University of Leeds
SCHOOL OF COMPUTER STUDIES
RESEARCH REPORT SERIES
Report 96.26

Succeed-first or Fail-first: A Case Study in Variable and Value Ordering[1]

by

Barbara M. Smith
Division of Artificial Intelligence

September 1996

[1]A revised version of a paper presented at the ILOG Solver and ILOG Schedule 2nd International Users' Conference, Paris, July 1996

Abstract

It is well known that appropriate variable and value ordering heuristics are often crucial when solving constraint satisfaction problems. A variable ordering heuristic which is often recommended, and is often successful, is based on the 'fail-first' principle: choose next the variable with the smallest remaining domain. General-purpose value ordering heuristics are less common, but it has been argued that a value which has least effect on future choices should be chosen, a kind of 'succeed-first' strategy.

This paper considers variable and value ordering heuristics for the car sequencing problem. A number of cars are to be made on a production line: each of them may require one or more options which are installed at different stations on the line. The option stations have lower capacity than the rest of the production line, e.g. a station may be able to cope with at most one car out of every two. The cars are to be arranged in sequence so that these capacities are not exceeded.

The choice of variable and value ordering heuristics has a dramatic effect on solution time for this problem. However, the fail-first variable ordering heuristic does not give good results: and in fact it is shown that dynamic variable ordering is unsuitable for this problem. Similarly, succeed-first value ordering does not work, and an ordering based on fail-first is a better choice. Reasons why conventional wisdom fails in this case, and could be expected to fail in similar cases, are identified.

1 Introduction

Many scheduling and similar problems can be expressed as constraint satisfaction problems (CSPs), in which there is a set of *variables*, each with a finite set of possible *values*, (its *domain*), and a set of constraints. Each constraint affects a subset of the variables and restricts the values that those variables can simultaneously take. A solution to a CSP is an assignment of a value to each variable such that every constraint is satisfied.

Constraint programming tools such as CHIP [12] and ILOG Solver [10] use a search algorithm based on forward checking [6] to solve CSPs. Such an algorithm maintains a partial solution (initially empty) which satisfies all the constraints, and attempts to extend it. The algorithm chooses a variable which is not currently assigned, and chooses a value from its domain to assign to it. The constraints are then used to propagate the effects of this assignment, and the previous assignments, to those variables which are still unassigned: any inconsistent values are removed from their domains. If the domain of an unassigned variable becomes empty, the current assignment is undone, the previous state of the domains is restored and an alternative assignment is tried, if necessary backtracking to a previous variable. The algorithm proceeds in this fashion until a complete solution is found or all possible assignments have been tried unsuccessfully, in which case there is no solution to the problem.

This algorithm involves two choices at each iteration: the next variable to assign, and the value to assign to it. How these choices are to be made is not specified in the algorithm; it is up to the programmer to decide. The order in which the variables and their values are considered can be decided in advance (a static ordering) or dynamically, using information available at the time that the choice is made. These choices can have a dramatic effect on the time taken to find a solution to a CSP: if the problem is large, so that a complete search is impracticable, they can determine whether or not a solution can be found at all. The choice of variable affects the extent to which the domains of the remaining variables are pruned, and hence how many branches remain to be explored; the choice of value will determine whether or not an unsuccessful branch has to be explored before backtracking to this variable to try again, if not all of the values of the current variable lead to a solution.

In choosing the next variable, we want to minimize the size of the search tree and to ensure that any branch that does not lead to a solution is pruned as early as possible. This was termed the 'fail-first' principle by Haralick and Elliott [6], described as "To succeed, try first where you are most likely to fail." In forward checking, one way of expressing this is as a dynamic variable ordering heuristic which chooses next the variable with smallest remaining domain. This is cheap to implement, and often very successful. The 'fail-first' principle is, however, more general than this heuristic, and can be applied in more problem-specific ways. For instance, we can (perhaps as a tie-breaker) choose a variable which has the greatest pruning effect on the domains of the future variables, because it is involved in many constraints and/or some of the constraints are relatively hard to satisfy.

In choosing the value to assign to the chosen variable, a good general strategy is to choose, if possible, a value which is likely to lead to a solution, and so reduce the risk of having to backtrack to this variable and try an alternative value. In practice, of course, the best we can normally do is to choose the value which seems least likely to lead to an immediate failure. This principle, which might be termed 'succeed-first', has not lead to widely-applicable value ordering heuristics comparable to the smallest-domain-first heuristic, but can give good heuristics tailored to individual problems, or types of problem.

In spite of the importance of variable and value ordering heuristics, little advice is available on how to choose them, beyond the summary just given. Finding good heuristics for a particular problem is often a matter of intuition, or trial-and-error. Furthermore, as will be seen from the case study which follows, even the basic principles, i.e. fail-first for variable ordering, succeed-first for value ordering, do not always apply.

2 The Car Sequencing Problem

The car sequencing problem arises from the manufacture of cars on an assembly line; the following description is based on that given by Parrello, Kabat and Wos [9]. A number of cars are to be produced; they are not identical, because different options are available as variants on the basic model. The assembly line has different stations which install the various options (air-conditioning, sun-roof, etc.). These stations have been designed to handle at most a certain percentage of the cars passing along the assembly line. Furthermore, the cars requiring a certain option must not be bunched together, otherwise the station will not be able to cope. Consequently, the cars must be arranged in a sequence so that the capacity of each station is never exceeded. For instance, if a particular station can only cope with at most half of the cars passing along the line, the sequence must be built so that at most 1 car in any 2 requires that option.

From the description of the car production scheduling at Renault given in [1], it seems that this is a simplification of the real problem. Nevertheless, it is hopefully not too far distant from reality, and presents some interesting features which may be relevant to other contexts.

The problem described by Parrello *et al.* was subsequently considered by Dincbas, Simonis and van Hentenryck [4] who showed that it could be formulated as a constraint satisfaction problem, using CHIP. Their basic formulation is described below. The example assembly line has 5 possible options, with the capacity of the corresponding station ranging from 1 car in any 5 to 2 cars in any 3. Given the specifications (in terms of options required) for N cars, we have to arrange these into a sequence such that none of the capacity constraints is violated.

		Class											
Option	*Constraint*	1	2	3	4	5	6	7	8	9	10	11	12
1	1/2		●	●				●	●	●			●
2	2/3	●		●	●	●		●				●	●
3	1/3		●					●		●	●	●	
4	2/5				●		●		●				●
5	1/5		●			●							
	no. of cars	3	1	2	4	3	3	2	1	1	2	2	1

Table 1: A sample problem with 25 cars and 5 options (● indicates that the class of car requires the option).

	Class																								
Option	7	4	8	5	3	10	12	4	2	1	3	10	4	4	9	5	1	6	7	1	6	11	5	6	11
1	●		●		●		●		●		●				●				●						
2	●	●		●	●		●	●		●	●		●	●		●	●		●	●		●	●		●
3	●					●			●			●			●				●			●			●
4		●	●				●	●					●	●				●			●			●	
5				●					●							●							●		

Table 2: A valid sequence for the sample problem.

3 Formulation

Following Dincbas *et al.*, the first step is to group the cars into classes (as in Table 1), such that the cars in each class all require the same options. Then we create N constrained variables $S(i)$, corresponding to the N slots in the sequence, and allocate a class to each slot. If there are M classes, the domain of each variable is $\{1, ...M\}$.

We also need additional constrained variables $O(i,j)$, with domains $\{0,1\}$, to represent whether or not the car in slot i requires option j.

The constraints of the problem are:

- the number of slots allocated a particular value (i.e. class) is at most the number of cars in that class.
- for all i, j: if $S(i)$ is assigned the value k, then $O(i,j) = 1$ if class k requires option j, 0 otherwise. (These constraints link the two sets of variables.)

- for all j: if option j has a capacity of M_j cars out of N_j, then the sum of any set of $O(i, j)$ variables corresponding to N_j consecutive slots in the sequence must be at most M_j.

These constraints can be easily expressed in constraint pogramming tools such as ILOG Solver or CHIP. Table 2 shows a sequence for the 25 cars shown in Table 1 which satisfies the constraints.

4 An Alternative Formulation

An alternative way of formulating the problem would be to use the variables to represent the cars, with their values being the slots in the sequence. A disadvantage of this formulation is that cars of the same class would give rise to symmetrical solutions, i.e. in any solution, cars requiring the same options can be exchanged without violating the constraints. Symmetries of this kind can lead to wasted search, considering infeasible partial solutions which are essentially the same, and should be avoided. Furthermore, the number of possible assignments would be N^N, rather than M^N when the cars are grouped into M classes. However, if there were more options the advantages of grouping the cars into classes might disappear, since cars requiring the same set of options would be less likely to occur; with only 5 options, there are only 32 possible classes, so that duplication is certain if there are more than 32 cars.

The constraints would be more difficult to express in this formulation, and would give rise to a large number of individual constraints: in particular the '2 out of 3' and '2 out of 5' options would each give rise to a set of ternary constraints, one for every three cars requiring the option.

5 Implied Constraints

Although the constraints already stated are sufficient to express the problem, [4] suggests adding implied constraints in order to allow failures to be detected earlier than would otherwise be possible.[2] The constraints given above suggest that the only important thing about the option capacities is not to exceed them, and going *below* capacity does not matter. This is not true, because of the fact that a certain number of cars requiring each option have to be fitted into the sequence, so that going below capacity in one part of the sequence may make it impossible to avoid exceeding the capacity elsewhere. Hence, there are implied constraints which have not yet been expressed.

For instance, suppose there are 30 cars, and 12 of them require option 1, which has capacity 1 car in any 2. Then at least one of cars 1 to 8 must require option 1; otherwise 12 of cars 9 to 30 will require option 1, which violates the capacity constraint. Similarly,

[2]Dincbas *et al.* call these *redundant constraints*, but in Operational Research redundant constraints mean constraints which can be removed without affecting the search in any way, and the term *implied constraints* is a better term for what is meant here.

cars 1 to 10 must include at least two option 1 cars, ... , and cars 1 to 28 must include at least 11 of the option 1 cars.

With these implied constraints, it is claimed in [4] that up to 100 cars can be sequenced in an average of less than a minute, when the average utilization of the options is about 70%, and 200 cars in an average of about 5 minutes, with average utilization up to 80%. (These results were based on randomly-generated problems, using the same 5 options as before.)

6 Variable Ordering

Dincbas *et al.* do not mention variable or value ordering heuristics, although as discussed in section 1, they can have a great effect on the time taken to solve a CSP. In this case, if no ordering heuristics are specified, so that the slots in the sequence are filled in numerical order (i.e. the variables are assigned in the order in which they are defined), and the values are assigned in the order in which they appear in the data (essentially in a random order), solving problems with 50 cars or more is very slow, requiring a great deal of backtracking, unlike the performance found in [4].

Given the discussion in section 1, it is natural to try the smallest-domain-first variable ordering heuristic on any CSP, and this is recommended for the car sequencing problem in a later paper by van Hentenryck, Simonis and Dincbas [13]. In fact, it may not obvious why this heuristic would not choose the variables consecutively, since if the first variable is assigned a value first, the capacity constraints will reduce the domains only of the next few variables, depending on the options required by the selected car. It might be expected that the second variable will then have the smallest domain (or one of the smallest) and will be assigned next, and so on. However, the implied constraints can lead to variables being considered in a different order. Suppose we consider an extreme case, in which the '1 out of 3' option is 100% utilized, so that every third slot in the sequence must be assigned a car requiring this option. If one of these cars is assigned to the first slot, then the implied constraints will show that the 4th variable must also be assigned a car with this option; this variable will thus have the smallest domain. So the 4th variable will be assigned next, then the 7th and so on.

In experiments, the fail-first heuristic gave poor results, despite the experience reported in [13]. Intuitively, leaving gaps, in the fashion just described, seems likely to cause difficulties in completing the sequence. The cars to be fitted into the gaps have to be compatible with the cars already placed before and after them, which is less likely to be possible than if car has only to be compatible with the preceding cars.

These considerations suggest that the variables should be assigned consecutively.[3] Any kind of dynamic variable ordering, which will leave gaps in the sequence, is likely to make finding a solution more difficult rather than less.

Moreover, the implied constraints, as defined in section 5, only make sense if the variables are to be considered in numerical order. There are many other implied constraints

[3]This would not preclude, for instance, starting in the middle of the sequence, and working backwards and forwards alternately, as long as no gaps are left.

that could be derived: for instance, in the example discussed, *any* set of 8 consecutive cars must have at least one requiring option 1, not just the first 8. If the variables are considered in numerical order, only those implied constraints discussed in [4] make a difference to the search. (The others are redundant in the O.R. sense.) On the other hand, if the variables were considered in a different order, a different set of implied constraints would be needed. Hence, there is an interaction between the implied constraints chosen, and the variable ordering: they cannot be chosen independently.

In summary, the default variable ordering which considers the slots in the sequence in numerical order, if not optimal, is at least better than any ordering that leaves gaps to be filled later, and is the one which fits the implied constraints. For improvements in performance, we therefore need to consider the value ordering.

7 Value Ordering

A static value ordering heuristic can be implemented simply by ordering the domains of the variables; having selected a variable for assignment, the algorithm will choose the first value in the domain, and only consider an alternative if the first assignment fails. As already mentioned, the default is simply to assign values in the order in which the domains happen to be defined, which in the car sequencing problem is essentially a random ordering of the car classes.

As discussed in section 1, it is often a good strategy to apply the 'succeed-first' principle to value ordering, and to choose a value which will have the least impact on future variables, and so seems most likely to lead ultimately to a solution. In this problem, the succeed-first principle would lead us to choose first the cars requiring the smallest number of options.

The basis of the succeed-first principle for value ordering is that since we need only choose one value for the current variable, it is a good idea to ignore values which look likely to cause difficulties, and go for a more promising value: with luck, we will never need to return to this variable and consider an alternative value for it. However, this argument does not apply in this problem, and in similar types of problem, where any possible solution is a permutation of a fixed set of values (the classes of the N cars) and the only question is which variable gets which value. In these circumstances, choosing the easy-to-assign classes first only postpones the assignment of the difficult classes: it would be better to assign the difficult classes first. Hence, rather than a succeed-first strategy, we should adopt a fail-first strategy.

Parrello *et al.* similarly cautioned against 'cherry picking' the easiest cars first, and advised scheduling the difficult cars as early as possible in the sequence (although the method they used was very different from constraint programming). The question remains of how to assess the difficulty of assigning the different classes of car. Parrello *et al.* suggested assigning a 'difficulty factor' to each option, and then measuring the difficulty of each car by the sum of the difficulty factors of the options it requires. This method has the disadvantage that the difficulty factor does not appear to take into account the number of cars requiring each option; the '1 in 5' option would be judged more difficult than the '1 in 2' option, but if there are very few cars requiring the former and many cars requiring the latter, then the cars requiring the '1 in 2' option may be more difficult to fit into the sequence.

It should be noted that in the alternative formulation discussed in section 4, since the variables represent the cars, the order in which the cars are assigned to slots in the sequence is a variable ordering. From that point of view, using the fail-first principle to order the cars is in line with advice given in section 1. From now on, the ordering of the cars will be discussed in terms of the formulation in which they are the variables.

Two intuitively plausible heuristics have been tried. Both use the utilization of each option, i.e. the proportion of cars that require that option as a percentage of the maximum capacity of the station. The first heuristic chooses first the cars that require the option with highest utilization, breaks ties by choosing cars that require the option with second highest utilization, and so on. The second sorts the cars first according to the *number* of options that they require and uses the option utilizations to break ties. Both heuristics choose cars which require most or all of the options first, and can be viewed as estimating the number of constraints involving each car variable.

A third heuristic has also been developed, which is based on the tightness of the option constraints as well as how many constraints there are. For instance, if we consider the binary constraints between the car variables that would arise in the alternative formulation from the '1 out of 2' and the '1 out of 5' options, it is clear that the '1 out of 5' constraint would be tighter than the '1 out of 2' constraint. Since the tightness of the constraints that a variable is involved in, as well as their number, clearly has an influence on the difficulty of assigning a value to it, this should also be taken into account.

The theory on which this heuristic was based was developed from an investigation of variable ordering heuristics, described in [5]. A number of heuristics which take into account the tightness of the constraints that a variable is involved in were proposed; in the particular case when all variables have the same domain size (which is true in this case, if we are looking for an initial ordering of the cars), several of these heuristics simplify to considering variables in ascending order of $\prod_{c \in C_i}(1 - p_c)$ where C_i is the set of constraints that involves variable i, and p_c is the tightness of constraint c, the tightness being the proportion of variable tuples not allowed by the constraint. The value of this measure depends on both the tightness of the constraints (the tighter the constraints, the smaller each term in the product) and the number of constraints (the more constraints a variable is involved in, the more terms there are in the product); the number of constraints on a particular car depends on both the options required and their utilization. The constraint tightness can be estimated for the ternary and binary constraints resulting from the 5 options used in [4], corresponding to the '2 out of n' and '1 out of n' options, respectively, and the number of each type of constraint can be calculated; from these, it is possible to find an ordering of the cars based on this heuristic.

8 Results

The three value ordering heuristics just described were implemented by sorting the cars appropriately before presenting them to the ILOG Solver program. They have been applied to several sets of randomly-generated problems (all using the same set of options as in [4]). The problems were produced by two problem generators, one developed at the University of Leeds, and one at the University of Essex, where GENET [2], a system based on a neural network approach, has been applied to the car sequencing problem.

Problems with up to 200 cars and average option utilization up to 90% have been tried. In most cases, all three heuristics can produce a solution in a few seconds, with very little backtracking. Occasionally, one or more of the heuristics fails to solve the problem within a few minutes. None of the three heuristics dominates the others: on average, the first heuristic (considering just the option utilizations) is worse than the second (considering the number of options as well), but on occasions it does better. As an example of their overall performance, they were applied to a set of 20 problems, each with 200 cars and average option utilization between 80 and 90%. In every case but one, at least one heuristic produced a solution in fewer than 15 backtracks, taking just a few seconds. The one problem which was not solved by any of the three was subsequently shown to have no solution: the utilization of the '1 out of 2' and '1 out of 3' options was extremely high (97% and 100% respectively) and there were not enough cars requiring both options to allow a feasible sequence to be built.

It should be stressed that for problems of this size, unless the utilization of the options is very low, so that the problems are very easy to solve, it is extremely unlikely that a random ordering of the cars will result in a solution to the problem in any reasonable time.

These results using the three heuristics are remarkably good, especially considering that they produce a initial ordering of the cars which is then fixed. During search, the characteristics of the subproblem consisting of the future variables and their remaining domains will change, and the cars which were originally most difficult to assign may no longer be. A dynamic ordering heuristic, which could reassess the problem at each iteration and choose the next car to place in the sequence accordingly, might do still better.

9 Related Work

A number of techniques other than constraint programming have been applied to the car sequencing problem described here, or variants of it. The original paper by Parrello, Kabat & Wos [9] used automated reasoning, incorporating inference rules and a special-purpose solution strategy.

David and Chew [1, 3] describe a number of problems arising from production planning for Renault; they describe a more complex type of car sequencing problem, involving approximately 750 cars per day, in which some of the constraints conflict. They found that simulated annealing gave good results and compare this approach favourably with constraint satisfaction.

Warwick and Tsang [14] used a genetic algorithm to solve car sequencing problems, including unsolvable problems (where the utilization of at least one option is greater than the capacity of the option): in this case the requirement is to find as good a solution as possible. This situation can be modelled as a partial constraint satisfaction problem, in which the capacity constraint violations are minimized and the cars requiring a particular option are spaced as evenly as possible.

A neural network approach is described by Smith, Palaniswami and Krishnamoorthy [11], and compared with more traditional methods (a local improvement steepest-descent heuristic and simulated annealing). They found that a stochastic Hopfield network gave performance comparable with simulated annealing, and better performance for large problems.

Hindi and Ploszajski [7] describe a one-pass heuristic method for the car sequencing problem. The heuristic repeatedly chooses a car to go into the next position in the sequence. The ideal spacing of the different options, given the option capacities and utilisations, is determined, and the car which minimises the deviation from the ideal spacing, without violating any of the constraints, is chosen. If none of the remaining cars can fit into this position, it is left vacant; it is claimed that this is realistic, since it tends to happen only towards the end of the sequence, when the schedule is more likely to be modified before being executed. This is similar to the method described here, except for the treatment of failures. In both cases, when the utilization is high, the car placed next in the sequence will tend to be one requiring most options, amongst those which can be placed next.

10 Discussion and Conclusions

Although the work described in the last section suggests that solving the car sequencing problem as a constraint satisfaction problem is not necessarily the best approach, constraint programming has successfully solved problems of reasonable size very quickly. Some lessons with application to other constraint satisfaction problems can be drawn from this case study.

For many problems, there may be valid alternative CSP formulations with the roles of variables and values reversed. Although one formulation may be clearly preferable to the other, it should be remembered that the conventional wisdom of fail-first for variable ordering, succeed-first for value ordering cannot apply in both. A succeed-first value ordering in one formulation is incompatible with a fail-first variable ordering in the other.

Keng and Yun [8] suggest that if the variables can be thought of as tasks and their values as resources, the most constrained task should be assigned first, i.e. the fail-first principle should apply. In the car sequencing problem, the cars should be thought of as the tasks and the slots as resources, since it is more appropriate to think of the cars competing for the slots in the sequence rather than the other way round. This leads to the fail-first value ordering of the cars, as discussed above.

It is also suggested in [8] that the resource whose assignment will have least impact on future assignments should be chosen next; however, it is difficult to see how this would apply this case. Since the capacity constraints (both original and implied) concern sets of consecutive slots in the sequence, the slots should be considered consecutively, as shown earlier. Furthermore, the specific implied constraints included in the formulation expect that the slots will be assigned in numerical order. As a general principle, the variable ordering must be compatible with the problem constraints, and their interaction should be considered carefully.

The case study of the car sequencing problem has shown that general principles apply to the choice of variable and value ordering heuristics, even when the conventional wisdom, that fail-first is a good strategy for variable ordering, has been overturned. The third value ordering heuristic discussed in section 7, which was based on calculating both the tightness of the constraints and their number, suggests that it may be be possible eventually to derive good variable and value ordering heuristics theoretically, rather than relying on trial and error combined with intuition, as at present.

Acknowledgement

I should like to thank James Borrett and Andrew Davenport of the University of Essex for the car sequencing problems produced by their random problem generator.

References

[1] T.-L. Chew, J.-M. David, A. Nguyen, and Y. Tourbier. Solving Constraint Satisfaction Problems with Simulated Annealing: The Car Sequencing Problem Revisited. In *Proceedings 12th Avignon Conference*, pages 405–416, 1992.

[2] A. Davenport, E. Tsang, K. Zhu, and C. J. Wang. GENET: A Connectionist Architecture for Solving Constraint Satisfaction Problems by Iterative Improvement. In *Proceedings AAAI'94*, pages 325–330, 1994.

[3] J.-M. David and T.-L. Chew. Constraint-based applications in production planning: examples from the automotive industry. In *Proceedings of PACT'95: Practical Applications of Constraint Technology*, pages 37–51, Apr. 1995.

[4] M. Dincbas, H. Simonis, and P. van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In *Proceedings ECAI-88*, pages 290–295, 1988.

[5] I. Gent, E. MacIntyre, P. Prosser, B. Smith, and T. Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Proceedings of CP'96*, pages 179–193, Aug. 1996.

[6] R. Haralick and G. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[7] K. S. Hindi and G. Ploszajski. Formulation and Solution of a Sequencing Problem in Car Manufacture. *Computers & Industrial Engineering*, 26:203–211, 1994.

[8] N. Keng and D. Y. Yun. A Planning/Scheduling Methodology for the Constrained Resource Problem. In *Proceedings IJCAI'89*, pages 998–1003, 1989.

[9] B. D. Parrello, W. C. Kabat, and L. Wos. Job-shop scheduling using automated reasoning: A case study of the car-sequencing problem. *Journal of Automated reasoning*, 2:1–42, 1986.

[10] J.-F. Puget. A C++ Implementation of CLP. In *Proceedings of SPICIS94 (Singapore International Conference on Intelligent Systems)*, 1994.

[11] K. Smith, M. Palaniswami, and M. Krishnamoorthy. Traditional Heuristic versus Hopfield Neural Network Approaches to a Car Sequencing Problem. *To appear in European J. of O.R.*, 1996.

[12] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[13] P. van Hentenryck, H. Simonis, and M. Dincbas. Constraint Satisfaction using Constraint Logic Programming. *Artificial Intelligence*, 58:113–159, 1992.

[14] T. Warwick and E. P. K. Tsang. Tackling car sequencing problems using a generic genetic algorithm. *Evolutionary Computation*, 3:267–298, 1995.