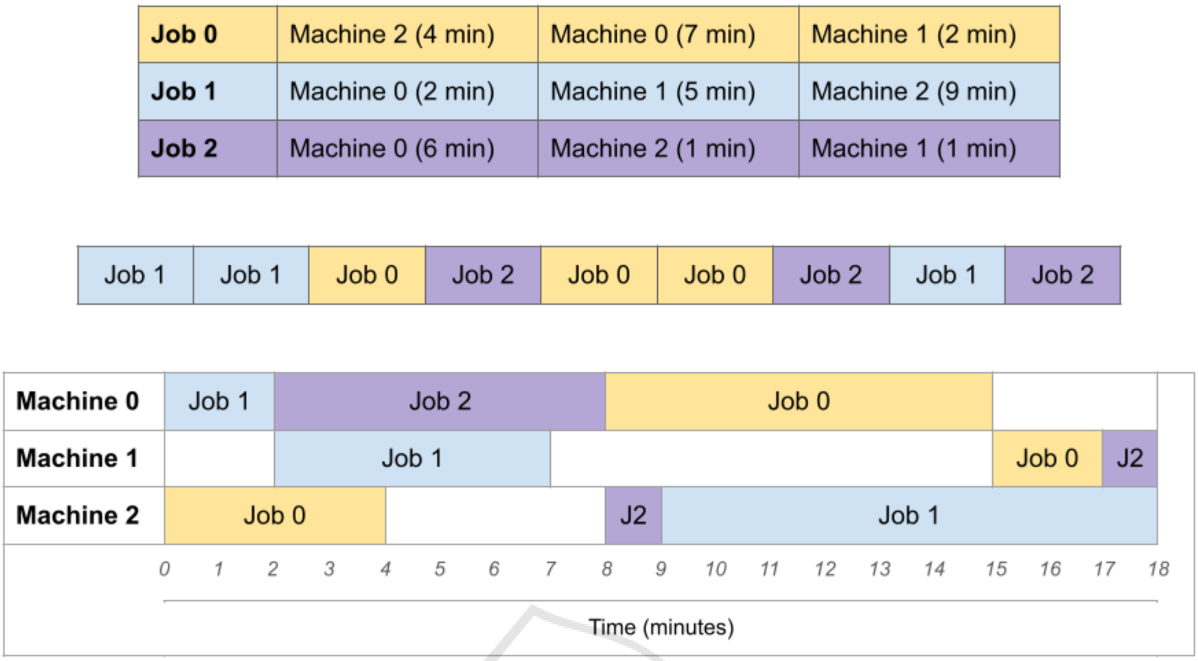
Practical applications do not require much imagination, as efficient scheduling of manufacturing processes is a way for businesses to reduce costs (Jain and Meeran, 1999). But also less intuitive and more mission-critical applications such as surgery scheduling in hospitals and clinics can be modeled as JSSP (Pham and Klinkert, 2008). Not only do surgical procedures make up a significant source of revenue (in some countries[1]), 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

[1] Obviously, the objective of 'revenue' depends on a country's health care system.

Pijning, I., Koppenhol, L., Dijkzeul, D., Brouwer, N., Thomson, S. and van den Berg, D.
The Performance of Frequency Fitness Assignment on JSSP for Different Problem Instance Sizes.
DOI: 10.5220/0012970500003837
In *Proceedings of the 16th International Joint Conference on Computational Intelligence (IJCCI 2024)*, pages 250-260
ISBN: 978-989-758-721-4; ISSN: 2184-3236



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, even though its search space increases factorially (See Eq. 1). A few decades later though, the JSSP was proven to be NP-Hard (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 Köningsberg 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 realistically sized instances.

But even if a problem is NP-hard, it is not a final verdict on the hardness of an individual instance. Many nuances exist, for example, the notion of a phase transition in a problem, (partially) separating the hardest instances from those that are trivially easy to solve (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 particular instances. In the extremes, when the ratio j/m is either really high or really low, "simple priority rules almost surely generate an optimal schedule" (Streeter and Smith, 2006). Furthermore, randomized initial solutions are already close to optimal, making this a region of easy instances for a variety of algorithms. 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 the ratio j/m is close to 1, the problem is likely hard (Streeter and Smith, 2006). Randomly generated schedules for instances with the ratio $j/m \approx 1$ are likely to be further away from known optimal solutions than for instances with smaller and larger j/m ratios. Local optima in the solution landscape are also known to be further away from global optima (Streeter and Smith, 2006).

However hard it may be to find an optimal solution for a JSSP instance, it does allow for easy generation of random initial solutions from which to start a heuristic optimization process. Although such a property feels natural to have, this is not the case; problems like the traveling tournament problem (Verduin et al., 2023b; Verduin et al., 2023a; Verduin et al., 2024) and HP protein folding (Jansen et al., 2023; Koutstaal et al., 2024; Kommandeur et al., 2024) lack a quick procedure for generating a uniformly random initial solution. 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 trivially 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 his set,the number of jobs range between 6 and 100, and the number of machines between 5 and 20, with the smallest instance consisting of 6 jobs on 6 machines and the largest consisting of 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 accross 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).

In this study however, we are less interested in the absolute performance of the algorithms, but more in how *instance size* (and thereby search space size) influences the performance of frequency fitness assignment (FFA). Is it really a "stochastic exhaustive search" as so aptly formulated by Ege de Bruin? (de Bruin et al., 2023a) We will find out by answering the main research question of this paper:

- How does instance size influence the performance of the Frequency Fitness Assignment (FFA) paradigm?

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. For a performance comparison of the FFA-hillclimber to the hillclimber however, we won't be needing those. The instances and algorithms' source code is publicly available (Pijning, 2024).

2 (FFA-)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 an initial valid random solution and tries to optimize the quality by making one mutation in each generation and accepting the mutated solution iff 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. These algorithms usually perform well, but have the risk of getting stuck in a local optimum rather than moving towards a 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. In analogy for FFA, the decision translates to whether the algorithm should accept only solutions with less encountered makespan values, or with equally often encountered makespan values as well.

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 the 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 hillclimbers 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-hillclimber, both choices for neutral moves policy seem to be applicable on JSSP (Weise et al., 2021; de Bruin, 2022; de Bruin et al., 2023b). Weise *et al.* state that "A plateau of the objective function is also a plateau un-

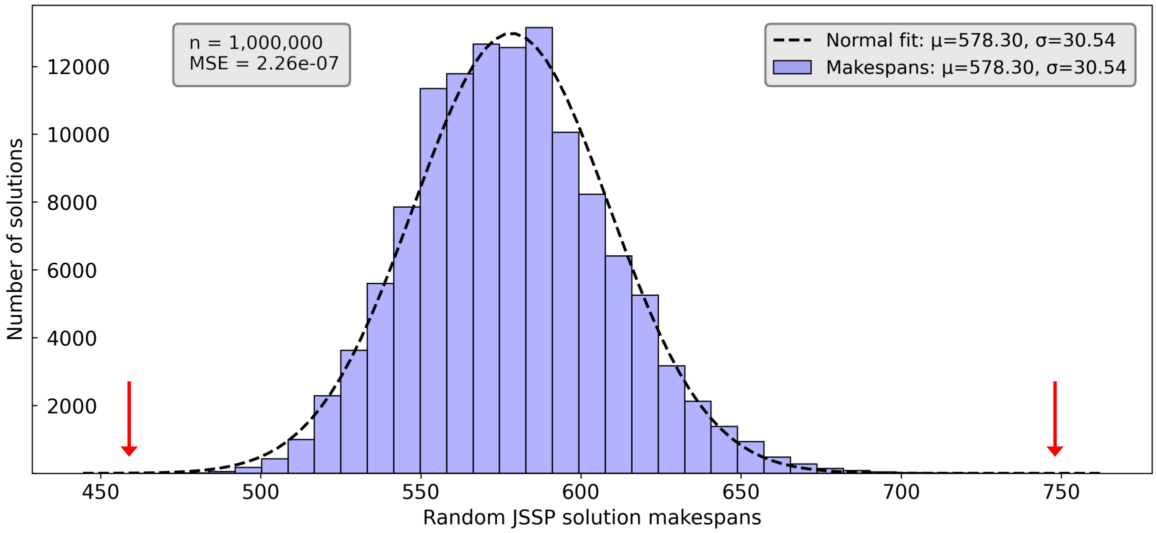


Figure 2: The distribution of makespans for 10^6 randomly generated unoptimized schedules for a JSSP instance with 50 jobs and 16 machines.

der 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.

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.

3 FREQUENCY FITNESS ASSIGNMENT

"Frequency Fitness Assignment" (FFA) 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 (Weise et al., 2013). The new 'plugin', Frequency Fitness Assignment, biases an evolutionary algorithm towards *rarer* objective values, rather than towards *better* objective values per se. The (a postiori?) 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). How universally applicable this rationale is remains open for debate, but in the mean time, 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., 2023b). 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 an instance of the JSSP with a hillclimber algorithm (HC) and a hillclimber with Frequency Fitness Assignment plugged into it (HC-FFA), the latter shows good but not better results on most problem instances studied in both (Weise et al., 2021) and (de Bruin et al., 2023b). However, the FFA-hillclimber does manage to outperform the classic hillclimber on some instances in these studies. One possible reason for this is that the FFA-hillclimber does not get stuck in local minima like the hillclimber does (de Bruin et al., 2023b). De Bruin et al.'s study does however point out that the FFA-hillclimber seems more likely to outperform the hillclimber for JSSP on *smaller* problem instances, and less so 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., 2023b) 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 machine 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 several earlier papers (Weise et al., 2021; de Bruin, 2022; de Bruin et al., 2023b).

Since the processing order on the machines is a hard requirement for a job, the m entries in the permutation are identical. Switching two identical jobIDs from different indexes in the permutation therefore does not lead to a new solution (e.g. in Figure 1: Job 2 in 4^{th} position and Job 2 in the 9^{th} position). This reflects in the number of possible 'reasonable' schedules (meaning: without trivially unnecessary machine idle time), which is

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

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. Finally, we should understand that having such an operation is a luxury; many constraint optimization problems appear *not* to have such a mutation, making them practically much harder (Verduin et al., 2023a; Jansen et al., 2023).

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 $j \in \{5, 10, 15, ..., 90, 95, 100\}$ jobs and $m \in \{5, 6, 7..., 23, 24, 25\}$ machines in all combinations, totalling $20 \times 21 = 420$ instances. These ranges of j and m completely envelop Weise et al.'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 attacked with two algorithms, a standard hillclimber and an FFA-hillclimber, both of which have two settings, EQ and 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., 2023b). Both the steering of an algorithm and its EQ - NOEQ setting are fixed aforehand, and do not change during a run.

The standard hillclimber ("the simplest local search possible" (Weise et al., 2021)) starts off with a single random but valid solution, and in every generation 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. Its FFA-plugin entails keeping a frequency log with every encountered 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 generation, it mutates the incumbent schedule identical to the hillclimber, randomly selecting two different job indices in the permutational representation, and subsequently swapping these iff the jobIDs are different (else a new random index is selected for the second job). It then calculates the makespan of the new schedule, increases that makespans 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 standard 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; this is contrary to the NOEQ setting, which ensures accepting only less frequently encountered makespan values' schedules.

In all four algorithmic settings, 3 runs of 10^6 function evaluations were completed for each of the 420 problem instances. The runtime of each algorithm on all 420 problem instances is about 3.5 days on a standalone 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., 2023b). 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., 2023b). However, even on the scale of 10^6 function evaluations we do get some very interesting

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.

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

6 RESULTS

When comparing the absolute performance of all four algorithmic settings, the standard hillclimber performs both best and worst. Summing up[2] 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. 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).

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. 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.

On the larger scale of things, these differences can be regarded as quite small. If we would rescale the makespan of best algorithmic setting (hillclimber with EQ) to 1, the setting FFA-hillclimber with EQ

[2] 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 if doing so however, would not differ. We *do* average results over runs with the same parameters though.

Table 2: The percentage of wins (or tie) per algorithmic setting for increasing number of evaluations over all 420 JSSP instances. In the NOEQ settings, the FFA-hilcClimber increasingly outperforms the hillclimber as the number of evaluations increases. For the EQ settings, hillclimber wins recede over evaluations as the percentage of ties increases.

	NOEQ			EQ		
Number of evaluations	**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%

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 this study, where improvement in a run can easily lead up to 50% better objective values (Figures 5 and 6). The objective of this study however, was to gather insight on the dominance of the FFA-hillclimber over the standard hillclimber (and vice versa) relative to the computational budget. In 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 $j \in \{5, 10, 15, ..., 90, 95, 100\}$ jobs, and machines $m \in \{5, 6, 7..., 23, 24, 25\}$ 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 standard 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 standard 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 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 standard hillclimber recedes.

When in NOEQ-mode, the picture is quite different. For low numbers of generations, the standard 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 standard 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 6 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 standard hillclimber, the instances are usually on the smaller side.

7 CONCLUSION AND DISCUSSION

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

1. (makespan = 1.00): hillclimber with EQ
2. (makespan = 1.12): FFA-hillclimber with EQ
3. (makespan = 1.13): FFA-hillclimber with NOEQ
4. (makespan = 1.16): hillclimber with NOEQ

The mandatory nuance though, is that this indeed pertains to 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

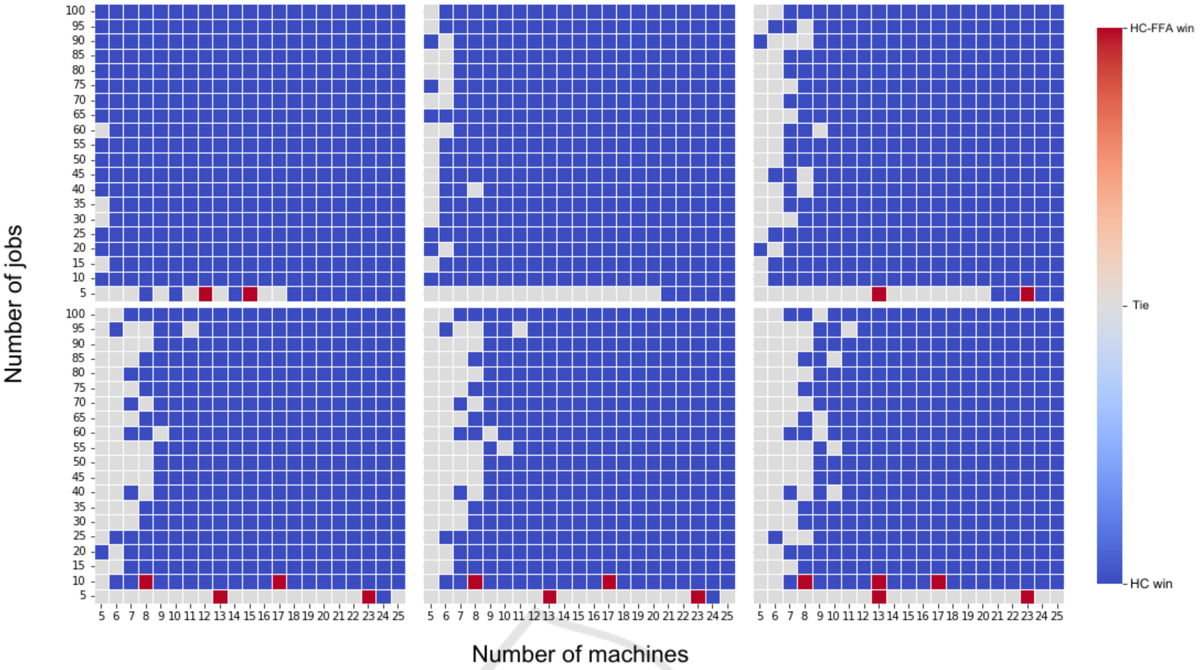


Figure 3: Best performance (or tie) per JSSP instance size for EQ settings. On the top row: the best found makespans at 10,000, 100,000, and 250,000 function evaluations. On the bottom row: those at 500,000, 750,000, and 1,000,000 evaluations.

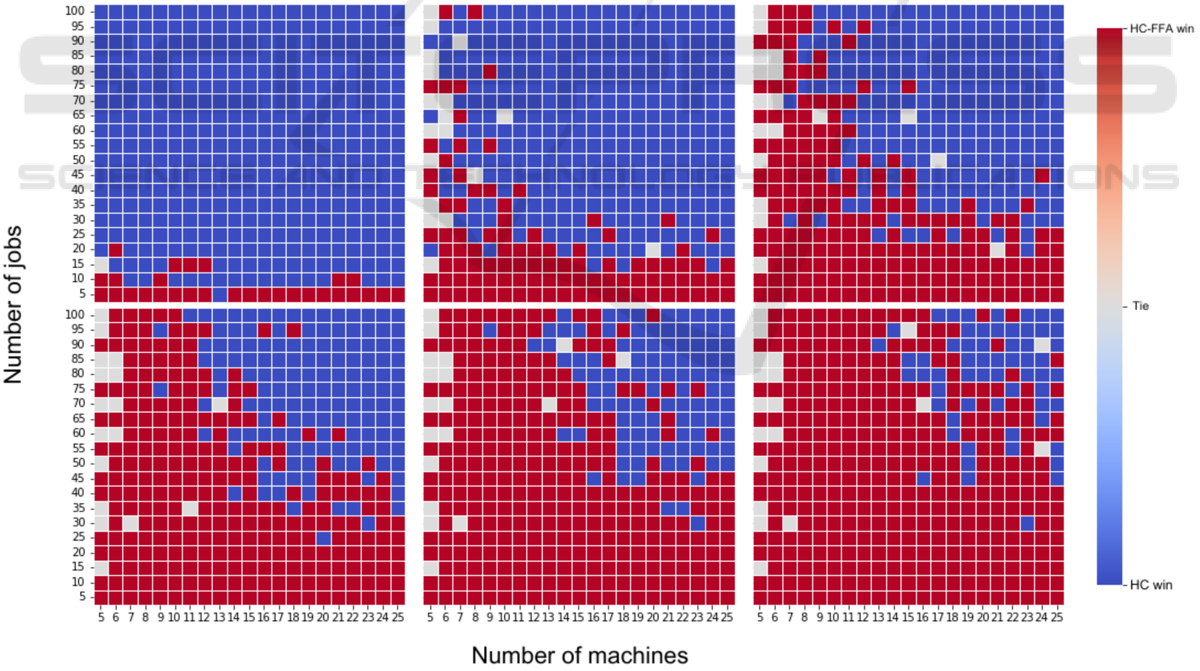


Figure 4: Best performance (or tie) per JSSP instance size for NOEQ settings. On the top row: the best found makespans at 10,000, 100,000, and 250,000 function evaluations. On the bottom row: those at 500,000, 750,000, and 1,000,000 evaluations.

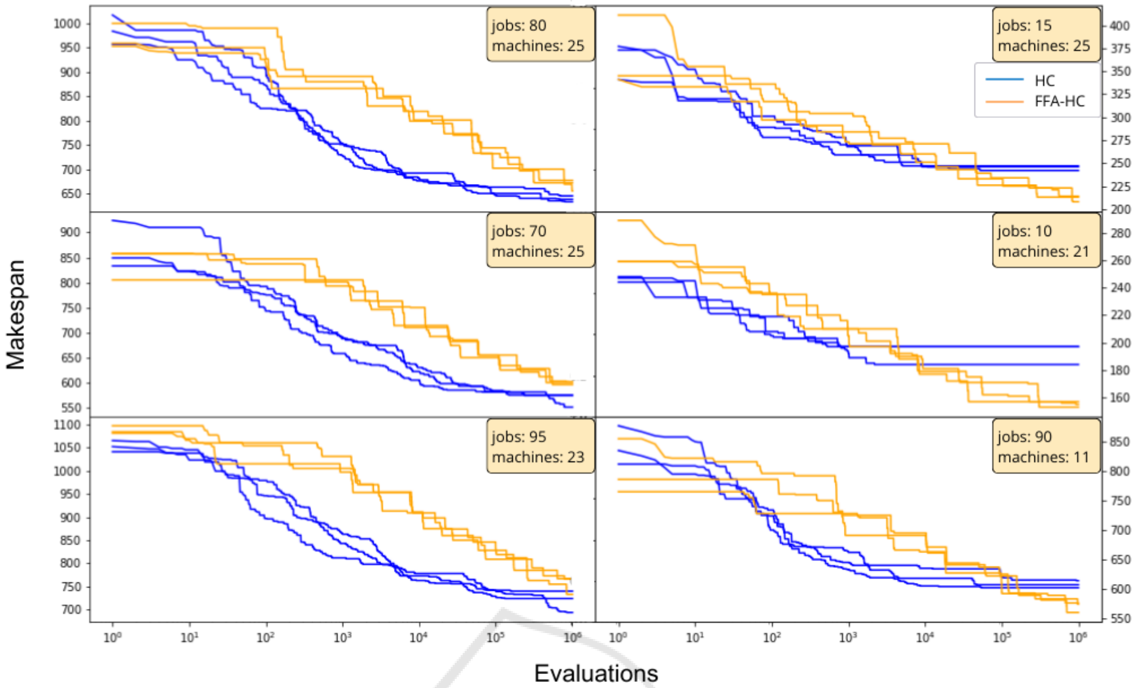


Figure 5: Some typical runs for the NOEQ setting, with on the left hand side the three instances where the standard hillclimber outperformed the FFA-hillclimber with the biggest absolute difference. On the right hand side are the three instances where the FFA-hillclimber found the biggest improvements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

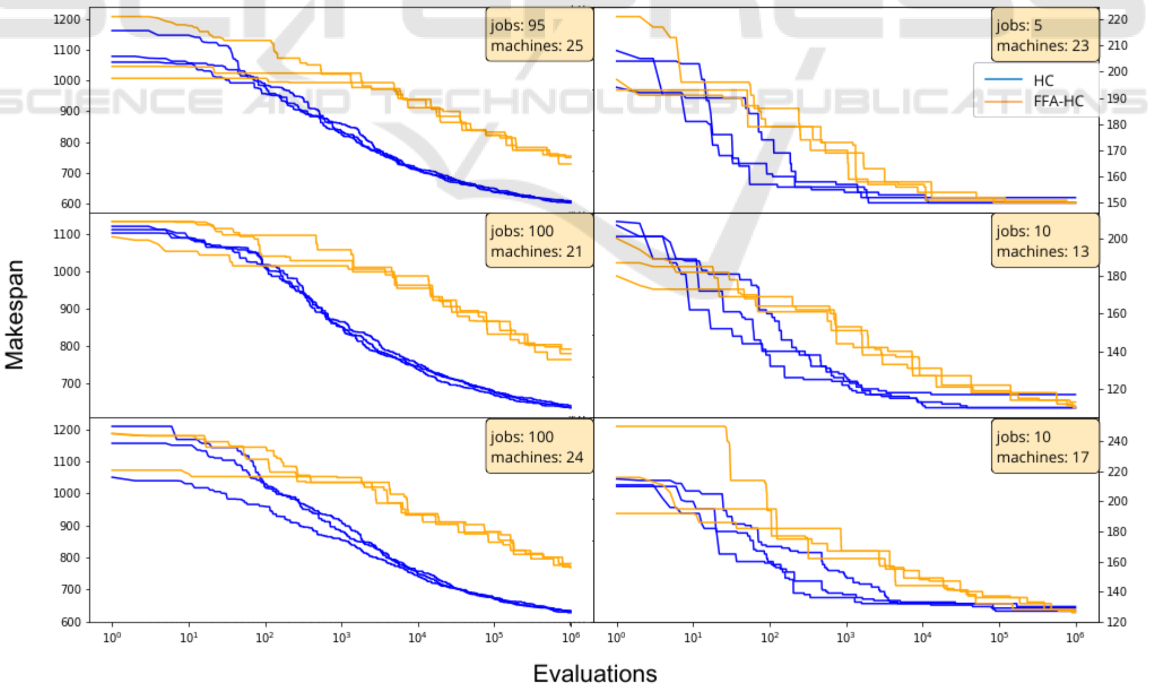


Figure 6: Some typical runs for the EQ setting, with on the left hand side the three instances where the standard hillclimber outperformed the FFA-hillclimber with the biggest absolute difference. On the right hand side are the three instances where the FFA-hillclimber found the biggest improvements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

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., 2023b). If this qualification is indeed truthful, FFA is expected to perform better on smaller JSSP instances, as these have smaller combinatorial search spaces, requiring fewer evaluations to exhaustively explore (either deterministically or stochastically). Our results appear to 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. We are unsure why this happens in `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 for this setting.

The increase of *ties* might also signal a degree of convexity. Considering that these too are found in the smaller instances, it is possible that ties simply 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 structurally 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). The increase might in fact be exponential, but that might still not be enough given the factorial nature of JSSP.

In future work, it would be nice do quantify the patterns, mainly in Figure 4, possibly in a way akin to (Koppenhol et al., 2022). It is well possible, that the exact state space size for each grid makes a critical difference, but also increasing the number of runs is necessary. Also, the *margin* of a win could be calculated and, in an ideal case, even compared to a known global optimum.

REFERENCES

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. (2023a). Frequency fitness assignment on jssp: A critical review. In *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*, pages 351–363. Springer.

de Bruin, E., Thomson, S. L., and Berg, D. v. d. (2023b). 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, combi-natorial 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*.

Koppenhol, L., Brouwer, N., Dijkzeul, D., Pijning, I., Sleegers, J., and Van Den Berg, D. (2022). Exactly characterizable parameter settings in a crossoverless evolutionary algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1640–1649.

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.

Pijning, I. (2024). https://github.com/irispijning/jssp_ffa.

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).

Sleegers, 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. (2024). 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.