

3.3 Crossover Operator

The GA crossover operator explores the structural search space by creating offspring strings from selected parent strings. A crossover operator needs to encourage exploration, yet not destroy the important information already contained in the population. The crossover operator should allow the offspring to inherit building blocks from the parents. GAcSP uses a uniform adaptive crossover (*UAX*) (Warwick & Tsang, 1993), which has an extended string representation. It is designed to exploit PCSP constraints by enabling links between string values to be inherited. The *UAX* is suitable for PCSPs because in a PCSP, the variables have no inherent ordering, but the value for each variable is highly dependent on the values for a set of other variables (due to the constraints). Represented in a string, the variables are given a particular order. In a standard crossover, short schemas are less likely to be destroyed. By using the *UAX*, we hope that break off points which reflect the dependency relations of the variables can be discovered and passed on to future generations.

The extra binary string acts as a template to control the creation of the offspring string during the crossover process. Successful strings will have the opportunity to become parents and pass their crossover points on to offspring. The first stage of the crossover operator weighted randomly selects two parents from the matepool. Parent strings are cut after each matching crossover point (to be determined by the parent templates) and alternating sections are used to create an offspring. At start, an offspring inherits the values from a randomly selected parent. This continues until the templates of both parents share the same value. When this happens, the offspring inherits values from the other parent, until the next common value is shared by the two parent templates. The following example should make this process clear:

	string position:	1	2	3	4	5	6	7	8	9	10
parent 1	string solution:	1	2	3	4	2	5	4	1	3	5
	template:	0	1	0	0	1	1	0	0	1	0
parent 2	string solution	2	1	3	2	4	1	5	3	4	5
	template	1	0	1	0	0	1	1	0	0	1

The offspring generated from parents 1 and 2 is:

	from parent:	1	1	1	2	2	1	1	2	2	2
offspring	string solution:	1	2	3	4	2	5	4	1	3	5
	template:	0	1	0	0	1	1	0	0	1	0

This offspring first inherited values from parent 1. At position 4, both parent templates share the value 0. Therefore, the offspring started to inherit values from parent 2. Two more switches were made at positions 6 and 8. The offspring replaces the lowest fitness member of the population.

3.4 Repair and Hill Climbing Operators

One effect of the crossover operator upon the representation is that offspring created will not always satisfy the CarSP constraints (i.e. production requirements). We ensure each offspring satisfies the production requirements by using a greedy repair function. This function is necessarily application dependent.

For the CarSP, we defined a greedy repair function which works in the following way: it first searches in the string for values which are over-represented ($> pr[j]$) and values which are under-represented ($< pr[j]$). Then an arbitrary set of string positions which take the over-represented values are selected, and their values replaced by under-represented values. This ensures that the string represents a schedule which satisfies the production requirements, which is a hard constraint.

After repair, each offspring is hill-climbed by a string element swap function for a pre-set time period. In each iteration of the hill climbing, an arbitrary pair of string positions are picked. If the swapping of values between these two positions result in a fitter string, then the swap will be accepted, and hill climbing continues from the new string. The same strategy is later used with success in the connectionist approach Genet (Davenport & Tsang 1995).

4 Empirical Results

4.1 Experiments overview

In our experiments we are concerned with GAcSP's ability to cope with CarSPs with both loose and tight constraints (Sections 4.2 and 4.4), various sizes (Section 4.3) and over-constrained (hence unsolvable) problems (Section 4.4). All solvable CarSPs were generated by a program supplied by Kangmin Zhu which provided a solution to each problem satisfying the capacity constraints (Zhu,

1993). All CarSPs tested have 5 options with capacity constraints; 1:2, 2:3, 1:3, 2:5 and 1:5. This range of capacity constraints allows us to test GAcSP performance and directly compare our results with those of other researchers.

Recently, Chow *et. al.* (1992) applied simulated annealing to the car sequencing problem; but since it uses a different formulation of the problem their results are not directly comparable with ours. Other work which apply GAs to CSPs include Eiben *et. al.* (1994) and Filipic (1992). GSAT or its extensions (Selman *et. al.* 1992, 1993, 1994) have not been included in our tests because adapting it to the CarSP is a non-trivial task.

All algorithms were written in C and tests were run on SUN 4/110 work-stations under the UNIX 4.0 operating system. The following parameters have been used for GAcSP throughout all the tests: (a) population size is 80, which was found to be effective in an earlier work (Tsang & Warwick, 1990); (b) 10% of the fittest members (elite) of the population were copied directly into the mating pool at reproduction phase of GAcSP; (c) The number of offspring created in each cycle was arbitrarily set at four; (d) the termination conditions are 400 cycles or 10 CPU hours; and (e) a maximum of 30 CPU seconds is allowed for hill climbing for each offspring.

It should be emphasized here that algorithms comparison is difficult in general. Run time can be seriously affected by the ways that algorithms are implemented. Besides, our comparison in experiment 4.2 is limited by the capacity of the algorithms that we compare GAcSP to.

It may be worth emphasizing that tabu search is in fact a class of algorithms. The instantiation of the tabu list plays a crucial part in its effectiveness and efficiency. The instantiation that we used in the comparison below is the most successful one that was developed in (Zhu, 1993).

4.2 GAcSP, HR and Tabu Tackling CarSPs of Different Tightness

In *Experiment 4.2* GAcSP, *Heuristic Repair* (HR) (Minton *et. al.* 1990) and a version of *Tabu search* (Tabu) (Glover, 1989; Glover, 1990) were tested on solvable 100 car CarSPs with average utilities μ ranging from .45, .50, ..., .90. The HR strategy assigns a random value to each variable, and then repeat the following steps: pick a variable whose current value violates some constraints, and re-assign to it a value which minimizes the number of constraints violated (which could result in

assigning the same value to it). This process terminates when no constraint is violated or resources (e.g. a maximum number of iterations) have run out. Tabu search is a local search strategy which uses a tabu list to restrict the moves for transforming one solution (state) into another. Both of these algorithms were adapted to tackle solvable and unsolvable CarSPs. The tabu search used in our experiments is identical to HR, except that the value that has been replaced in the preceding iteration will not be used in the current iteration.

The pseudo code for HR and the version of tabu search used in our experiments are presented in Appendix A. We compare GAcSP against HR and Tabu because (a) both can be extended to tackle solvable and unsolvable CarSP's; (b) like GAcSP, they can handle optimization problems (most search techniques in constraint satisfaction were developed for satisfiability problems only); and (c) they are well known algorithms in constraint satisfaction research (which motivated this work).

The iteration limit for HR and Tabu was set to 100,000 adjustment cycles. It was found (empirically) that allowing more iterations (say 1,000,000) did not improve the quality of the best results (Zhu, 1993). For each of the 10 average utilities tested, we randomly generated 10 solvable CarSPs, and 10 runs were carried out on each problem. Therefore, there were a total of 100 runs for each utility test. (HR and Tabu results were supplied by Dr. Zhu.) The experimental results are summarized in Table 3.

Table 3: Summary GAcSP, HR and TABU Experiment 4.2 Results

avg utility μ	.45	.50	.55	.60	.65	.70	.75	.80	.85	.90
avg car types k	8.7	12.3	12.9	16	18.2	20	21.2	22	23.4	23.3
GAcSP - number solns	100	100	99	100	99	99	92	61	21	1
GAcSP - avg violation	0.00	0.00	0.01	0.00	0.01	0.01	0.09	0.50	1.40	3.90
GAcSP - avg run-time sec	29	49	69	43	60	212	457	1122	2104	4421
HR - number solns	98	97	94	96	94	97	88	58	15	1
HR - avg violation	0.02	0.04	0.07	0.08	0.07	0.04	0.19	1.04	2.42	7.00
HR - avg run-time sec	19	26	46	40	57	44	144	451	856	975
TABU - number solns	100	100	100	100	100	100	97	21	0	0
TABU - avg violation	0.00	0.00	0.00	0.00	0.00	0.00	0.05	1.62	5.74	11.85
TABU - avg run-time sec	4	6	11	4	8	10	111	818	956	960

The following keys are used in tables 3 to 5:

avg car types k	The average number of car types for each average utility
number solns	Number of runs returning solutions (out of 100), i.e. where $S_{\text{cost}} = 0$
avg violation	The mean of minimum violations in the 100 runs (including solutions)
avg run-time sec	The mean of run-times (in CPU seconds)

For each algorithm, the number of times that it returns a solution, the number of constraints violated in the best (partial) solution during each run, as well as the run time are measured. The statistically significant difference between the number of solutions found by GAcSP and HR (see Table 4 and Figure 2) demonstrates that GAcSP out-performed HR in finding solutions to Experiment 4.2 CarSPs. GAcSP has found solutions in all runs for the .45, .50, and .60 average utility tests and found 99% for .55, .65 and .70 average utility tests. GAcSP average performance for finding solutions to .45 to .70 tests is 99.5%. Tabu has found solutions to all runs in the .45 to .70 utility tests. Across all utilities, GAcSP found solutions in 77.2% of its runs. HR and Tabu found solutions in 73.8% and 71.8%, respectively, of their runs.

Table 4: Summary of F-tests on results in Experiment 4.2

	Observed F value			Critical F values	
	number solns (36.63)	avg violation 2.73	avg run-time 2.99	$\alpha = 0.01$ 10.6	$\alpha = 0.05$ 5.12
Conclusion: there is a 99% level of confidence to reject the hypothesis that there is no difference between the performance of GAcSP and HR in finding solutions					

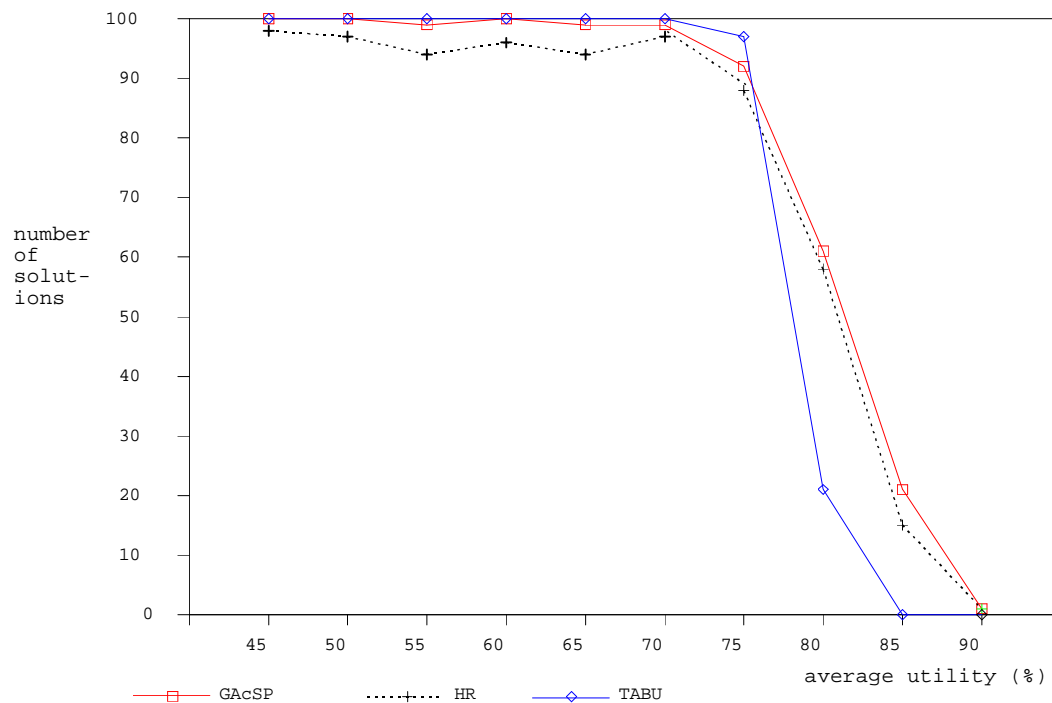


Figure 2: Experiment 4.2, GAcSP, HR and Tabu on 100 cars CarSPs

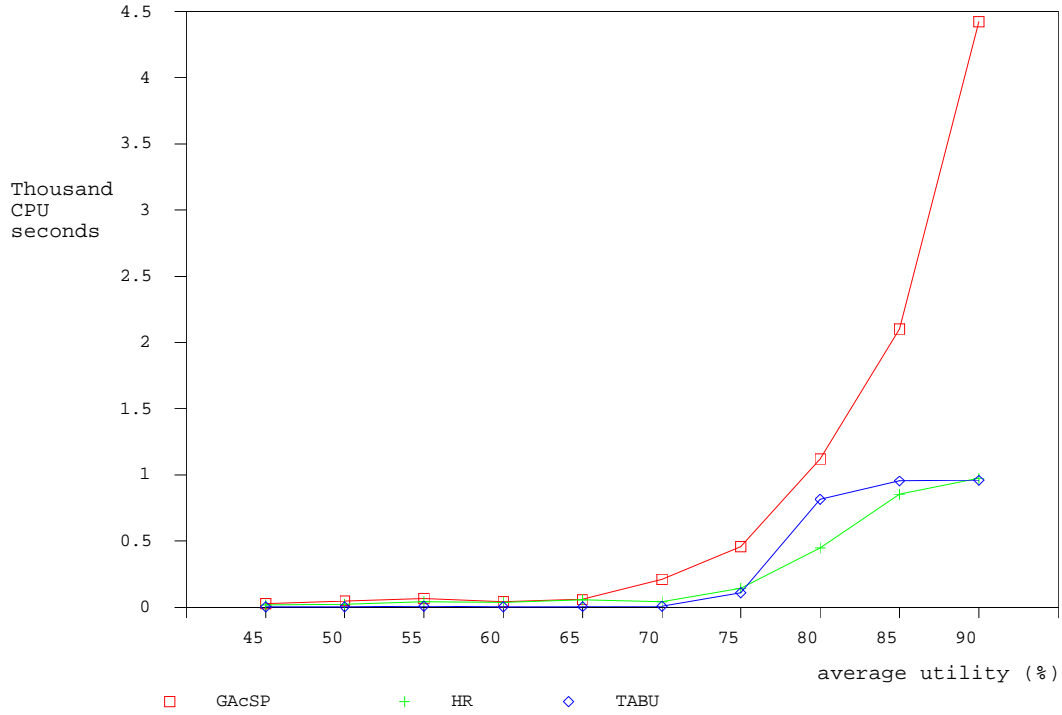


Figure 3: Average CPU time for CarSPs in Experiment 4.2

GAcSP does better in finding solutions and in minimizing the number of violations than HR and Tabu when the algorithms are put under pressure by the increasing complexity of CarSPs. Both GAcSP and Tabu are able to find solutions in nearly all .45 to .75 average utility runs within reasonably quick run-times (see Figure 3). However, it is only at the .80 to .90 utility results that we can clearly distinguish between the behaviors of the algorithms tested. At .80 all algorithms suffer a severe reduction in solution finding ability, with Tabu failing to find any solutions at .85 and .90. Both HR and GAcSP find more solutions than Tabu from .80 to .90, with GAcSP finding more solutions than HR.

The dramatic reduction in performance of the algorithms on the .80 average utility test is due to the interactions between the options, which make CarSPs more difficult to solve. This option interaction is due to a combination of the number of options in the car types and the capacity constraints. GAcSP, HR and Tabu depend upon local information to explore the search space. With increasing option interactions local information is less effective in guiding the search towards solutions and

consequently the starting points used become more important. Local information becomes less helpful when a car in a schedule can violate more than one option, which creates two difficulties for local search techniques: (1) there are fewer alternative positions to move cars to in order to reduce their option violations; and (2) there are more cars with option violations.

Both HR and Tabu will suffer from their dependence on the quality of good starting points. Tabu is expected to perform better than HR because it can escape from local minima, due to a limited memory for previous choices. With GAcSP, increasing CarSP utility provides reduced feedback about the fitness space used to guide both the GA and HC components. But information from a population of search points reduces the chance of GAcSP being trapped in local minima. Although HC will contribute less directly to the search process it will still assist in the development of good building blocks. In which case, the work of the GA component is increased. This balance of work shared between GA and HC is an important feature of GAcSP -- it improves its robustness. Furthermore, performance of GAcSP could be improved by controlling this balance by fine tuning GAcSP parameters. On the other hand, HR and Tabu mainly depend upon local information, and therefore, may be harder to improve their performance.

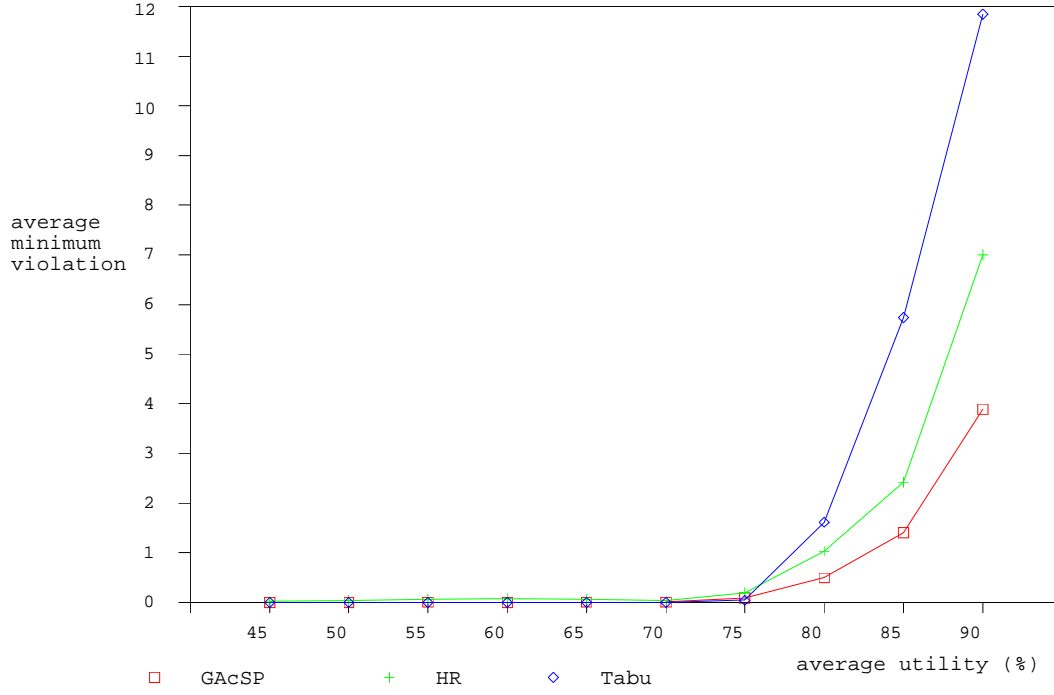


Figure 4: Average minimum violations in Experiment 4.2 results

In the .45 to .75 utility tests the average minimum violation results are dominated by the number of solutions found by the algorithms. But as the algorithms find less solutions in the .80 to .90 tests, the significance of average minimal violation as a measure of performance is increased. Although the number of solutions returned by GAcSP, HR and Tabu for increasing utility tests above .75 decline significantly, reduction in the average minimum violation is not as severe. Figure 4 shows that the rate of increase in GAcSP's average minimum violation results for .80 to .90 is not as great as those in HR and Tabu. GAcSP can remain consistently near optimal even at .90. (In Experiment 4.4 below, we demonstrate that GAcSP can retain this near optimal performance on $\mu > .90$.) The mean of average minimal violation of GAcSP, HR and Tabu on Experiment 4.2 CarSPs are 0.592, 1.097 and 1.926, respectively.

4.3 GAcSP Tackling CarSPs of Different Sizes

In Experiment 4.3 GAcSP was tested on solvable CarSPs with 100, 120, ..., 200 cars, and utilities .50, .60, .70, and .80. There were 5 randomly generated CarSPs for each utility and 5 runs carried out on each problem. Results of Experiment 4.3 are summarized in Table 5.

Table 5: GAcSP performance on CarSPs of different sizes

	.05 average utility						.06 average utility					
number cars N	100	120	140	160	180	200	100	120	140	160	180	200
avg car types k	12.6	11.2	12.2	11.4	17.8	15.2	16.4	17.4	18.2	19.6	22.6	22.2
number solns	25	25	25	23	25	24	25	25	25	25	25	19
avg violation	0	0	0	.08	0	.04	0	0	0	0	0	0.4
avg run-time <i>sec</i>	20	92	218	1228	96	421	117	214	331	916	159	3070
	.07 average utility						.08 average utility					
number cars N	100	120	140	160	180	200	100	120	140	160	180	200
avg car types k	20.6	21.4	22.8	21.6	25	25	22.6	22.4	24.2	24.2	25.8	262
number solns	25	25	25	23	24	23	7	17	14	15	10	9
avg violation	0	0	0	.08	0.04	.12	.92	.32	.92	1.36	.88	1.2
avg run-time <i>sec</i>	339	369	699	1704	539	4177	2033	3422	5261	7079	5969	7289

Note that a different set of 100-cars problem were generated, hence discrepancy between the results in this table and those in table 3.

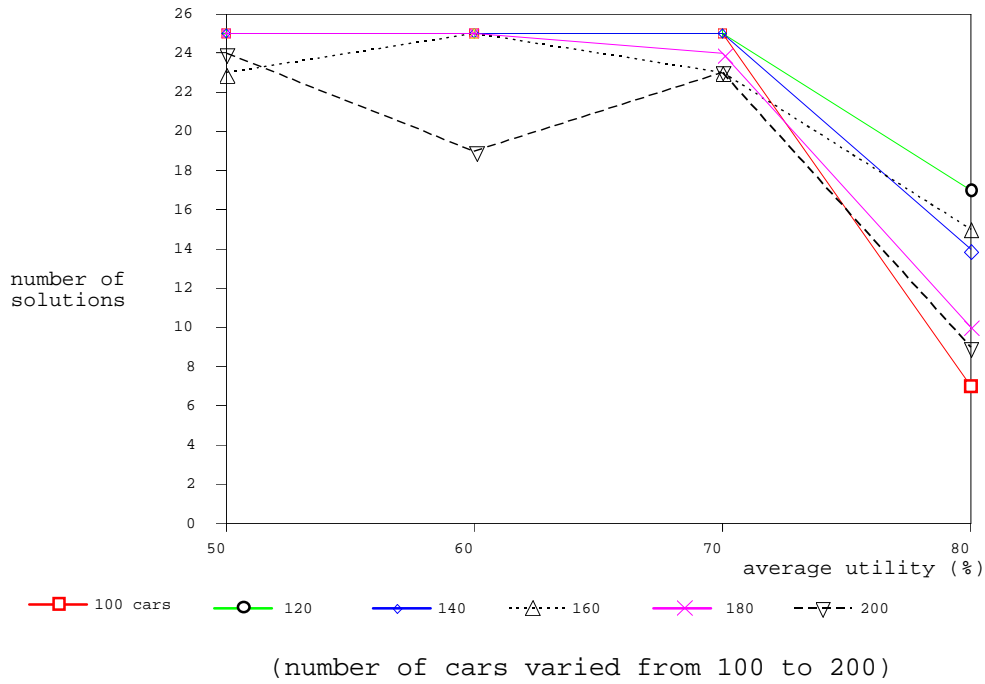


Figure 5: Experiment 4.3, GAcSP on CarSPs of different sizes

The ability of GAcSP to return solutions decreases as the utility is increased, supporting our observations from Experiment 4.2 (see Figure 5). Although there is a slight reduction in the number of solutions with the increase in the number of cars, generally the ability of GAcSP is consistent. The

loss in performance is not significant (see Table 6), yet the increase in size of the search space for these CarSPs is significant (see Table 7). Table 6 shows that there is no correlation between the number of cars and the run time. This shows that the combinatorial explosion problem can be contained by GAcSP.

Table 6: Summary Experiment 4.3 result F-test statistics - significance in parentheses

hypothesis:	Observed F-value				Criterion F-value	
	number solns	avg violation	avg cost	avg run-time	$\alpha = 0.01$	$\alpha = 0.05$
Number of cars: N	1.87	(5.72)	2.21	2.79	4.56	2.90
One significant statistical conclusion from Table 6 is: there is a 99% level of confidence to reject the hypothesis that there is no correlation between the number of cars and the average violation achieved by GAcSP for Experiment 4.3 CarSPs.						

Table 7: Search space sizes for Experiment 4.3 CarSPs

N^k	100^k	120^k	140^k	160^k	180^k	200^k
avg utility $\hat{u} = .50$	1.6E+25	1.9E+25	1.5E+26	1.3E+25	1.4E+40	9.5E+34
avg utility $\hat{u} = .60$	6.3E+32	1.5E+36	1.1E+39	1.6E+43	9.3E+50	1.2E+51
avg utility $\hat{u} = .70$	1.6E+41	3.1E+44	8.5E+48	4.1E+47	2.4E+56	3.4E+57
avg utility $\hat{u} = .80$	1.6E+45	3.7E+46	8.6E+51	2.2E+53	1.5E+58	1.9E+60

Since the HC time limit is held constant for all CarSPs tested, the extra work undertaken by GAcSP must be due to the GA component. This work sharing GAcSP behavior is an important design feature and suggests that the time allowed to HC depends more on problem characteristics of the number of car type options and production requirements (as we have seen with Experiment 4.2 tests) than problem size. This emphasizes the fact that the GA component of GAcSP ensures robustness whilst the HC component adds a specialist ability.

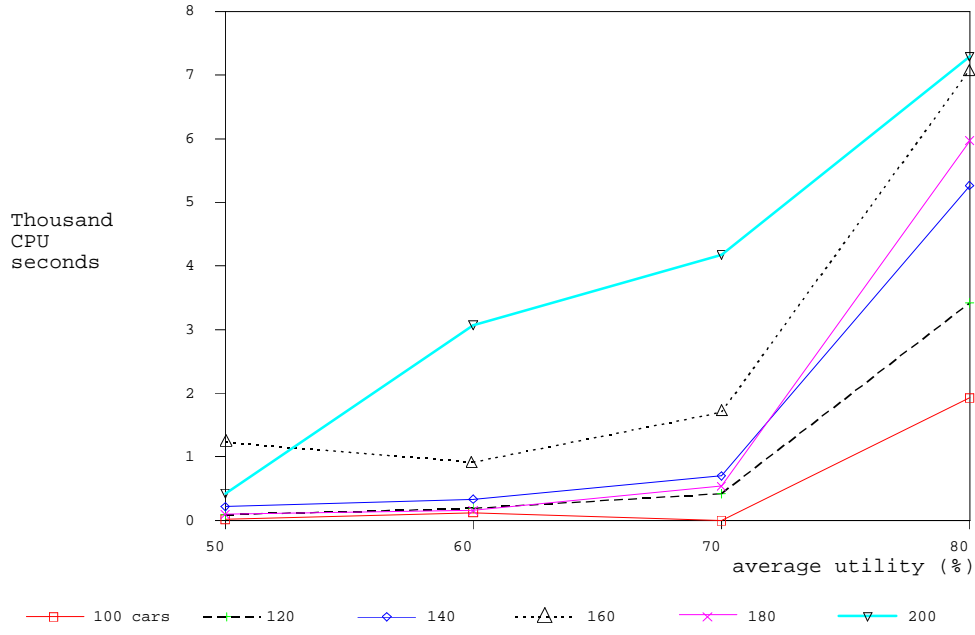


Figure 6: Average run-time for GAcSP in Experiment 4.3

In general, the ability of GAcSP in finding solutions is not necessarily restricted by search space size. However, an important effect of increasing the CarSP size is to increase the computational workload of the GA, which can slow GAcSP down. This increase in the case of GAcSP is due mainly to the CPU requirements of the evaluation function and crossover mechanism. The average CPU run-time in Figure 6 shows this increase for all run-time averages shown in Table 5 with the exception of the .50 180 car CarSP. In this case, all the test runs resulted in solutions, enabling the GAcSP to terminate before complete convergence.

We can make a limited comparison with the results from Experiment 4.2 and 4.3, with those reported by Parrello *et. al.* (1986, 1988): using an Automated Reasoning Program (ITP) and OPS5 to sequence 5 cars with 5 options took 35 minutes and 15 minutes, respectively. Dincbas *et. al.* (1988) tackled solvable CarSPs with CHIP, a *constraint logic programming* system. They reported that CHIP could sequence 100 car schedules with an average utilization of .80 in under 60 seconds and 200 cars between 336 and 345 second, but only one problem was used for each schedule. Besides, they have only tackled solvable CarSPs.