A Tabu Search Evolutionary Algorithm for Solving Constraint Satisfaction Problems

B.G.W. Craenen & B. Paechter

Napier University
10 Colinton Rd, Edinburgh, EH10 5DT
{b.craenen|b.paechter}@napier.ac.uk
http://www.soc.napier.ac.uk

Abstract. The paper introduces a hybrid Tabu Search-Evolutionary Algorithm for solving the constraint satisfaction problem, called STLEA. Extensive experimental fine-tuning of parameters of the algorithm was performed to optimise the performance of the algorithm on a commonly used test-set. The performance of the STLEA was then compared to the best known evolutionary algorithm and benchmark deterministic and non-deterministic algorithms. The comparison shows that the STLEA improves on the performance of the best known evolutionary algorithm but can not achieve the efficiency of the deterministic algorithms.

1 Introduction

The last two decades saw the introduction of many evolutionary algorithms (EAs) for solving the constraint satisfaction problem (CSP). In [1], the performance of a representative sample of these EAs was compared on a large randomly generated test-set of CSP-instances. In [2] a more extensive comparison, including a large number of algorithm variants, was included, this time on a test-set generated by the latest random CSP generator. One variant algorithm, the Stepwise-Adaptation-of-Weights EA with randomly initialisated domain sets (rSAWEA) outperformed all other EAs. However, when the effectivity and the efficiency of this algorithm was compared to non-evolutionary algorithms, it was found that the effectivity of the other algorithms was approached by the rSAWEA but that the efficiency still fell short of the other algorithms.

A major reason for this lack of efficiency is that EAs tend to recheck previously checked candidate solutions during their run, wasting computational effort. This paper investigates a way of reducing this waste: the use of a tabu list.

Tabu lists are used in Tabu Search (TS) algorithms ([3]). They are used to ensure that the algorithm does not return to an already searched neighbourhood or check a candidate solution twice. Tabu lists and TS have found their way into EAs before (i.e. [4–6]) but to the authors' knowledge never for EAs solving the CSP. Tabu lists, in essence, store already checked candidate solutions. The algorithm can use the list to determine future search avenues or simply to forgo checking the candidate solution: making it tabu. Because the (more simple) tabu

lists are used as reference memory (only insertion and lookup is allowed), they can be implemented efficiently as a hash set. This ensures a constant time cost ($O(1)$) when a suitable hash function is used and the table is sufficiently large. This paper will show that combining EAs with tabu lists will provide both an effective and efficient CSP solving algorithm.

The article is organised in the following way: in section 2, the constraint satisfaction problem is defined. Section 3 defines the proposed algorithm. The experimental setup is explained in section 4. Section 5 discusses the results of the experiments. Finally, the paper is concluded in section 6.

2 Constraint Satisfaction Problems

The *Constraint Satisfaction Problem* (CSP) is a well-known satisfiability problem that is NP-complete ([7]). Informally, the CSP is defined as a set *variables* X and a set of *constraints* C between these variables. Variables are only assigned values from their respective *domains*, denoted as D. Assigning a value to a variable is called *labelling* a variable and a *label* is a variable-value pair, denoted: $\langle x, d \rangle$. The simultaneous assignment of several values to their variables is called a *compound label*. A constraint is a set of compound labels, this set used to determine when a constraint is *violated*. If a compound label is not in a constraint, it *satisfies* the constraint. A compound label that violates a constraint is called a *conflict*. A *solution* of the CSP is defined as the compound label containing all variables in such a way that no constraint is violated. The number of distinct variables in the compound labels of a constraint is called the *arity* of the constraint and these variables are said to be relevant to the constraint. The arity of a CSP is the maximum arity of its constraints. In this paper we consider only CSPs with an arity of two, called *binary CSPs*. All constraints of a binary CSP have arity two.

In this paper we will use the test-set constructed in [2]. The test-set consists of model F generated solvable CSP-instances ([8]) with 10 variables and a uniform domain size of 10 values. Complexity of the instances is determined by two commonly used complexity measures for the CSP: density (p_1) and average tightness ($\overline{p_2}$), both presented as a real number between 0.0 and 1.0 inclusive. The mushy region is the region in the density-tightness parameter space where the hard-to-solve CSP-instances can be found. The nine density-tightness combinations used are $1 : (0.1, 0.9)$, $2 : (0.2, 0.9)$, $3 : (0.3, 0.8)$, $4 : (0.4, 0.7)$, $5 : (0.5, 0.7)$, $6 : (0.6, 0.6)$, $7 : (0.7, 0.5)$, $8 : (0.8, 0.5)$, and $9 : (0.9, 0.4)$. For these density-tightness combinations 25 CSP-instances were selected from a population of 1000 generated CSP instances (for selection criteria see [2]) for a total of 225 CSP-instances. The test-set can be downloaded at: `http://www.emergentcomputing.org/csp/testset_mushy.zip`.

3 The Algorithm

The proposed EA is called the *Simple Tabu List Evolutionary Algorithm* (STLEA) and is a hybrid between a TS algorithm and an EA. In keeping with the simple definition of TS as "a meta-heuristic superimposed on another heuristic" ([3]), the STLEA only uses the tabu list. The tabu list is used to ensure that the STLEA does not check a compound label twice during a run. The basic structure of the STLEA is similar to other EAs and is shown in algorithm 1. A population P of *popsize* individuals is initialised (line 2) and the compound labels in the population are added to the tabu list (line 3). The individuals' representation and how they are initialised are described in section 3.1, the tabu list is described in section 3.3. The STLEA iterates for a number of generations (line 4 to 10) until either a solution is found or the maximum number of conflict checks allowed ($maxCC$) has been reached or exceeded (The *stop condition* in line 4). At each iteration parents are selected from P into offspring population S using biased linear ranking selection ([9]) with bias *bias* (line 5). The offspring population creates a new population using the variation operator (line 6), further described in section 3.4. The new offspring population is then evaluated by the objective function (line 7), described in section 3.2. Each new individual in the offspring population is also added to the tabu list (line 8). Finally, the survivor selection operator selects individuals from the offspring population (S) into an emptied population (P) to be used for the next generation (line 9). The survivor operator selects individuals with the best fitness value (see section 3.1) until the new population (P) is equal to *popsize*.

Algorithm 1: STLEA

```
1  funct STLEA(popsize, maxCC, bias) ≡
2     P := initialise(popsize);
3     updateTabuList(P);
4     while ¬solutionFound(P) ∨ CC < maxCC do
5           S := selectParents(P, bias);
6           S := variationOperator(S);
7           evaluate(S);
8           updateTabuList(S);
9           P := selectSurvivors(S);
10    od
```

3.1 Representation & Initialisation

An individual in the STLEA consists of three parts: a compound label over all variables of the CSP used as the candidate solution; the subset of constraints of the CSP that are violated by the compound label; and a parameter indicating which variable was altered in the previous generation (changed variable parameter). A new individual is then initialised by: uniform randomly labelling all variables in the compound label from the respective domains of each variable; adding

each constraint violated by the compound label to the set of violated constraints; and leaving the changed variable parameter *unassigned*. The biggest difference to other commonly used representations is that this representation maintains: the actual set of violated constraints instead of the derivative *number* of violated constraints; and the variable changed in the previous generation. The size of the constraint set is used if a fitness value for the individual is needed. By numbering the constraints of the CSP, we can store only this number as a reference to the actual constraint.

3.2 Objective function

The objective of STLEA is to minimise the number of violated constraints, thus finding a solution. The objective function then maintains the set of violated constraints of an individual. The number of conflict checks needed for one fitness evaluation is reduced by only considering the constraints relevant to the last changed constraint. First all constraints relevant to the last changed variable are removed from the set of violated constraints of the individual. The objective function then checks each constraint relevant to the last changed variable of the individual. If it is violated, the constraint is added to the set of violated constraints of the individual.[1]

3.3 Simple Tabu List

The STLEA maintains a simple tabu list of compound labels implemented as a hash set. The tabu list is used in only two ways: adding a compound label (insertion), and checking if a compound label is in the list (lookup). There is no need to alter or remove a compound label once it has been added to the tabu list. New compound labels are added immediately after the new individuals have been evaluated. Depending on the quality of the hash-function and given adequate size of the hash table, insertion and lookup in a hash table set constant time ($O(1)$).

3.4 Variation Operator

The variation operator takes a single individual to produce many children (offspring). The basic premise of the variation operator is simple: select a variable from the CSP and generate children for all not previously checked values in the domain of the selected variable. The variation operator uses the tabu list to check whether a child has already been checked. All not previously checked children are added to the offspring population, and the last changed variable parameter is set to the selected variable.

If all children are in the tabu list, the variation operator iterates the procedure with another variable selected. No variable will be selected twice in one operator

[1] This objective function only works when only one variable is changed, although a version where more than one variable is changed can be defined analogously.

invocation per individual. It is possible that after all variables have been selected, no unchecked child was found. At this stage, the search environment around the individual has been exhaustively searched and the search path can be terminated. At this point the variation operator inserts a new randomly initialised individual into the offspring population, in effect, starting a random new search path. This is, in essence, a gradual restart strategy. The variation operator never selects the variable selected in the previous generation, since all values for that variable have already been checked in the previous generation of the algorithm.

The variation operator selects the variable in three stages, uniform randomly from the set of variables:

1. relevant to the constraints violated by the individual's compound label (*first stage variable set*);
2. related to but excluding the variables in the first stage variable set by constraint arc (*second stage variable set*); and
3. that are not in the previous two sets (*third stage variable set*).

The first stage variable set is created by adding all relevant variables for each constraint in the violated constraints set of the individual to a multiset. A multiset is used so that variables relevant to more than one violated constraint have a higher chance of being selected. This provides a higher chance to satisfy more than one constraint by a single relabelling.

The second stage variable set is a multiset of variables, excluding the variables of the first stage variable set but including those variables that are relevant to constraints that have a relevant variable in the first stage variable set. These variables are said to be *relevant-by-arc* to a violated constraint. After all variables from the first stage variable set have been tried, it is necessary to expand the local-search neighbourhood. It may be useful to change the value of a relevant-by-arc variable to another value first to escape the local-search neighbourhood. The second stage variable set gives a higher selection chance to variables that are relevant-by-arc to more violated constraints.

The third stage variable set includes all variable not in the previously two variable sets. Since no preference can be established, all variables in the set have equal probability for selection.

4 Experimental Setup

The STLEA is run on the test-set used in [2] (see section 2). Two measures are used to assess the performance of the algorithm: the success rate (SR), and the average number of conflict checks to solution ($ACCS$). The SR will be used to describe the effectiveness of the algorithm, the $ACCS$ will be used to describe the efficiency of the algorithms.

The SR of an algorithm is calculated by dividing the number of successful runs by the total number of runs. A successful run is a run in which the algorithm solved the CSP-instance. The SR is given as a real number between 0.0 and 1.0 but can also be expressed as a percentage. A SR of 1.0 means that all runs

were successful. The SR is the most important performance measure to compare algorithms on, after all, an algorithm which finds more solutions should be valued over an algorithm that does not. The accuracy of the SR measure is influenced by the total number of runs.

The $ACCS$ of an algorithm is calculated by averaging the number of conflict checks needed by an algorithm over several successful runs. A conflict check is the check made to see if a compound label is in a constraint. Unsuccessful runs of an algorithm are discarded, and if all runs of an algorithm are unsuccessful, the $ACCS$ measure is undefined. The $ACCS$ measure is a secondary measure for comparing an algorithm and its accuracy is affected by the number of successful runs as well as the total number of runs of an algorithm (the ratio of which is the SR).

A efficiency performance measure has to account for the computational effort of an algorithm. The $ACCS$ measure uses the number of conflict checks as the atomic measure to quantify the computational effort. The STLEA, however, also spends computational effort on the maintenance of the tabu list. It was found that the computational effort needed to insert and lookup compound labels in the tabu list was negligible in comparison to the computational effort of performing a conflict check when the CSP-instance to solve was sufficiently complex. The computational effort needed to maintain the tabu list became relatively substantial when the average number of relevant constraints to a variable in the CSP-instance is smaller than two. This was not the case for the CSP-instances in the test-set.

The STLEA has relatively few parameters to fine-tune: the *popsize*; the *maxCC* allowed; and the bias of the biased linear ranking parent selection operator. We chose to select an equal number of parents for use by the variation operator as there were individuals in the population (*popsize*). A bias of 1.5 for the biased linear ranking selection operator was used because this gave the best performance in preliminary experiments, and is also used in other studies ([1, 2]).

This leaves just the *popsize* and *maxCC* parameters to fine-tune. With EAs for solving CSPs it is common practice to use a small population size. The reasoning is that with a small population to maintain, more computational effort can be spend on increasing the fitness of the individuals, following the (small number of) search paths of which they are part. Large populations, on the other hand, need a lot of computational effort to maintain but provide for more search avenues to explore; in general keeping population diversity high. The trade-off, is investing computational effort, either in following a few search paths in depth, or in maintaining many search paths but (perhaps) following them to a lesser depth. The STLEA, however, doesn't seem to lend itself well to conventional wisdom. The combination of a tabu list and a powerful local search technique appears to address both issues at simultaneously. Since the common practice does not seem to apply to the STLEA, only experimentation with a large number of combinations for the *popsize* and *maxCC* parameters can provide guidelines.

The experimental setup for the proposed algorithm is then as follows: all 225 CSP-instance in the test-set the algorithm is run 10 times for a total of 2250 runs. The *popsize* and *maxCC* parameters are varied. The *popsize* parameter is taken from the following set: $\{10\} \cup \{50, 100, \ldots, 5000\}$. The *maxCC* is taken from the following set: $\{100000, 200000, \ldots, 5000000\}$. In total $2250 \cdot 101 \cdot 50 = 11,362,500$ runs were performed.

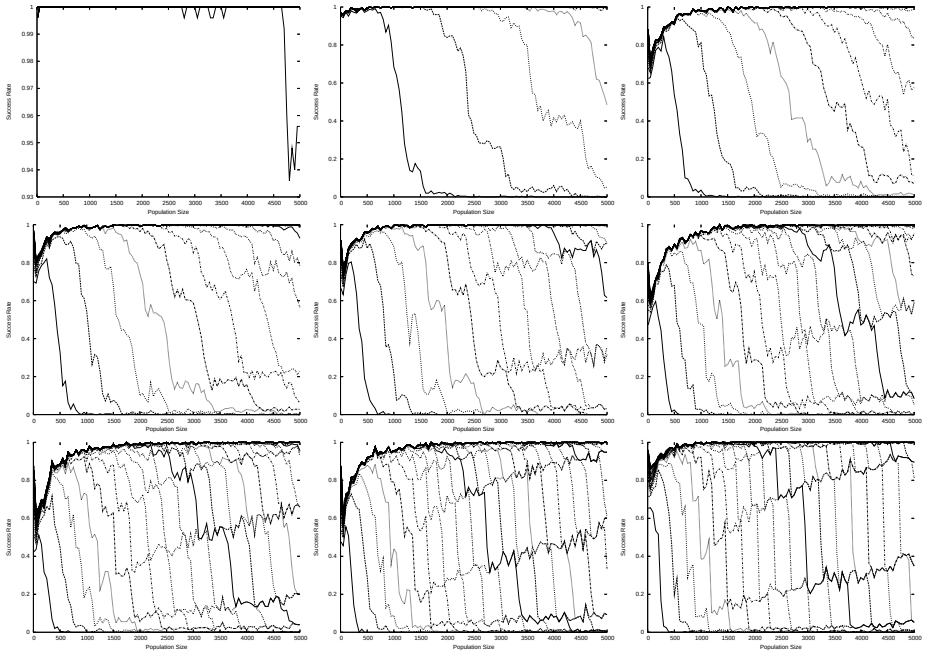

Fig. 1. The relationship between the population size (x-axis) and the success rate (y-axis) of the algorithm for different maximum number of conflict checks allowed.

5 Results

The results of the experiments are summarised in figure 1. Figure 1 consists of 9 graphs, each showing the result for each density-tightness combination in the test-set. The top row shows the results for density-tightness combinations 1 to 3, the middle row the results for density-tightness combinations 4 to 6, and the bottom row the results for density-tightness combinations 7 to 9.

Figure 1 shows the influence of different values for *maxCC* on the *SR* for different values of *popsize*. The trend for the *SR* is the same for all different density-tightness combinations. The *SR* increases when larger values for *popsize* are used. The *SR* drops abruptly when *popsize* gets too large relative to *maxCC*.

At the point where the algorithm achieves maximum *SR*, *maxCC* is just enough to allow the algorithm to reach this peak but not much more. If the *popsize* is increased beyond this point, the *maxCC* for maintaining a population of this size are not available and the *SR* drops abruptly. The inner most arc seen from the left-bottom corner of the graph (worst performance) invariably depicts the experiments with the least *maxCC*. Note that the difference in complexity of the density-tightness combinations in the test-set is also apparent by the *maxCC* allowed. Density-tightness combination 1 for example is known to be easier to solve than density-tightness combination 9, and the number of conflict checks needed to sustain the population while reaching a success rate of 1.0 is therefore lower for the first then for the latter. Note also the stepwise drop in *SR* after the optimal *SR* has been reached. Each step is caused be the inability of the algorithm to perform another generation. The slight increase in *SR* at each step is cause by the maximisation of the *popsize* for the number of generations that can still be performed.

	SR	**ACCS**	**popsize**	**maxCC**
1	1.0	2576	50	100000
2	1.0	67443	550	200000
3	1.0	313431	1650	500000
4	1.0	397636	1800	600000
5	1.0	319212	1150	500000
6	1.0	469876	1350	800000
7	1.0	692888	1750	1100000
8	1.0	774929	1700	1400000
9	1.0	442323	900	800000

Table 1. Success rate (*SR*) and average conflict checks to solution (*ACCS*) for the best population size (*popsize*) and maximum conflict checks allowed (*maxCC*) parameters.

Table 1 shows the first parameter-combination (*popsize*-*maxCC*) for which the *SR* is 1.0 for each density-tightness, with *popsize* minimised first and *maxCC* second. There is significant difference between the parameter values for different density-tightness combinations. CSP-instances for density-tightness combination 1 for example can be solved with $popsize = 50$ and $maxCC = 100000$ while density-tightness combination 7 needs $popsize = 1750$ and $maxCC = 1100000$. This reflects the difference in effort needed for solving the CSP-instances for the different density-tightness combination more then an inherent aptitude for the different density-tightness combinations of the algorithm.

Table 2 shows a comparison of the performance of the STLEA with the best algorithm from [2] and some benchmark algorithms. Table 2 clearly shows that the STLEA outperforms rSAWEA in *SR* and *ACCS* on all but density-tightness combination 9. Especially the fact that, given a large enough population

and allowed number of conflict checks to work, the STLEA has a success rate of 1.0 is an improvement on rSAWEA. The STLEA also compares favourable with the HCAWR algorithm, with efficiency of the algorithm on average several magnitudes better (except density-tightness combination 9). Compared with the deterministic algorithms CBA and FCCDBA, however, the STLEA still has, on average, inferior efficiency, both algorithms outperforming it by several orders of magnitude (except for density-tightness combinations 1 and 2 for CBA). Overall, the STLEA is more effective and efficient then the best EA published so far but, although a step in the right direction, is still unable to beat deterministic algorithms on efficiency.

In [2] the notion of *memetic overkill* was introduced as well. Memetic overkill occurs when an algorithm for solving CSPs incorporates a heuristic so capable of finding solutions that the evolutionary components actually hamper performance. Especially hybrid algorithms are susceptible to suffer from memetic overkill. De-evolutionarising the STLEA through additional experiments (as explained in [2]) showed that the STLEA does not suffer from memetic overkill.

	STLEA		**rSAWEA**		**HCAWR**		**CBA**		**FCCDBA**	
	SR	**ACCS**	**SR**	**ACCS**	**SR**	**ACCS**	**SR**	**ACCS**	**SR**	**ACCS**
1	1.0	2576	1.0	9665	1.0	234242	1.0	3800605	1.0	930
2	1.0	67443	0.988	350789	1.0	1267015	1.0	335166	1.0	3913
3	1.0	313431	0.956	763903	1.0	2087947	1.0	33117	1.0	2186
4	1.0	397636	0.976	652045	1.0	2260634	1.0	42559	1.0	4772
5	1.0	319212	1.0	557026	1.0	2237419	1.0	23625	1.0	3503
6	1.0	469876	1.0	715122	1.0	2741567	1.0	44615	1.0	5287
7	1.0	692888	1.0	864249	1.0	3640630	1.0	35607	1.0	4822
8	1.0	774929	1.0	1012082	1.0	2722763	1.0	28895	1.0	5121
9	1.0	442323	1.0	408016	1.0	2465975	1.0	15248	1.0	3439

Table 2. Comparing the success rate and average conflict checks to solution of the STLEA, the Stepwise-Adaptation-of-Weights EA with randomly initialised domain sets (rSAWEA), Hillclimbing algorithm with Restart (HCAWR), Chronological Backtracking Algorithm (CBA), and Forward Checking with Conflict-Directed Backjumping Algorithm (FCCDBA).

6 Conclusion

In this paper we introduced a hybrid Tabu Search — Evolution Algorithm for solving the CSP, called Simple Tabu List Evolutionary Algorithm (STLEA). In [2] it was found that EAs for solving the CSP were able to approach the effectiveness of other (deterministic) algorithms but that they were still far behind in

efficiency while doing so. A reason behind this lack of efficiency is the tendency of EAs to recheck previously checked compound labels during their search for a solution. The rational behind the STLEA is to reduce this rechecking by using a tabu list, effectively making previously checked compound labels tabu. The basic structure of the STLEA resembles the basic EA structure but incorporates a local-search technique into a single variation operator. A slightly altered representation allows for further efficiency improvement as well. A large number of experiments were performed for different combinations of the algorithm's parameters in order to find the best parameter settings. Using these parameters, it was found that the STLEA outperforms the best EA for solving the CSP published so far but still has inferior efficiency to deterministic algorithms.

Future research is focussed on comparing the relative behaviour of the STLEA to other algorithms when the complexity of the CSP-instances is increased (scale-up experiments) and the effects on the performance of the STLEA when other kinds of tabu lists are used.

References

1. Craenen, B., Eiben, A., van Hemert, J.: Comparing evolutionary algorithms on binary constraint satisfaction problems. IEEE Transactions on Evolutionary Computing **7**(5) (2003) 424–445
2. Craenen, B.: Solving Constraint Satisfaction Problems with Evolutionary Algorithms. Doctoral dissertation, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands (2005)
3. Glover, F., Laguna, M.: Tabu search. In Reeves, C., ed.: Modern Heuristic Techniques for Combinatorial Problems. Blackwell Scientific Publishing, Oxford, England (1993) 70–141
4. Costa, D.: An evolutionary tabu search algorithm and the nhl scheduling problem. Orwp 92/11, Ecole Polytechnique Fédérale de Lausanne, Département de Mathématiques, Chaire de Recherche Opérationelle (1992)
5. Burke, E., Causmaecker, P.D., VandenBerghe: A hybrid tabu search algorithm for the nurse rostering problem. In: Proceedings of the Second Asia-Pasific Conference on Simulated Evolution and Learning. Volume 1 of Applications IV. (1998) 187–194
6. Greistorfer, P.: Hybrid genetic tabu search for a cyclic scheduling problem. In Voß, S., Martello, S., Osman, I., Roucairol, C., eds.: Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization, Boston, MA, Kluwer Academic Publishers (1998) 213–229
7. Rossi, F., Petrie, C., Dhar, V.: On the equivalence of constrain satisfaction problems. In Aiello, L., ed.: Proceedings of the 9th European Conference on Artificial Intelligence (ECAI'90), Stockholm, Pitman (1990) 550–556
8. MacIntyre, E., Prosser, P., Smith, B., Walsh, T.: Random constraint satisfaction: theory meets practice. In Maher, M., Puget, J.F., eds.: Principles and Practice of Constraint Programming – CP98, Springer Verlag (1998) 325–339
9. Whitley, D.: The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In Schaffer, J., ed.: Proceedings of the 3rd International Conference on Genetic Algorithms, San Mateo, California, Morgan Kaufmann Publisher, Inc. (1989) 116–123