The Car-Sequencing Problem as n-Ary CSP – Sequential and Parallel Solving

Mihaela Butaru and Zineb Habbas

LITA, Université de Metz,
UFR M.I.M., Ile du Saulcy, F-57045 Metz Cedex 1, France
{butaru, zineb.habbas}@univ-metz.fr

Abstract. 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 (designed to handle at most a certain percentage of the cars passing along the assembly line) which install the various options. 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. The solving methods for constraint satisfaction problems (CSPs) [2], [3], [4] represent good alternatives for certain instances of the problem. Constraint programming tools [5], [6] use a search algorithm based on Forward Checking (FC) [7] to solve CSPs, with different variable or value ordering heuristics. In this article, we undertake an experimental study for the instances of the car-sequencing problem in CSPLib, encoded as an 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. Moreover, the parallel solving using a shared memory model based on OpenMP makes it possible to increase the number of solved problems.

Keywords: Constraint satisfaction, heuristics, problem solving, scheduling.

1 The Car-Sequencing Problem Encoded as a CSP

The car-sequencing problem can be encoded as a CSP (see [2]) in which slots in the sequence are variables, cars to be built are their values. Following [8], 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 the present options in each class. We have to arrange the cars to produce into a sequence such that none of the *capacity constraint* is violated. These constraints are formalized q_i/p_i (i.e. 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: 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 [8], the authors suggest adding *implied constraints* in order to allow failures to be detected earlier than it would otherwise be possible.

2 Value Ordering Heuristics for Car-Sequencing Problem

The ordering of variables and values was studied by Smith [2]. More specifically, the effects of the *fail-first* and the *succeed-first* were tested for the car-sequencing 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 principle consists in choosing the variable or the value that is likely to lead to a solution, and so reduces the risk of having to backtrack to this variable and try an alternative value. In [2] the author suggests that for the car-sequencing problem the variables should be assigned consecutively. In [9] we describe seven value ordering heuristics for this problem: $MaxUtil$, $MinCars$, $MaxOpt$, $MaxPQ$ based on fail-first priciple and $MinUtil$, $MaxCars$, $MinOpt$ based on succeed-first one.

3 Implementation Framework and Experimental Results

In our implementation, we generate the car-sequencing problem as an n-ary 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 valid tuples, generated respecting the capacity constraints for the options and the total production for each car. We implemented [10] five versions of n-ary FC algorithms (i.e. nFC0, nFC2, nFC3, nFC4, nFC5) which differ between them in the extent of look-ahead they perform after each variable assignment [4]. Due to our implementation, we can apply any of the developed algorithms, not only some algorithms specific to car-sequencing problem. Of course, we take into account the presence of the implied constraints in the problem. We present here the results corresponding to nFC2 algorithm, noticed as the best one. The heuristics in Section 2 are evaluated on two groups of instances of car-sequencing problem in the CSPLib[1]: the first group includes 70 instances of 200 cars, the second one contains 9 instances of 100 cars.

We also present in this paper our first results (see the full paper [9]) of parallel solving for car-sequencing problem, using the search tree distribution approach within a shared memory model (see [11] for details) based on OpenMP.

Our solver has been developed in C++ using a Unix CC compiler and executed on a SGI3800 machine of 768 R1400 processors 500 MHz. In the tables below, T_{max} is either the necessary time to solve the problem or, in the case of

[1] `http://4c.ucc.ie/~tw/csplib/prob/prob001/data.txt`

a unsolved problem, the maximum time spent to seek a solution (restricted to 900 seconds); D is the number of positions in the sequence which it was possible to affect; t_D is the necessary time to reach this depth; *#nodes*, *#ccks* and *#BT* counts respectively the number of visited nodes, constraint checks and backtracking to reach D; *#OK* is the number of solved problems solved; *Y/N* indicates if the problem was solved; T_g is the time CPU corresponding to the *guided* tasks allocation in the parallel execution (within the OpenMP environment, there is a *static*, *dynamic* or *guided* tasks allocation [12]; we present here only the last one, which performs better, even if with small differences, than the two others).

Tables 1, 2 give the results for the first group, while Tables 3, 4 give the results for the second group.

Table 1. Sequential results of *MaxUtil* and *MaxPQ* 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	70	Avg:	6.36	200	6.36	261	1511647	61	70

Table 2. Parallel results of $nFC2$, *MinCars* and *MaxOpt* for the first group

$nFC2$					*MinCars*					*MaxOpt*				
Pb.	Serial		Parallel		Pb.	Serial		Parallel		Pb.	Serial		Parallel	
	T_{max}	#OK	T_g	#OK		T_{max}	#OK	T_g	#OK		T_{max}	#OK	T_g	#OK
60	721.84	2	524.1	7	60	451.43	5	119.86	9	60	349.15	8	97.87	9
65	722.98	2	401.27	6	65	272.5	7	110.10	9	65	276.74	7	19.87	10
70	734.36	2	423.16	6	70	284.55	7	12.36	10	70	364.67	6	102.43	9
75	816.75	1	590.43	5	75	183.98	8	5.04	10	75	214.84	8	21.39	10
80	900	0	773.27	3	80	116.69	9	22.63	10	80	301.17	7	92.09	10
85	900	0	724.63	3	85	810.72	1	364.51	6	85	186.4	8	98.72	9
90	811.2	1	772.03	2	90	723.25	2	297.15	7	90	23.11	10	9.06	10
Avg: 800		8	Avg: 601	32	Avg: 406		39	Avg: 133	61	Avg: 245		54	Avg: 63	67

Table 3. Serial results of *MaxUtil* and *MaxPQ* for the second group

MaxUtil								*MaxPQ*							
Pb.	T_{max}	D	t_D	#nodes	#ccks	#BT	Y/N	Pb.	T_{max}	D	t_D	#nodes	#ccks	#BT	Y/N
4_72	900	90	1	152	732450	62	N	4_72	900	91	320	41124	48467390	41033	N
6_76	900	70	2	398	652343	328	N	6_76	900	58	0	74	434567	16	N
10_93	900	75	6	406	2565764	331	N	10_93	900	76	34	2662	98008555	2586	N
16_81	155	100	155	35966	15391983	35866	Y	16_81	900	95	12	1664	2077529	1570	N
19_71	900	91	29	6559	3577762	6468	N	19_71	900	90	71	9162	12538120	9072	N
21_90	900	91	3	874	641208	783	N	21_90	900	88	7	1532	1133881	1444	N
36_92	900	70	2	387	682380	317	N	36_92	900	75	23	3653	3922874	3578	N
41_66	0.903	100	0.903	101	310180	1	Y	41_66	1.22	100	1.225	179	355985	79	Y
26_82	900	95	20	6412	2284875	6317	N	26_82	900	85	764	108937	110949640	108852	N
Avg:	717	87	24	5632	2982105	5545	2	Avg:	800	84	148	18775	21076505	18691	1

The results obtained showed the superiority of the fail-first strategy against a succeed-first one. Moreover, *MaxUtil* and *MaxPQ* solved all the instances of 200 variables. The same heuristics in [13] solved 12 respectively 51 instances. These problems were solved in little time (6 seconds on average), which can be justified by our encoding. The longest time (13 seconds) was spent for the

Table 4. Parallel results of *MaxUtil*, *MaxOpt* and *MaxPQ* for the second group

MaxUtil					*MaxOpt*					*MaxPQ*				
Pb.	Serial		Parallel		Pb.	Serial		Parallel		Pb.	Serial		Parallel	
	T_{max}	Y/N	T_g	Y/N		T_{max}	Y/N	T_g	Y/N		T_{max}	Y/N	T_g	Y/N
4_72	900	N	2.009	Y	4_72	900	N	4.253	Y	4_72	900	N	2.324	Y
16_81	155.085	Y	65.241	Y	16_81	900	N	900	N	16_81	900	N	900	N
41_66	0.903	Y	0.9	Y	41_66	900	N	900	N	41_66	1.255	Y	1.2	Y
26_82	900	N	862.24	Y	26_82	900	N	900	N	26_82	900	N	900	N

instance 90_09, whereas with ILOG Solver the least powerful time exceeds 1 minute. For the second group, *MaxUtil* solves the problems 16_81 and 41_66, while *MaxPQ* solves the problem 41_66. For this group, [13] did not solve any instance. The parallel execution using a shared memory model based on OpenMP increased the number of solved problems for the first group, and all the problems known as satisfiable in the second one using *MaxUtil*, which remains the best heuristic because it is surprisingly backtrack-free.

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. Smith, B.M.: Succeed-first or fail-first: A case study in variable and value ordering. Report 96.26, University of Leeds (1996)
3. Régin, J.C., Puget, J.F.: A filtering algorithm for global sequencing constraints. Constraint Programming (1997) 32–46
4. Bessière, C., Meseguer, P., Freuder, C., Larossa, J.: On forward checking for non binary constraint satisfaction. Artificial Intelligence **141** (2002) 205–224
5. van Hentenryck, P.: Constraint Satisfaction in Logic Programming. MIT Press, Cambridge (1989)
6. Puget, J.F.: A c++ implementation of clp. In: Proceedings of SPICIS94 (Singapore International Conference on Intelligent Systems). (1994)
7. Haralick, R.M., Elliot, G.L.: Increasing the search efficiency for constraint satisfaction problems. Artificial Intelligence **14** (1980) 263–313
8. Dincbas, M., Simonis, H., van Hentenryck, P.: Solving the car-sequencing problem in constraint logic programming. In: Proceedings ECAI-88. (1988) 290–295
9. Butaru, M., Habbas, Z.: Sequential and parallel solving for the car-sequencing problem. Rapport interne 2005–101, Université de Metz, Laboratoire d'Informatique Théorique et Appliquée (2005)
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. Butaru, M., Habbas, Z.: Parallel solving with n-ary forward checking: A shared memory implementation. In: Proceedings of the First International Workshop on OpenMP (IWOMP05), Eugene, Oregon, USA (June 1-4, 2005) to appear.
12. OpenMP Architecture Review Board: OpenMP Application Program Interface. (2005) http://www.openmp.org.
13. 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)