

4.4 GAcSP Tackling Unsolvable CarSPs

In Experiment 4.4, GAcSP was tested on unsolvable CarSPs. Each unsolvable CarSP was generated by making a single option over-utilized (as described above). This allows us to calculate the lower bounds of the optimal costs (using equation 9).

We carried out tests on 4 groups of problems: unsolvable CarSPs were generated from solvable CarSPs with average utilities of .50, .60, .70, and .80. A total of 5 unsolvable CarSPs were generated for each group in the following way. From a CarSP in each of these groups, we produced 5 new unsolvable CarSPs by over-utilizing each of the 5 options. For each option m , where $m = 1, 2, \dots, 5$, the solvable CarSP has option m added to randomly selected car types until $u_m > 1$. Five runs were made for each unsolvable CarSP. Therefore, there are a total of 25 runs for each group.

By over-utilizing a single option in creating each unsolvable CarSP we have increased the average utility significantly. For example, a number of the new average utilities are greater than .90 and in one particular case 1.024. (On average, group .50 average utilities increased by 40%; .60 by 26%; .70 by 17%; and .80 by 10%.) Yet in general, the minimum and average violation solutions are close to the theoretical lower bound (see Figure 7). We can assume that the optimal minimal violation solutions for each group of tests is within the range of these two values. GAcSP results from Experiment 4.4 have been summarized in Table 8 and Figure 7.

Table 8: Frequency, number of violations above theoretical lower bound

(25 runs were made in each group; percentages in bracket)										
number violations	0	1	2	3	5	6	7	8	13	15
group .50 (%)	6 (24)	8 (32)	1 (4)	3 (12)		4 (16)		1 (4)	1 (4)	1 (4)
group .60 (%)	8 (32)	10 (40)	5 (20)		1 (4)		1 (4)			
group .70 (%)	7 (28)	9 (36)	5 (20)	4 (16)						
group .80 (%)	4 (16)	7 (28)	5 (20)	7 (28)	1 (4)		1 (4)			

Table 8 summarizes the following results for each group of problems: (1) the number of solutions obtained that are n constraint violations above the lower bound ($n = 0, \dots, 15$); and (2) the percentage of solutions in each category. The same GAcSP performance statistics were summarized for

Experiment 4.4 as for Experiment 4.3. The theoretical lower bounds were calculated using Equation 9. We can see from Table 8 that an average 25% of the theoretical optimal solutions (solutions whose costs are at the theoretical lower bound) are found, with a further average of 64% within 3 violations (see Figure 8).

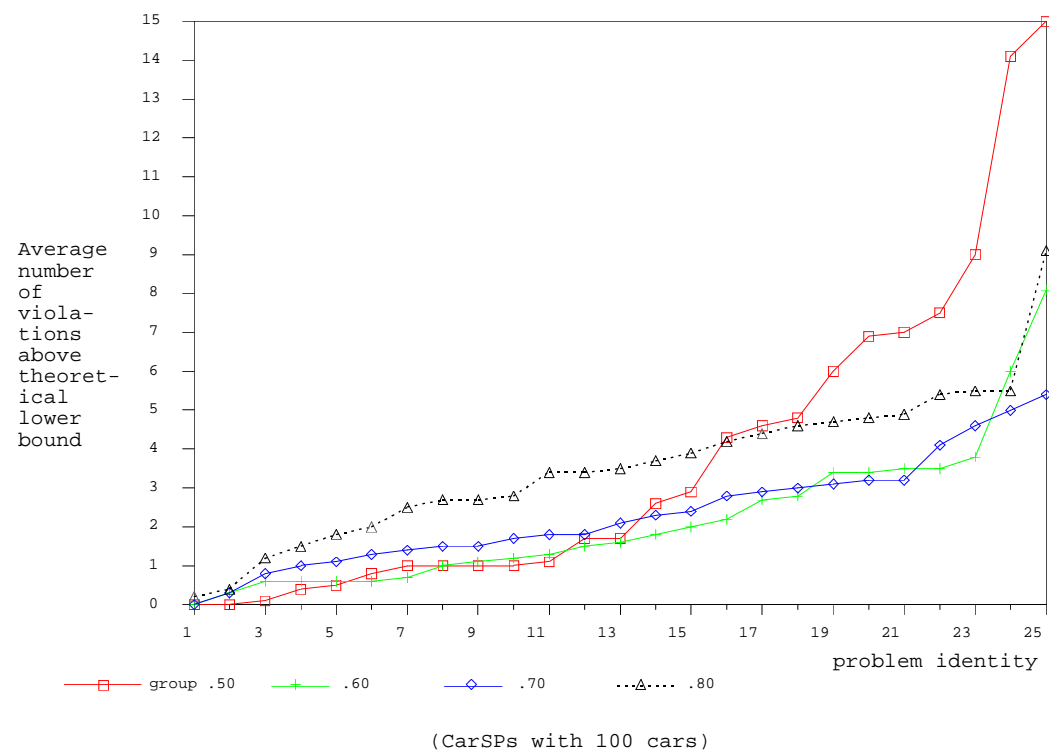


Figure 7: Average number of violations for Experiment 4.4 CarSPs

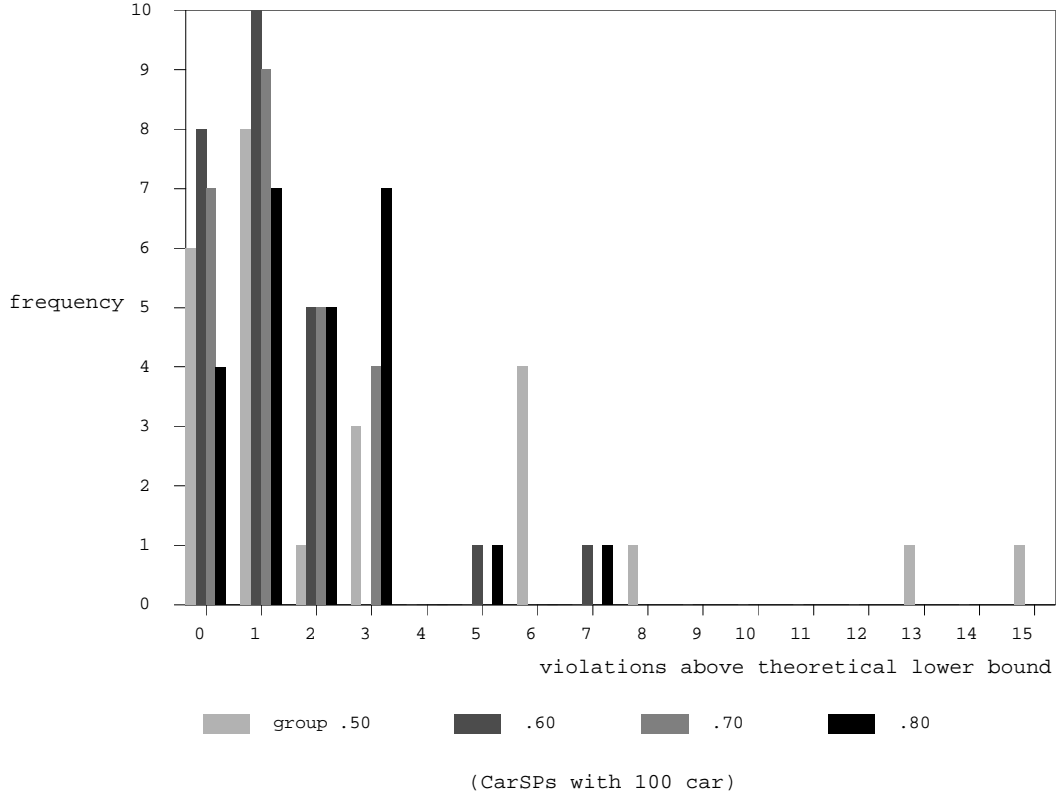


Figure 8: Summary of number of violations over lower bound

GAcSP can exploit tight utility ratio constraints to sustain the search in tackling unsolvable CarSPs, through the action of the crossover operator. In Figure 9 we present the mean number of cycles (y-axis) to convergence (to a state in which all strings have the same fitness) for each utility ratio run (a few runs were terminated at the maximum 400 cycles). The x-axis in Figure 9 represents decreasing capacity constraint tightness, measured as the number of non-option spaces allowed in a schedule by the capacity constraint:

$$capacity\ constraint\ tightness = 1 - \frac{p_m}{q_m} \quad (10)$$

where $p_m:q_m$ is the capacity constraint for option m .

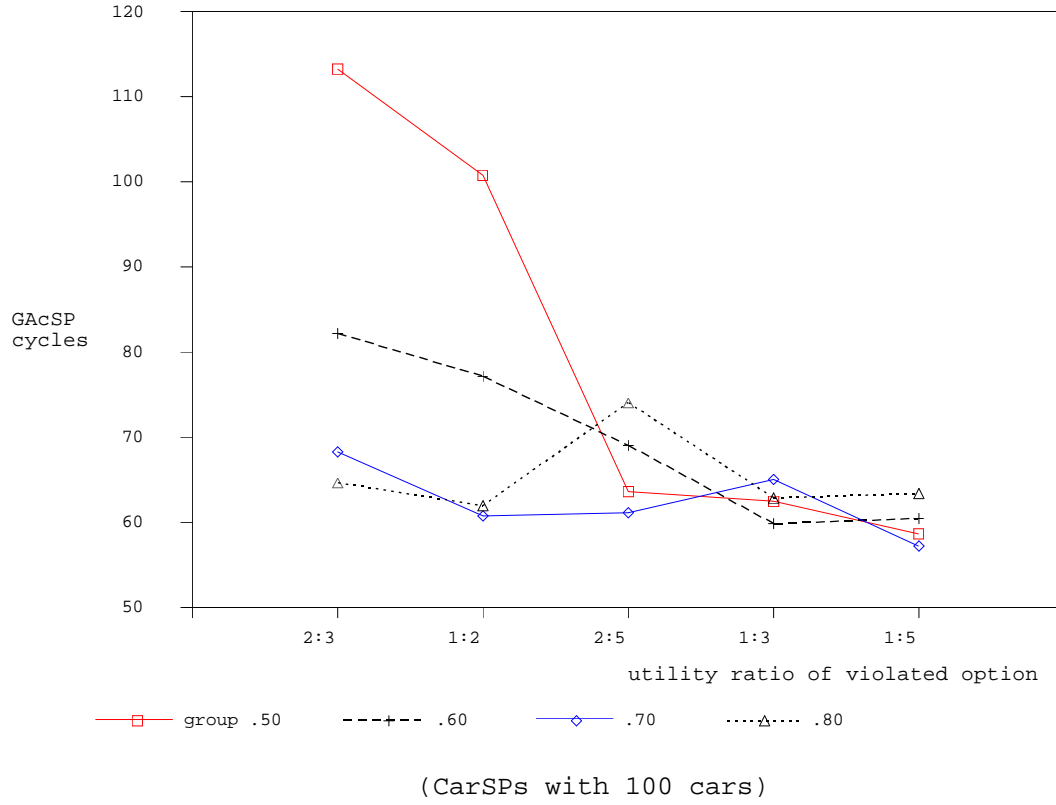


Figure 9: Number of cycles by GAcSP in Experiment 4.4

In general, the number of cycles for each test utility decreases as the capacity constraint tightness decreases, demonstrating a positive correlation between utility ratio tightness and GAcSP cycles. The curves for groups .50, .60, and .70 demonstrate this correlation, but not so strongly with .80 (because the average utility tightness for .80 unsolvable CarSPs has an effect on the results). The average utility tightness reflects soft constraint interaction and influences GAcSP through the objective function. In the less tight utility ratio tests, GAcSP was unable to sustain the search as long. However, if we consider Figure 10 which summarizes the closeness to the lower bound that was achieved by each over-utilized category of problems, we find that the quality of results between tight and loose utility ratio tests is slightly improved. GAcSP was able to exploit the hard position dependent constraints in sustaining the search with increasing utility ratio tightness.

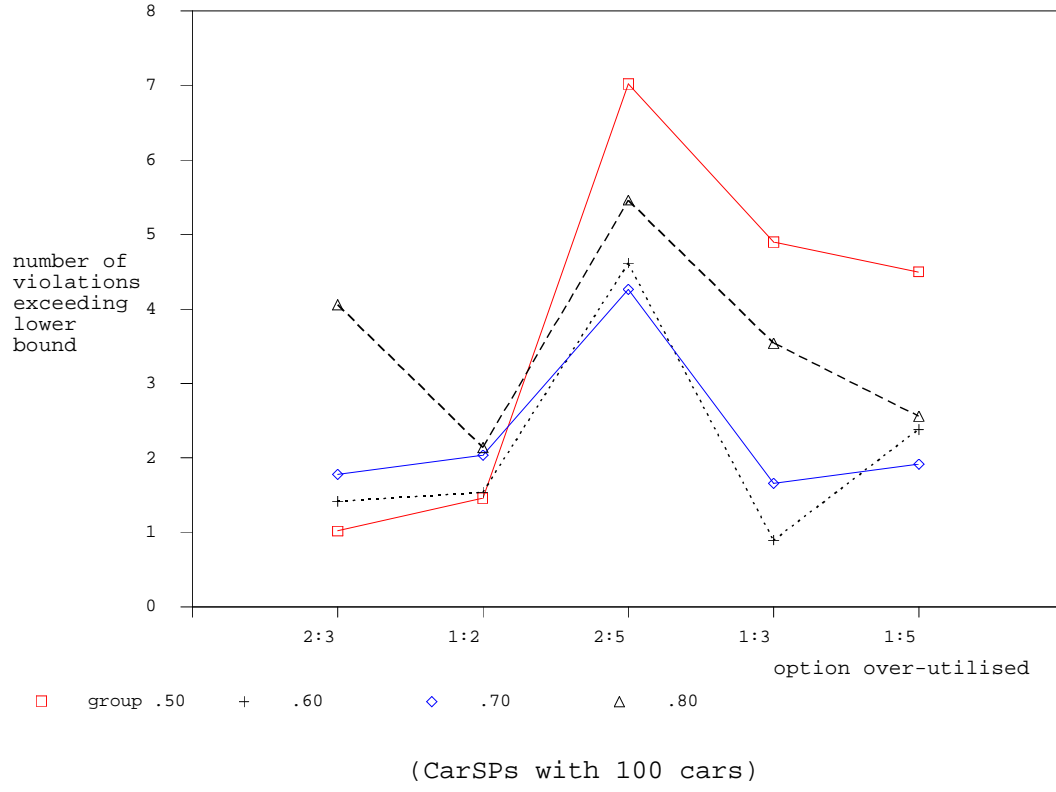


Figure 10: Experiment 4.4 results compared to lower bound

In order to sustain the search, the crossover mechanism must use knowledge of constraints in a purposeful way. The crossover operator can use position dependency due to tight constraints to try and form good building blocks. GAcSP can creep towards the optimal solutions by ensuring that tightly positioned options are recorded on the binary templates. With loose constraints (e.g. 2:5), GAcSP is required to select from a combination of alternative car positions. The alternative combinations increase the work required by GAcSP in finding a minimal violation. Furthermore, there may only be one combination of car positions for an option which will achieve a lower bound violation. Therefore, options which have lower capacity constraints allow more alternative arrangements in placing options in a schedule which satisfy the capacity constraint. The HC component can fine tune near optimal solutions to minimize the capacity violation in tightly constrained CarSPs. However, achieving an optimal sequence is more difficult and beyond the localized ability of the HC. Only UAX has the necessary ability to simultaneously sequence a number of cars to achieve this. Although alternatives

require more work from GAcSP to achieve the minimum lower bound, near optimal results could be found.

Run-times shown in Figure 9 are longer for Experiment 4.4 tests of the same size and average utility than those in Experiment 4.2 (from which they were derived) due to the fact that runs were terminated only after complete convergence. The intention was to ensure that the theoretical lower bound could not be improved upon, and to demonstrate typical run-times for unsolvable problems. The price to pay for tackling unsolvable CarSPs is increased computation, resulting in longer run times in comparison with the run times for solvable problems and times achieved by Dincbas *et. al.* (1988). Compromises can be made should one be prepared to sacrifice optimality for speed in unsolvable CarSPs.

5 Conclusion

Partial constraint satisfaction is a general problem. In this paper, we have presented a generic GA named GAcSP for tackling partial constraint satisfaction problems (PCSPs). We have demonstrated its effectiveness in a case study using the car sequencing problem (CarSP). The “engine” of GAcSP is a crossover operator (UAX) which remembers valuable crossover points in order to help retaining useful building blocks which may be separated in the string representation. The UAX attempts to exploit PCSP constraints by using an extended binary string representation, which encodes information about “preferred” cut off points.

The CarSP results show that GAcSP is not restricted to tackling solvable problems only, can be effective in both loosely and tightly constrained problems, is a robust search technique and is not deterred by the problem size (demonstrated in 100 - 200 car CarSPs). GAcSP out-performed both HR and Tabu, techniques which are applicable to both solvable and unsolvable CarSPs.

Through the action of the crossover operator, GAcSP can exploit the constraints to improve on solution quality. The GA component ensures robustness whilst the HC component adds a specialist ability. The balance of work between the GA and HC components can be controlled according to the scale of the problem. As larger problems are tackled, the GA component can undertake more responsibility for the search. With larger search spaces the GA component of GAcSP offers more guidance to locate the areas of hills for the hill-climber to exploit. Unlike other stochastic

optimization techniques for PCSPs, GAcSP is a robust exploration strategy which does not easily get trapped in local minima. Therefore, GAcSP could provide a useful and practical tool for tackling a class of combinatorial problems where current solving techniques are limited or infeasible.

Apart from the CarSP, GAcSP has been tested on the processor configuration problem (PCP) (Warwick & Tsang, 1993). Promising results in these tests support to our claim that GAcSP is a generic PCSP solver, which can achieve optimal or near optimal solutions to both solvable and unsolvable classes of PCSPs.

Acknowledgement

Warwick's Ph.D. research was supported by an SERC grant. The authors are grateful to Dr. Zhu for providing us with the HC and Tabu results, and the constructive and detailed comments by De Jong and the anonymous referees.

Appendix A -- Pseudo Codes

Pseudo codes for GAcSP:

```

PROCEDURE PCSP(P);
/* setup GAcSP parameters and then call GAcSP to solve the PCSP */
BEGIN
    p_size ← population size, e.g. 80;
    n_ospring ← number of offspring created each GAcSP cycle, e.g. 4;
    elite ← number of elite population members to select, e.g. 8;
    hc_time ← maximum time in CPU seconds to hill-climb offspring;
    /* when hc_time = 0, HC is switched off ), e.g. hc_time can be set to 30 */
    terminator ← any termination condition, e.g. maximum cycles 400;
    GAcSP(P, p_size, n_ospring, elite, hc_time, terminator);
END /* PCSP */

PROCEDURE GAcSP((Z, D, C, g), p_size, n_ospring, elite, hc_time, terminator);
/* main procedure to solve the PCSP = (Z, D, C, g) where Z = set of variables, D = function which
maps every variable in Z to a finite domain, C = set of constraints and g is a function to optimize */
BEGIN
    population ← Initialisation(Z, D, g, p_size);
    REPEAT
        matepool ← Reproduction(Z, g, population, p_size, elite);
        matepool ← Crossover(Z, D, C, g, matepool, p_size, n_ospring, hc_time);
        population ← matepool;
        cycle ← cycle + 1;
    UNTIL (terminator);
END /* GAcSP */

```