Solving constraint satisfaction sequencing problems by iterative repair

Andrew J. Davenport[1], Edward P. K. Tsang[2]

[1] *Department of Mechanical and Industrial Engineering,*
University of Toronto, Toronto, Ontario, CANADA
[2] *Department of Computer Science, University of Essex,*
Colcehster, Essex, UNITED KINGDOM

andrewd@ie.utoronto.ca, edward@essex.ac.uk

Abstract

Many constraint satisfaction problems involve sequencing constraints, where the aim is to find a sequence for a domain of values such that all the constraints on the sequence are satisfied. Specialised techniques have been developed to tackle this problem within the constraint programming framework using constructive, backtracking search. In this paper we investigate local search techniques to tackle this problem. By taking advantage of the structure of the sequencing problem we show that within a local search framework we can reduce the size of the search space and the number of constraints which are required to be satisfied. We present SwapGenet, an iterative repair-based algorithm designed specifically for solving the constraint satisfaction sequencing problem. We present results of an empirical evaluation demonstrating the superiority of SwapGenet over Genet on the car sequencing problem.

Introduction

Many constraint satisfaction problems are in fact constraint satisfaction sequencing problems, where the aim is to find a sequence for a domain of values such that all the constraints on the sequence are satisfied. Examples of constraint satisfaction sequencing problems include rostering and the car-sequencing problem. In recent years specialised propagation and search techniques have been developed to deal with such problems within the constraint programming framework (Dincbas, Simonis, & Van Hentenryck, 1988; Smith, 1996; Regin & Puget, 1997). These techniques all use constructive search, powerful propagation techniques and backtracking. In this paper we examine an alternative approach: local search techniques to tackle the constraint satisfaction sequencing problem.

The advantages of developing local search techniques specifically for sequencing problems are twofold. Firstly, by considering a "sequencing neighbourhood", we can reduce the number of possible complete assignments of values to variables. Intuitively it follows that this would result in a smaller search space, since the number of possible complete assignments of values to variables provides an upper bound on the size of the search space. In this paper we prove that the number of possible complete assignments of values to variables in a sequencing formulation of a constraint satisfaction sequencing problem is smaller than in the generic constraint satisfaction problem formulation. Secondly, we show that we can also reduce the number of constraints to be satisfied, since the constraints which were needed to specify that a problem was a sequencing problem in the generic constraint satisfaction representation are now implicitly satisfied in the sequencing representation of the problem.

In this paper we present SwapGenet, an iterative repair-based algorithm designed specifically for solving constraint satisfaction sequencing problems. SwapGenet is derived from Genet, a min-conflicts repair-based algorithm for solving CSPs which has been shown to be very effective at solving hard, binary and general constraint satisfaction problems (Davenport, Tsang, Wang, & Zhu, 1994). We present results of an empirical evaluation demonstrating the superiority of SwapGenet over Genet on hard car-sequencing problems.

Problem definition and analysis

Definition 1 (The constraint satisfaction problem; (Tsang, 1993)) *A "generic" constraint satisfaction problem is a triple (Z, D, C) where:*

- *Z is a finite set of* variables*,*
- *D is a function which maps every variable in Z to a set of objects of arbitrary type. We denote by D_x the set of objects mapped by D from x, where $x \in Z$. We call the set D_x the domain of x and the members of D_x possible* values *of x.*
- *C is a set of* constraints*. Each constraint in C restricts the values that can be assigned to the variables in Z simultaneously.*

In this paper we are concerned with finite constraint satisfaction problems, where the variables have a finite domain of values. A *label*, denoted by $\langle x, v\rangle$, is a variable-value pair which represents the assignment of value v to variable x. A *compound label* is the simultaneous assignment of values to variables. We use $(\langle x_1, v_1\rangle, \ldots, \langle x_n, v_n\rangle)$ to denote the compound label of assigning $v_1, \ldots, v_n$ to $x_1, \ldots, x_n$ respectively. A *solution tuple* of a CSP is a compound label for all the variables in the CSP which satisfies all the constraints in C.

Definition 2 (The constraint satisfaction sequencing problem) *A constraint satisfaction sequencing problem is an extension of the constraint satisfaction problem in the following way. All the variables in Z are partitioned into one or more disjoint subsets Z_i, for each of which there is a* sequence constraint *of the form* sequence(Z_i,D_i)*. This specifies that the values assigned to the variables in Z_i must be a sequence of the values in the bag ${D_i}^1$. Thus each subset of variables Z_i is mapped by the function D to a single bag of objects D_i, where $|Z_i| = |D_i|$. Each object in D_i is to be assigned to exactly one variable in Z_i such that every variable is assigned a value and all the constraints in C are satisfied.*

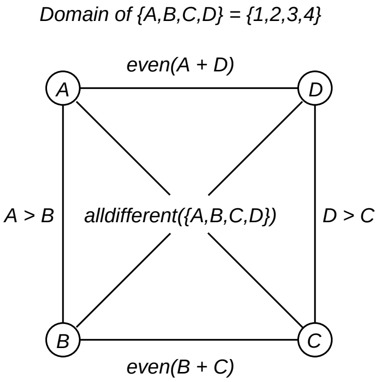

Figure 1: CSP with alldifferent constraint.

Example 1 Consider the CSP given in figure 1. We can represent this CSP using the generic CSP formulation, in which case we have four variables $Z = \{A, B, C, D\}$, each with a domain $D_i = \{1, 2, 3, 4\}$. This would give us $4^4 = 256$ possible complete assigments of values to variables. The alldifferent constraint in this CSP signifies that no two variables may take the same value. This constraint can be represented using binary or general constraints.

However the alldifferent constraint also allows us to regard this CSP as a sequencing problem, since there are four variables, four values for each variable and each variable must take on a different value. In this case we

---

[1] We use the Z formulation of $[\![x_1, \ldots, x_n]\!]$ to denote a bag, or multi-set, of objects $x_1, \ldots, x_n$ (Diller, 1990).

have a single subset of variables $Z_1 = \{A, B, C, D\}$ for which we want to assign a sequence of the values $D_1 = \llbracket 1, 2, 3, 4 \rrbracket$, which we represent by the constraint *sequence(*$\{A, B, C, D\}, \llbracket 1, 2, 3, 4 \rrbracket$ *).* Note that this sequence constraint replaces the alldifferent constraint. The number of possible ways of sequencing the values in D_1 is ${}^4P_4 = 4! = 24$. The alldifferent constraint will be satisfied in every one of these sequences.

Theorem 1 *Given a constraint satisfaction sequencing problem one may represent it using either the generic constraint satisfaction problem formulation or specifically as a sequencing problem. The sequencing representation will always have a smaller number of assignments of values to variables*[2].

Proof 1 Consider a problem with n variables in Z where we have *sequence(Z, D)*. There are three cases to consider:

Case 1 All the values in D are distinct (*e.g.*, the CSP in figure 1).

In this case the representation of this problem using a generic CSP formulation would require n variables of domain size n (since there must be n values in D). Thus we would have n^n possible complete assignments of values to variables. The number of possible sequences would be ${}^nP_n = n!$ which is always less than or equal to n^n.

Case 2 The values in D are not all distinct.

In this case some of the permutations of the values in D are not distinct *e.g.,sequence(*$\{A, B, C\}, \llbracket 1, 1, 2 \rrbracket$*).* If there are d *distinct values* in D then the generic CSP formulation of the problem will have d^n possible complete assignments of values to variables[3]. If each value in d occurs d_i times then the number of possible sequences S in the sequencing formulation is:

$$S = \frac{{}^nP_n}{\prod_{i=1}^{d} {}^{d_i}P_{d_i}} = \frac{n!}{\prod_{i=1}^{d}(d_i!)} \tag{1}$$

S will be largest when $d_d = d_{d\ 1} = \cdots = d_1 = n/d$. Thus to show that the sequencing search space is always smaller than the generic CSP search space we need to show that:

$$d^n > \frac{n!}{[(n/d)!]^d} \tag{2}$$

---

[2] As long as it contains more than one variable.

[3] A sequencing problem formulated as a generic CSP will always have a uniform domain for all its variables.

We can rewrite this as:

$$d^n\left[(n/d)!\right]^d > n! \tag{3}$$

If we write $m = n/d$ this becomes:

$$\begin{aligned}
d^{md}\,(m!)^d &> n! \\
(d^m \times m!)^d &> n! \\
(n \times (n-d) \times (n-2d) \times \cdots \times 2d \times d)^d &> n! \\
n^d \times (n-d)^d \times (n-2d)^d \times \cdots \times (2d)^d \times d^d &> n!
\end{aligned}$$

To show that this is the case consider:

$$\begin{aligned}
n^d &= n \times n \times \cdots \times n \\
&> n \times (n-1) \times (n-2) \cdots \times (n-d+1) \\
(n-d)^d &= (n-d) \times (n-d) \times \cdots \times (n-d) \\
&> (n-d) \times (n-d-1) \times \cdots \times (n-2d+1) \\
&\vdots \\
d^d &= d \times d \times \cdots \times d \\
&> d \times (d-1) \times (d-2) \times \cdots \times 1
\end{aligned}$$

Case 3 More than one set of variables to sequence *e.g.,sequence(*$\{A, B\}$, $[\![1, 2]\!]$*)* and *sequence(*$\{C, D\}$, $[\![1, 2]\!]$*)*. In this case the number of possible sequences is always less than if we were finding a single sequence for all the variables and values *e.g.,sequence(*$\{A, B, C, D\}$, $[\![1, 2]\!]$*)*. Hence this must always be less than the number of possible assignments of values to variables in the generic CSP formulation. □

Overview of Genet

Before presenting SwapGenet we first briefly describe Genet. Genet is a min-conflicts repair-based algorithm (Minton, Johnston, Philips, & Laird, 1992) for solving constraint satisfaction problems (Davenport et al., 1994). The Genet procedure can be implemented in a connectionist architecture, and thus is capable of being fully parallelised, although even on a sequential processor Genet has been shown to be very effective at solving difficult binary and general CSPs.

Genet solves CSPs by hill-climbing, using a variation of the min-conflicts heuristic. The main difference between Genet and min-conflicts hill-climbing

(MCHC) is that GENET has the ability to escape local minima. This it does by using a constraint weighting scheme similiar to that proposed in (Morris, 1993). Each constraint in the CSP has an associated weight which represents the cost of violating that constraint. All constraint weights are positive, and initially these are all set to 1[4]. The cost of a constraint c_k in a GENET network state S is defined as:

$$c_k(S) = \begin{cases} w_k, & \text{if } c_k \text{ is violated in } S \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

where w_k is the weight associated with constraint c_k and the state of a GENET network is simply a complete assignment of values to variables, represented by a compound label. Thus if the constraint c_k is violated then a cost of w_k is incurred, otherwise no cost is incurred. The cost function which GENET minimizes by hill-climbing is given as:

$$g(S) = \sum_{c_k \in C} c_k(S) \tag{5}$$

The value of $g(S)$ will be zero in state S where no constraints are violated, and positive otherwise.

Constraints in GENET may be represented either intensionally or extensionally. This gives GENET the flexibility to represent constraints such as *atmost* or *atleast* (Davenport et al., 1994). These constraints can be infeasible in practice to represent as sets of allowed or disallowed tuples.

Pseudo-code for GENET is presented in algorithm 1. Initially all variables in the CSP are assigned values randomly from their domains. GENET then hill-climbs using a variation of the min-conflicts heuristic. The pseudo-code for GENET hill-climbing is given in algorithm 2. A variable is selected for repair, and is reassigned a value which minimizes the cost function g. In the case that several values minimize this function one of these values is chosen randomly. Sideways moves are allowed, where the value of the cost function g remains unchanged after a repair, even though the value of a variable may change. Note that the value of g will never increase due to a repair since a variable can always be reassigned its current value, thus resulting in a sideways move.

Like many other local searches, GENET will encounter local minima in the search space, where some constraints are violated but no improvements

---

[4] This has been changed, for the purpose of clarity, from the presentation of constraint weights described in (Davenport et al., 1994).

```
procedure GENET(Z, D, C, S)
begin
    i ← 0;
    $S_i$ ← an arbitrary assignment of values to variables;
    repeat
        Hill-Climb($Z, D, C, S_i, S_{i+1}$);
        if $g(S_{i+1}) = g(S_i)$ then
        begin
            for c ← each element of C do
                if c is violated in $S_{i+1}$
                    $w_c \leftarrow w_c + 1$;
        end
        $i \leftarrow i + 1$;
    until $g(S_i) = 0$ or stopping criteria met;
    $S \leftarrow S_i$;
end
```

Algorithm 1: Genet Procedure

can be made to the cost function g by changing the assignment of any of the variables. When Genet encounters a local minima the weights of all the constraints which are violated in the minima are increased. This increases the cost of violating constraints which are violated in the minima, thus increasing the value of the cost function g in the minima and enabling the network to escape to other states. The version of Genet described in this paper uses what we call *limited sideways* moves. Here we allow sideways moves to be made, however if the value of the cost function remains the same after two consecutive calls to *Hill-Climb* (or convergence cycles) we activate learning.

SwapGenet

SwapGenet works in a similiar way to Genet, except that the hill-climbing procedure is modified to select possible moves from a "swap" neighbourhood. This is inspired partly by the 2-Opt heuristic used in solving the travelling salesman problem (Lin & Kernighan, 1973).

When solving a sequencing problem SwapGenet uses sequence constraints to structure the search space. Firstly, an initial assignment of values to variables is generated in the following way: for each sequence constraint *sequence(Vars, Vals)* in C, each variable in *Vars* will be randomly assigned

```
procedure GENET-Hill-Climb(Z, D, C, $S_i$, $S_{i+1}$)
begin
    $S = S_i$;
    for $\langle x, v_i \rangle \leftarrow$ each element of $S_i$ do
        begin
            for $v \leftarrow$ each element of $D_x$ do
                $g_v \leftarrow g(S \quad (\langle x, v_i \rangle) + (\langle x, v \rangle))$;
            BestSet $\leftarrow$ set of values with minimum $g_v$;
            $v_{i+1} \leftarrow$ random value in BestSet;
            $S \leftarrow S \quad (\langle x, v_i \rangle) + (\langle x, v_{i+1} \rangle)$;
        end
    $S_{i+1} = S$;
end
```

Algorithm 2: GENET hill-climbing.

a value in *Vals*, such that each value in *Vals* is assigned to exactly one variable. Thus the sequence constraints will be satisfied in the initial state of the search.

Sequence constraints are also used to constrain the search space when hill-climbing. Pseudo-code for SWAPGENET hill-climbing is given in algorithm 3. The procedure is as follows: A variable x is selected to be repaired. The cost of swapping the value of x with the value of another variable y is determined for all variables which *do not have the same value as* x. The variable for which a swap of values would result in the lowest value of g for the state resulting from the swap is then selected with any ties being broken randomly. A new state generated by a swap will always be a permutation of the previous state—thus every search space state in SWAPGENET will satisfy all the sequence constraints.

Example 2 We may generate the initial state for the CSP in figure 1 to be $(\langle A, 1\rangle, \langle B, 2\rangle, \langle C, 3\rangle, \langle D, 4\rangle)$, which satisfies the alldifferent constraint. Three constraints are violated in this state: *even(A + D), even(B + C)* and $A > B$. We can improve this assignment by swapping the values of variables A and B, resulting in the state $(\langle A, 2\rangle, \langle B, 1\rangle, \langle C, 3\rangle, \langle D, 4\rangle)$ which satisfies all the constraints. Since we only generate new states by swapping the values of two variables the alldifferent constraint will be satisfied in all the states of the search space.

Sequence constraints can be used to specify requirements other than the

```
procedure SwapGENET-Hill-Climb(Z, D, C, $S_i$, $S_{i+1}$)
begin
    $S = S_i$;
    for $x \leftarrow$ each element of $Z$ do
        begin
            $v_x \leftarrow$ value of $x$;
            for $\langle y, v_y \rangle \leftarrow$ each element of $S$ do
                if sequence(Vars, Vals) $\in C$ and $\{x, y\} \subset$ Vars
                    if $v_x \neq v_y$
                        $g_y \leftarrow g(S \quad (\langle x, v_x \rangle, \langle y, v_y \rangle) + (\langle x, v_y \rangle, \langle y, v_x \rangle))$;
            BestSet $\leftarrow$ set of values with minimum $g_y$;
            $z \leftarrow$ random value in BestSet;
            if $z \leq g(S)$ then
                $S \leftarrow S \quad (\langle x, v_x \rangle, \langle z, v_z \rangle) + (\langle x, v_z \rangle, \langle z, v_x \rangle)$;
        end
    $S_{i+1} = S$;
end
```

Algorithm 3: SwapGenet hill-climbing.

alldifferent constraint. For instance they can replace production constraints in scheduling problems. An application of this is described in the next section for the car-sequencing problem.

Empirical evaluation

Although the search space should always be smaller for SwapGenet than for Genet on a given sequencing problem, this does not necessarily mean that SwapGenet will always solve such problems in less time, since the CPU time required to determine which swap to make for SwapGenet may be higher than that of determining which repair to make for Genet. The main reason for this is that since the values of two variables will be affected by a swap the number of constraints this will affect will be larger than for changing the value of a single variable, hence more CPU time will be needed to evaluate the effect of a swap. (Zweben, Davis, Daun, & Deale, 1993) discuss the same issue.

Furthermore the neighbourhood of possible moves may be larger for SwapGenet than for Genet. Given a problem with n variables and a domain of values D to sequence, if there are d distinct values in D then SwapGenet has a neighbourhood of moves whose size is proportional to n, whereas the size of the neighbourhood for GENET is proportional to d.

Thus the purpose of empirical evaluation was to determine whether in practice SWAPGENET was faster at solving constraint satisfaction sequencing problems than GENET. Although any extra cost in determining moves will be countered by a reduction in constraints to satisfy and search space size, this may not be worthwhile for all problems. We compared SWAPGENET and GENET the car sequencing problem, and present results below.

The car sequencing problem

The car-sequencing problem is a real-life general CSP which is considered particularly difficult due to the presence of global atmost constraints (Parrello & Kabat, 1986; Dincbas et al., 1988).

In modern car production, cars are placed on conveyor belts which move through different work areas. Each of these work areas specialises to do a particular job, such as fitting sunroofs, car radios or air-conditioners. When a car enters a work area, a team of engineers in that area travels with the car while working on it. The production line is designed so as to allow enough time for the engineers to finish their job while the car is in their work area. For example, if the time taken to install a sunroof is 20 minutes, and one car enters the conveyor belt every four minutes, then the area for sunroof installation will be given a capacity of carrying five cars.

A production line is normally required to produce cars of different models. The number of cars required for each model is called the *production requirement*. Since cars of different models require different options to be fitted, not every car requires work to be done in every work area. For example, one model may need air-conditioning and power brakes to be installed, but not a sunroof.

Each work area is constrained by its resource constraint. For example, if three teams of engineers are designated to fitting sunroofs, and the sunroof area has a space capacity for five cars, then the sunroof work area can cope with no more than three out of five cars requiring the fitting of sunroofs in any sub-sequence of cars on the conveyor belt. If more than three cars in any sequence of five cars require sunroofs, then the engineers would not have time to finish before the conveyor belt takes the cars away. The ratio 3/5 is called the *capacity constraint* of the work area for the sunroof.

Given a number of car types with their option requirements, we can specify a car sequencing problem by its production constraints and capacity constraints. Production constraints are global constraints which, along with the capacity constraints, make the car sequencing problem particularly difficult.

Example 3 We can express the car sequencing problem as a CSP by letting each variable represent a position on the conveyor belt and the domain of each variable be the types of cars to be scheduled. For instance, table 1 shows a car sequencing problem where we want to schedule fifty cars, each of which may be one of three types. Here car type 1 requires sunroof and radio to be fitted, but no air-conditioning. For this problem we would have fifty variables, each with a domain of $\{1, 2, 3\}$. We can represent the production requirements in GENET with atmost constraints. Atmost constraints are of the form *atmost(n, Var, Val)* and specify that at most n variables from the set *Var* may take values from the set *Val*. The production requirements for the problem given in table 1 can be represented by three atmost constraints: $\text{atmost}(10, \{1, \ldots, 50\}, \{1\})$, $\text{atmost}(20, \{1, \ldots, 50\}, \{2\})$ and $\text{atmost}(20, \{1, \ldots, 50\}, \{3\})$. However to represent this as a sequencing problem we represent all the production constraints by a single sequence constraint. In this case the sequence constraint would be:

	Car Type			*Capacity Constraints*
options	*type 1*	*type 2*	*type 3*	
sunroof	1	1	0	2/3
radio	1	0	1	3/4
air-conditioning	0	1	1	2/3
required	10	20	20	

Table 1: An example of a small car sequencing problem

$$\textit{sequence}(\{1, \ldots, 50\}, [\![1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2 \\ 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]\!])$$

The capacity constraints are also implemented as atmost constraints. In our example option 1 has a capacity constraint of 2/3. This means that we need a capacity constraint on option 1 for every subsequence of three consecutive variables. However, since our variables are representing car types, not car options, we need to constrain all the car types which have option 1 *i.e.*, car types 1 and 2. So, for option 1 the capacity constraints will be atmost $(2, \{1, 2, 3\}, \{1, 2\}))$, atmost $(2, \{2, 3, 4\}, \{1, 2\})$, ..., atmost $(2, \{48, 49, 50\}, \{1, 2\})$.

A measure of the difficulty of a car sequencing problem is given by its *mean utilisation percentage.* The utilisation percentage of an option i for a N-car sequencing problem expresses the ratio of number of cars requiring option i to the maximum number of cars which could have option i while satisfying the capacity constraint on i. For instance if the capacity constraint on i is $2/4$ and $N = 100$ then the maximum number of cars in a sequence which could take option i is 50. If, to satisfy the production requirements, 40 cars need option i then the utilisation percentage of option i is 80%. The mean utilisation percentage of a car sequencing problem is the mean of the utilisation percentages for all its options.

In our experiments we used randomly generated problems of size 200 cars. We varied the mean utilisation percentage from 60% to 90%. At each utilisation percentage we generated ten problems following the specification given in (Dincbas et al., 1988). The problems all had 200 variables with uniform domains varying from 17 to 28 values and approximately 1000 atmost constraints of varying arity. All the problems we used were soluble.

Tables 2 and 3 present the results of our experiments. GENET and SWAPGENET were both run ten times on each problem. SWAPGENET takes less repairs to solve car sequencing problems at all utilisation percentages than GENET. However the extra time needed to evaluate which swaps to make mean that at low utilisation percentages it is actually slower than GENET. For the harder problems at higher utilisation SWAPGENET easily outperforms GENET in CPU time as well as number of repairs needed to solve these problems.

utilisation %	*mean* CPU *time (sec.)*	*median repairs*	*mean repairs*	*lowest repairs*	*highest repairs*	*std.dev. repairs*
60	1.40	470	927	209	7960	1104
65	1.40	496	879	207	4915	903
70	1.78	523	921	229	7130	1024
75	2.65	732	1033	282	5146	916
80	12.66	2077	2593	530	15147	1987
85	22.51	2939	3452	796	11640	2175
90	186.11	8652	11164	2626	99402	10797

Table 2: GENET on 200 car sequencing problems.

utilisation %	*mean* CPU *time (sec.)*	*median repairs*	*mean repairs*	*lowest repairs*	*highest repairs*	*std.dev. repairs*
60	2.70	317	453	132	2533	376
65	2.55	319	402	129	1635	285
70	3.32	332	477	160	3781	434
75	4.66	354	601	301	6649	724
80	8.45	739	948	305	3032	621
85	10.47	775	1017	323	4296	698
90	22.67	1314	1508	585	6343	824

Table 3: SwapGenet on 200 car sequencing problems.

Conclusions

We have shown that for the constraint satisfaction sequencing problem the search space will be smaller if the problem is represented specifically as a sequencing problem rather than as a generic constraint satisfaction problem. Furthermore a sequencing formulation will implicitly satisfy the constraints needed to specify that a problem is a sequencing problem in the generic constraint satisfaction formulation.

We have presented a new iterative repair algorithm, SwapGenet, for solving the constraint satisfaction sequencing problem. SwapGenet is derived from Genet, a min-conflicts based algorithm for solving constraint satisfaction problems. SwapGenet differs from Genet in the way it performs hill-climbing: whereas Genet makes moves by modifying the assignment of a single variable at a time, SwapGenet makes moves taken from a neighbourhood of possible swaps of the values of two variables.

Through empirical evaluation we have shown that although SwapGenet takes less repairs to solve easy sequencing problems than Genet, in practice it requires more CPU time since the cost of evaluating possible repairs is often higher for SwapGenet than for Genet. However for hard constraint satisfaction sequencing problems this extra repair cost is more than countered by the reduction in constraints and search space size, so that SwapGenet outperforms Genet both in terms of number of repairs and the CPU time needed to solve these problems. Thus we believe SwapGenet to be of use for hard constraint satisfaction sequencing problems.

Acknowledgements

The authors are very grateful for the help of Dr. John Ford in constructing the proof of equation 2. We would like to thank James Borrett, Alvin Kwan and Chris Voudouris for useful discussions and comments on this paper. Andrew Davenport is supported by an Engineering and Physical Sciences Research Council Ph.D studentship award. This research has also been supported by an Engineering and Physical Sciences Research Council grant (ref. GR/H75275).

References

Davenport, A. J., Tsang, E. P. K., Wang, C. J., & Zhu, K. (1994). GENET: A connectionist architecture for solving constraint satisfaction problems by iterative improvement. In *Proceedings of AAAI-94*, Vol. 1, pp. 325–330.

Diller, A. (1990). *Z: An Introduction to Formal Methods.* Chichester: Wiley.

Dincbas, M., Simonis, H., & Van Hentenryck, P. (1988). Solving the car-sequencing problem in logic programming. In *Proceedings of ECAI-88*, pp. 290–295.

Lin, S., & Kernighan, B. W. (1973). An effective heuristic algorithm for the travelling salesman problem. *Operations Research*, *21*, 498–516.

Minton, S., Johnston, M., Philips, A., & Laird, P. (1992). Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, *58*, 161–205.

Morris, P. (1993). The breakout method for escaping from local minima. In *Proceedings of AAAI-93*, pp. 40–45. AAAI Press/The MIT Press.

Parrello, B., & Kabat, W. C. (1986). Job-shop scheduling using automated reasoning: A case study of the car-sequencing problem. *Journal of Automated Reasoning*, *2*, 1–42.

Regín, J. C., & Puget, J.-F. (1997). A filtering algorithm for global sequencing constraints. In *Proceedings of Principles and Practice of Constraint Programming*, pp. 32–46.

Smith, B. M. (1996). Succeed-first or fail-first: A case study in variable and value ordering. In *Proceedings ILOG Solver and ILOG Scheduler Second International Users' Conference.*

Tsang, E. P. K. (1993). *Foundations of Constraint Satisfaction.* Academic Press.

Zweben, M., Davis, E., Daun, B., & Deale, M. (1993). Informedness vs. computational cost of heuristics in iterative repair scheduling. In *Proceedings of IJCAI-93*, pp. 1416–1422.