Solving the Car-Sequencing Problem as a Non-binary CSP

Student name: Mihaela Butaru
Supervisor name: Zineb Habbas

Université de Metz, Laboratoire d'Informatique Théorique et Appliquée,
UFR M.I.M., Ile du Saulcy, F-57045 Metz Cedex 1, France
`butaru@univ-metz.fr`, `zineb.habbas@univ-metz.fr`

Abstract. A search algorithm based on non-binary forward checking (nFC) is used to solve the car-sequencing problem. The choice of value ordering heuristics having a dramatic effect on solution time for this problem, different ordering heuristics were implementented. The results obtained by using these methods are compared on the instances reported in the CSPLib.

1 Introduction

The *car-sequencing* problem arises from the manufacture of cars on an assembly line (based on [1]). 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. Among the methods proposed in literature to solve this problem, the exact methods such as the linear programming [2] and the solving methods for constraint satisfaction problems (CSPs) [3], [4], [5] represent good alternatives for certain instances of the problem. Many scheduling and similar problems can be expressed as 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 links 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 [6] and ILOG Solver [7] use a search algorithm based on Forward Checking (FC) [8] to solve CSPs. FC is one of the most common *look-ahead* algorithms (that is, they check for inconsistencies that involve future variables as well as the current and past variables). This algorithm involves two choices at each iteration: the next variable to assign, and the value to assign to it. 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.

In this article, we undertake an experimental study for the instances of the car-sequencing problem in CSPLib[1], encoded as a n-ary CSP using an implementation with constraints of fixed arity 5. By applying value ordering heuristics based on fail-first principle, a great number of these instances can be solved in little time.

2 The Car-Sequencing Problem Encoded as a CSP

The problem described by Parrello *et al.* [1] was subsequently considered by Dincbas *et al.* [9] who showed that it could be formulated as a constraint satisfaction problem, using CHIP. In their basic formulation the assembly line has 5 possible options. The car-sequencing problem can be encoded as a CSP (see [3]) in which slots in the sequence are variables, cars to be built are their values. Following [9], the first step is to group the cars into classes, such that the cars in each class all require the same option. A matrix of binary elements of size the number of classes multiplied by the number of options specifies which are the options present in each class. Given the specifications (in terms of options required) we have to arrange the cars to produce into a sequence such that none of the capacity constraint is violated. These *capacity constraints* are formalized using constraints of the form q_i/p_i, which indicate that the unit is able to produce at most q_i cars with the option i out of each sequence of p_i cars (this should be read q_i *outof* p_i). The constraints already stated are sufficient to express the problem; it seems that the only important thing about the options capacities is not to exceed them, and going *below* the 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 the capacity in one part of the sequence could make it impossible to avoid exceeding the capacity elsewhere. In [9], the authors suggest adding implied constraints in order to allow failures to be detected earlier than would otherwise be possible.

3 Variable and Value Ordering Heuristics

A search algorithm based on forward checking involves two choices at each iteration, concerning the next variable to assign, and the value to assign to it. The

Table 1. Value ordering heuristics

Formula	Maximize		Minimize		Static/Dynamic
	Heuristic	Type	Heuristic	Type	
$\sum_{i \in Opt(cls)} Util_i$	$MaxUtil$	fail-first	$MinUtil$	succeed-first	D
Req_{cls}	$MaxCars$	succeed-first	$MinCars$	fail-first	D
$nbOpt_{cls}$	$MaxOpt$	fail-first	$MinOpt$	succeed-first	S
$\sum_{i \in Opt(cls)} 1/c_i$	$MaxPQ$	fail-first	-	-	S

ordering of variables and values was studied by Smith [3]. More specifically, the effects of the *fail-first* and the *succeed-first* were tested for the car-sequencing

[1] `http://4c.ucc.ie/~tw/csplib/`

problem. The fail-first principle consists in choosing a variable or a value which has the greatest pruning effect on the domains of the future variables, while the succeed-first priciple consists in choosing the variable or the 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 [3] the author suggests that for the car-sequencing problem the variables should be assigned consecutively; since any possible solution is a permutation of a fixed set of values and the only question is which variable gets which value, a succeed-first strategy for value ordering only postpones the assignment of the difficult classes. Table 1 presents the formulas of computation for the different value ordering heuristics. By maximizing or minimizing these formulas we obtain seven heuristics, four of them are based on fail-first priciple and three of them on succeed-first principle. In these formulas the following notations are used: $nbCars$ represents the number of cars to be produced; for each options i, $c_i = q_i/p_i$ is the capacity constraint, Req_i establishes the request and $Util_i = Req_i/(nbCars * c_i)$ represents the *utilization*; for each class cls, Req_{cls} establishes the request, $nbOpt_{cls}$ is the number of options it requires and $Opt(cls)$ is a function which returns these options.

4 Experimental Results

4.1 Implementation and Experimental Framework

In our implementation, we generate the car-sequencing problem as a non-binary CSP with n variables (the slots in the sequence), d values (the cars to be built) and $m = n - 4$ constraints of fixed arity 5 are posted on any 5 consecutive variables. The relations corresponding to the constraints are explicitly built as

Algorithm 1: Procedure for the implied constraints

```
Procedure CheckImplied: Boolean
begin
    $i \leftarrow 1$
    $OK \leftarrow$ TRUE
    while $OK \wedge i \leq 5$ do
        $possible_i \leftarrow nbPos * \frac{q_i}{p_i}$
        if $possible_i < Req_i$ then $OK \leftarrow$ FALSE
        $i \leftarrow i + 1$
    return $OK$
end
```

allowed combinations of values, *i.e.* valid tuples. These tuples are generated respecting the capacity constraints for the options and the total production for each car. In our research project [10] we implemented five versions of n-ary FC algorithms, namely nFC0, nFC2, nFC3, nFC4, nFC5. These versions differ between them in the extent of look-ahead they perform after each variable assignment [5]. On the data file generated in the fashion described above, we can apply all the n-ary algorithms of FC developed, not only some algorithms specific to car-sequencing problem. Of course, we take into account the presence of the implied constraints in the problem, *i.e.* if the domain of an unassigned variable becomes empty or one of the implied constraint is violated (the procedure

CheckImplied in Algorithm 1 returns *false*), 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. In the Algorithm 1 *nbPos* represents the number of available positions in the sequence; if the remaining production Req_i of cars requiring a certain option i exceeds the possible production $possible_i$, an inconsistency is detected. The heuristics introduced in the Section 3 are evaluated on two groups of instances of car-sequencing problem in the CSPLib. The first group includes 70 instances of 200 cars gathered by the utilization percentage varying between 60% to 90%; all these instances concern satisfiable problems because it was proven that there was a sequence not violating any capacity constraint. The second group contains 9 instances of 100 cars for which it was shown that 4 of them were satisfiable, 4 other were infeasible and only one remains with no solution known. For solving each of these instances, the execution time was restricted at 900 seconds. The n-ary algorithms have been developed in C++ using a Unix CC compiler and executed on a SGI3800 machine of 768 R1400 processors 500 MHz. We noticed that nFC0 was much slower than the others and the winner was nFC2, that is why we present below the results corresponding to the last one.

4.2 Results for the First Group of Instances

The number of instances of the first group solved by using the different value ordering heuristics is presented at the Figure 1. We notice that the use of the

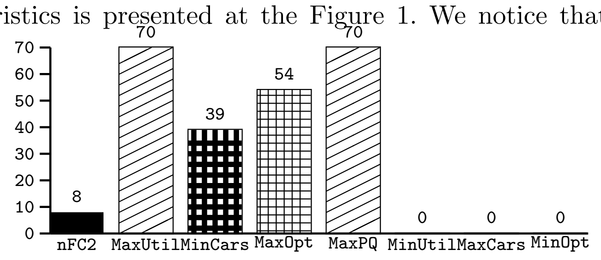

Fig. 1. Number of successful for the first group

algorithm nFC2 allowed solving only 8 of the 70 instances. The use of the value ordering heuristics makes possible to increase the number of solved problems. Indeed, the heuristics of fail-first type, *MaxUtil*, *MinCars*, *MaxOpt* and *MaxPQ*, solved respectively 70, 39, 54 and 70 instances of problems. Regarding the heuristics of succeed-first type, *MinUtil*, *MaxCars* and *MinOpt*, they did not succeed in solving any instance of problem. We can thus conclude, as Smith [3] had underlined it, that the value ordering heuristics based on fail-first principle perform better than those of succeed-first type. In particular, the *MaxUtil* and *MaxPQ* heuristics obtain interesting results for these problems, even if *MaxUtil* is a dynamic heuristic. Table 2 presents the detailed results by using the heuristics *MaxUtil* and *MaxPQ* for solving the instances of 200 variables gathered by the utilization percentage of the options and we count 10 instances for each of them. The results in table correspond to the average results

Table 2. Results of *MaxUtil* and *MaxPQ* heuristics for the first group.

MaxUtil								*MaxPQ*							
Pb.	T_{max}	D	t_D	#nodes	#ccks	#BT	#OK	Pb.	T_{max}	D	t_D	#nodes	#ccks	#BT	#OK
60	3.15	200	3.15	200	525705	0	10	60	3.37	200	3.37	300	527670	100	10
65	3.97	200	3.97	200	805918	0	10	65	4.44	200	4.44	302	806229	102	10
70	4.70	200	4.70	200	1029707	0	10	70	4.91	200	4.91	268	1021358	68	10
75	5.65	200	5.65	200	1326867	0	10	75	5.75	200	5.75	257	1281343	57	10
80	6.82	200	6.82	200	1682611	0	10	80	6.91	200	6.91	239	1705391	39	10
85	8.35	200	8.35	200	2292852	0	10	85	8.24	200	8.24	226	2205163	26	10
90	10.73	200	10.73	200	3078951	0	10	90	10.93	200	10.93	240	3034675	40	10
Avg:	6.20	200	6.20	200	1534659	0	Total=70	Avg:	6.36	200	6.36	261	1511647	61	Total=70

obtained for each utilization percentage. Let us note that the column T_{max} indicates either the necessary time to solve the problem or, in the case of a unsolved problem, the maximum time spent to seek a solution. The column D shows the number of positions in the sequence which it was possible to affect during the execution (*i.e.* the maximum depth in the search tree), t_D represents the necessary time to reach this depth in the search tree, #nodes, #ccks and #BT count respectively the number of visited nodes, constraint checks and backtracking to reach the maximum depth in the search tree. Finally, #OK indicates the number of problems solved by the algorithm. The last row of the table gives an average for each column as well as the total number of successful instances. A comparison between these two heuristics enables us to conclude that *MaxUtil* remains the best heuristic. Moreover, it is surprisingly backtrack-free.

4.3 Results for the Second Group of Instances

For the second group of problems, including very difficult instances, none of them is solved without heuristic in 900 seconds. However, using *MaxUtil* the problems

Table 3. Results of *MaxUtil* and *MaxPQ* heuristics for the second group.

MaxUtil								*MaxPQ*							
Pb.	T_{max}	D	t_D	#nodes	#ccks	#BT	yes/no	Pb.	T_{max}	D	t_D	#nodes	#ccks	#BT	yes/no
4_72	900	90	1	152	732450	62	no	4_72	900	91	320	41124	48467390	41033	no
6_76	900	70	2	398	652343	328	no	6_76	900	58	0	74	434567	16	no
10_93	900	75	6	406	2565764	331	no	10_93	900	76	34	2662	98008555	2586	no
16_81	155.085	100	155.085	35966	15391983	35866	yes	16_81	900	95	12	1664	2077529	1570	no
19_71	900	91	29	6559	3577762	6468	no	19_71	900	90	71	9162	12538120	9072	no
21_90	900	91	3	874	641208	783	no	21_90	900	88	7	1532	1133881	1444	no
36_92	900	70	2	387	682380	317	no	36_92	900	75	23	3653	3922874	3578	no
41_66	0.903	100	0.903	101	310180	1	yes	41_66	1.225	100	1.225	179	355985	79	yes
26_82	900	95	20	6412	2284875	6317	no	26_82	900	85	764	108937	110949640	108852	no
Avg:	717.33	87	24.33	5632	2982105	5545	2	Avg:	800.13	84	147.88	18775	21076505	18691	1

16_81 and 41_66 (known as satisfiables) are solved, while the *MaxPQ* heuristic solves the problem 41_66 (see Table 3, where the column "yes/no" indicates if the problem was solved). Let us note that the instance 41_66 was solved in less than one second with *MaxUtil* and 1.22 seconds with *MaxPQ*. For this category of problems the approach developed in [11] did not solve any instance.

5 Conclusions and Future Work

In this article, a comparison of the performances for different value ordering heuristics making possible to guide the search for solutions for car-sequencing problem was presented. The problem was encoded as a non-binary CSP and the filtering method is based on n-ary Forward Checking. The results obtained showed the superiority of a strategy of fail-first type against to a succeed-first strategy. Moreover, the $MaxUtil$ and $MaxPQ$ heuristics allowed a better exploration of the space of solutions and solved all the instances of problems with 200 variables, whereas the same heuristics in [11] solved 12 respectively 51 instances among the 70. It should be underlined the fact that these problems were solved in little time (6 seconds on average) and the longest time is 13 seconds for the instance 90_09, whereas for ILOG Solver the least powerful time exceeds 1 minute. This result can be justified by our encoding. Indeed, we encoded the maximum of constraints (the capacity of each option, the request for each class) inside an explicit 5-ary constraint with very high tightness (close to 0.95). $MaxUtil$ remains the best heuristic because it is surprisingly backtrack-free. Within the future work, the filtering method will be improved in order to solve a greater number of instances of problems and, in particular, those with 100 variables. A hybridization of the optimization methods represents another way of interesting research. The use of parallelism also seems an interesting direction for solving this type of problem and the works in this direction are in progress.

References

1. Parrello, B.D., Kabat, W.C., Wos, L.: Job-shop schedulind using automated reasoning: a case study of the car-sequencing problem. Journal of Automated Reasoning **2** (1986) 1–42
2. Gravel, M., Gagné, C., Price, W.L.: Review and comparison of three methods for the solution of the car sequencing problem. Journal of the Operational Research Society (2004)
3. Smith, B.M.: Succeed-first or fail-first: A case study in variable and value ordering. Report 96.26, University of Leeds (1996)
4. Régin, J.C., Puget, J.F.: A filtering algorithm for global sequencing constraints. Constraint Programming (1997) 32–46
5. Bessière, C., Meseguer, P., Freuder, C., Larossa, J.: On forward checking for non binary constraint satisfaction. Artificial Intelligence **141** (2002) 205–224
6. van Hentenryck, P.: Constraint Satisfaction in Logic Programming. MIT Press, Cambridge (1989)
7. Puget, J.F.: A c++ implementation of clp. In: Proceedings of SPICIS94 (Singapore International Conference on Intelligent Systems). (1994)
8. Haralick, R.M., Elliot, G.L.: Increasing the search efficiency for constraint satisfaction problems. Artificial Intelligence **14** (1980) 263–313
9. Dincbas, M., Simonis, H., van Hentenryck, P.: Solving the car-sequencing problem in constraint logic programming. In: Proceedings ECAI-88. (1988) 290–295
10. Butaru, M., Habbas, Z.: Problèmes de satisfaction de contraintes n-aire: une étude expérimentale. In: Actes des Premières Journées Francophones de Programmation par Contraintes (JFPC05), Lens, France (8-10 Juin, 2005)
11. Boivin, S., Gravel, M., Krajecki, M., Gagné, C.: Résolution du problème de car-sequencing à l'aide d'une approche de type fc. In: Actes des Premières Journées Francophones de Programmation par Contraintes (JFPC05), Lens, France (8-10 Juin, 2005)