A Conflict Tabu Search Evolutionary Algorithm for Solving Constraint Satisfaction Problems

B.G.W. Craenen and B. Paechter

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

Abstract. This paper introduces a hybrid Tabu Search - Evolutionary Algorithm for solving the binary constraint satisfaction problem, called CTLEA. A continuation of an earlier introduced algorithm, called the STLEA, the CTLEA replaces the earlier compound label tabu list with a conflict tabu list. Extensive experimental fine-tuning of parameters was performed to optimise the performance of the algorithm on a commonly used test-set. Compared to the performance of the earlier STLEA, and benchmark algorithms, the CTLEA outperforms the former, and approaches the later.

1 Introduction

Solving constraint satisfaction problems (CSP) with evolutionary algorithms (EAs) has been studied extensively over the years. This has resulted in the introduction of a large number of algorithms. A study of the performance of a representative sample of EAs, using a large randomly generated test-set, was carried out in [1]. A more comprehensive study, using an updated test-set, also including a large number of algorithm variants, can be found in [2]. There, it was found that one algorithm variant, the Stepwise-Adaptation-of-Weights EA with randomly initialised domain sets (rSAWEA), outperformed all other EAs. However, when comparing the effectiveness and efficiency of this algorithm with non-evolutionary algorithms, it was found that although the former could be approximated, the algorithm still fell short of achieving the later.

A reason for this lack of efficiency was identified to be the lack of preventing EAs from traversing already studied search-paths, something non-evolutionary algorithms are usually prevented from doing. In [3] therefore, the Simple Tabu List Evolutionary Algorithm (STLEA) was introduced, using a tabu list preventing it from wasting computational effort on already traversed search-paths. Tabu lists are a part of the Tabu Search (TS) meta-heuristic ([4]). They are used to ensure that an algorithm does not return to an already searched neighbourhood by making it tabu. The Tabu Search meta-heuristic has found its way into EAs before (e.g. [5,6,7,8]), especially for EAs handling constrained problems. An important feature of tabu lists is that they are only referenced, i.e., only insertion and look-up of elements is used. As such, they can be implemented as a hash-set, ensuring constant time cost when a suitable hash function is chosen. In [3] it was found that the STLEA, with a tabu list containing candidate solutions, increased efficiency enough to surpass the rSAWEA, with equal or better effectiveness. However, while comparing the performance of the STLEA to that of non-deterministic algorithms, it was still found to have equal effectiveness but inferior efficiency. Although, in [3], an important step was made in the right direction, it was still not good enough to beat deterministic algorithms.

This paper endeavours to set the next step by refining the tabu list used in the algorithm. Instead of focussing on candidate solutions for the tabu list, it focusses on storing conflicts instead. There are two advantages to this approach.

First, the number of conflicts in a CSP-instance is smaller than the number of possible candidate solutions, making the tabu list itself much easier to handle. This makes the tabu list approach more viable for large CSP-instances as well.

Second, a conflict tabu list can be used more directly to guide the search-path, than a candidate solution tabu list can. Whereas a candidate solution tabu list is only useful to exclude generated candidate solutions (individuals), the conflict tabu list can be used directly, for example by the crossover or mutation operator. This change of focus for the tabu list results in the introduction of the Conflict Tabu List Evolutionary Algorithm (CTLEA).

The paper is organised as follows: in section 2, a definition of constraint satisfaction problems is given. Section 3 defines the proposed algorithm. The experimental setup is explained in section 4. Section 5 discusses the results of the experiments, and finally in section 6, the conclusions that can be drawn from this paper are set forth.

2 Constraint Satisfaction Problems

The *Constraint Satisfaction Problem* (CSP) is a well-known NP-complete satisfiability problem ([9]). Defined informally as a set *variables* X and a set of *constraints* C between these variables, it only allows variables to be assigned values from their respective *domains*, denoted as $D_x, x \in X$. A *label* is then a variable-value pair, denoted: $\langle x, d\rangle, x \in X, d \in D_x$, and assigning a value to a variable is called *labelling* it. A *compound label* is a simultaneous assignment of several values to their respective variables, and a constraint is then a set of compound labels, with each compound label determining when the constraint is *violated.* A compound label not in a constraint is said to *satisfy* that constraint, while one that is, is called a *conflict.* A *solution* of a CSP is then defined as a compound label that contains all variables, but no conflicts from any constraint.

The number of distinct variables in the compound labels of a constraint is called the *arity* of that constraint, and these variables are said to be relevant to this constraint. The maximum arity of all constraints in a CSP is the arity of the CSP itself. In this paper, we only consider CSPs with an arity of two, meaning that all constraints in the CSP have arity two as well. Such CSPs are called *binary CSPs*. Restricting the arity of the studied CSPs is not a restriction in itself though, as [10] shows that every CSP can be transformed into an equivalent binary CSP.

This paper will use the same test-set constructed in [2], and [3]. It consists of model F generated solvable CSP-instances ([11]) with 10 variables and a uniform domain size of 10 values. Complexity of these instances is determined by using two 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. *Density* is the ratio between the maximum number of constraints of a CSP ($\binom{|X|}{2}$ for a binary CSP) and the actual number of constraints ($|C|$). The *tightness* of a constraint is defined as one minus the ratio between the maximum number of possible conflicts ($|D_{x_1} \times D_{x_2}|$ for a binary constraint relevant to variables x_1 and x_2) and the number of conflicts. The *average tightness* of a CSP is then the average tightness of all constraints in the CSP.

In the density-tightness parameter space of all randomly generated CSP-instances, the hard-to-solve instances can be found in what is called the *mushy region*. For the test-set used, the following density-tightness combinations lie within the mushy region $1 : (0.1, 0.9)$, $2 : (0.2, 0.9)$, $3 : (0.3, 0.8)$, $4 : (0.4, 0.7)$, $5 : (0.5, 0.7)$, $6 : (0.6, 0.6)$, $7 : (0.7, 0.5)$, $8 : (0.8, 0.5)$, and $9 : (0.9, 0.4)$. For each of these density-tightness combinations, 25 CSP-instances were selected from a population of 1000 randomly generated CSP-instances (see [2] for selection criteria). In total, the test-set includes $9 \cdot 25 = 225$ CSP-instances. The test-set can be downloaded at: `http://www.emergentcomputing.org/csp/testset_mushy.zip`.

3 The Algorithm

The *Conflict Tabu List Evolutionary Algorithm* (CTLEA) is an evolutionary algorithm using the Tabu Search meta-heuristic. In keeping with the simple definition of Tabu Search as “a meta-heuristic superimposed on another heuristic” ([4]), the CTLEA uses only the tabu list. In the STLEA, as described in [3], the tabu list was used to ensure that a compound label was not checked twice during a run. A major criticism of this type of tabu list is that the number of possible compound labels (candidate solutions), and therefore the amount of memory needed to maintain it, could become quite large, depending mostly on CSP parameters. The CTLEA therefore focusses on conflicts, and uses the tabu list to ensure that a *conflict* is not rechecked during a run. There are two advantages from using tabu lists in this way. First, the number of conflicts of a CSP-instance is smaller than the number of possible candidate solutions, addressing the criticism above. This not only makes the tabu list easier to maintain, but allows for the use of the tabu list for large CSP-instances as well. Second, a tabu list focussing on conflicts can be used by the algorithm to guide the search-path directly. Whereas a candidate solution tabu list is only useful to exclude whole candidate solutions (individuals), a conflict tabu list can be used by, for example, the crossover and mutation operators of the EA, to determine directly which variables and values can be labelled.

An example may provide more insight into the difference of size and subsequent cost of maintenance between a compound label and conflict tabu list. Let us consider the worst-case scenario for both tabu lists. The test-set used has 10 variables, and a uniform domain size of 10 elements. If the compound label tabu list were used, this means that in the worst-case, it should be able to store $10^{10} = 100,000,000,000$ compound labels, i.e., all possible candidate solutions. On the other hand, if the conflict tabu list were used, again in the worst-case, it should be able to store $\binom{10}{2} \times 10 \times 10 = 45 \times 10 \times 10 = 4500$ conflicts, i.e., all possible conflicts. This difference increases with scale as well. It must be noted however, that on average, only a fraction of the compound labels was stored by the STLEA in [3], although still substantially more than the number of conflicts stored by the CTLEA.

The basic structure of the CTLEA, shown in algorithm 1, was kept purposely close to that of the canonical EA. The biggest difference is that the CTLEA uses a single variance operator, called move-operator, instead of a separate crossover- and mutation operator.

The CTLEA works as follows. A population P of *popsize* individuals is initialised (line 2) and evaluated (line 3). The representation used by the individual, and initialisation is described in 3.1, the objective function used to evaluate them is described in section 3.2. The CTLEA then enters a while-loop wherein it iterates for a number of generations (line 4 to 9) 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 the beginning of each iteration of the algorithm, parents are selected from P into population S using biased linear ranking selection ([12]) with bias *bias* (line 5). These parents are used by the move-operator to create a new offspring population (line 6), as described in section 3.4. The new offspring population is then evaluated by the objective function (line 7). Finally, at the end of each iteration, the survivor selection operator selects individuals from the offspring population (S) into a new population (P) to be used for the next iteration/generation (line 8). No 'elitism' is used by the CTLEA, i.e., no individuals from the previous iteration/generation are forcefully preserved for the next iteration/generation. The tabu list, described in section 3.3, is used by both the objective function and the move-operator.

Algorithm 1: CTLEA

```
1 funct $CTLEA(popsize, maxCC, bias)$ $\equiv$
2   $P := initialise(popsize)$;
3   $evaluate(P)$;
4   while $\neg solutionFound(P) \vee CC < maxCC$ do
5         $S := selectParents(P, bias)$;
6         $S := moveOperator(S)$;
7         $evaluate(S)$;
8         $P := selectSurvivors(S)$;
9   od
```

3.1 Representation and Initialisation

A CTLEA individual consists of three parts: a compound label over all variables of the CSP used as a candidate solution; a subset of the constraints defined by the CSP, all violated by the candidate solution; and a field indicating which variable was altered previously, called the *changed variable field*. The variable stored by the changed variable field is called the changed variable.

A new individual is initialised by: labelling all variables in the compound label uniform randomly from the respective domains of each variable; initialising the subset of violated constraints to be empty (later to be used by the objective function); and setting the changed variable field to *unassigned* (later to be used by the move-operator).

Like the representation used in the STLEA in [3], the actual set of violated constraints is used, instead of the derivative *number* of violated constraints commonly used. This reduces the number of conflict checks needed by the objective function to determine the fitness of the individual. In exchange for this, more care has to be taken while maintaining this set during the run. Note that instead of actually storing the constraint itself, the index of the constraint in the set of constraints from the CSP is used. This index can be used to easily retrieve the actual constraint, reducing the memory needed were it to be stored in the individual itself.

The changed variable field is used by the objective function and set by the move-operator to quickly identify which variable has been changed, and consequently which relevant constraints need to be checked. Although limited here to a single variable, this mechanism can be extended in case more than one variable can be changed, although this is not necessary for the CTLEA.

3.2 Objective Function

The objective of the CTLEA is to minimise the number of violated constraints. A solution is found when a candidate solution violates no constraints. The objective function in the CTLEA then maintains the set of violated constraints of an individual. The number of conflict checks necessary for one fitness evaluation is reduced by only considering constraints relevant to the variable stored in the changed variable field. First, the constraints relevant to the changed variable are removed from the set of constraints stored by the individual. Then, all constraints in the CSP relevant to the changed variable are checked, and if violated by the candidate solution, added to the set of constraints stored by the individual. Eventually, the set of constraints stored by the individual will contain all constraints violated by the candidate solution stored by the individual.

For newly initialised individuals, all constraints are checked, and if violated by the candidate solution, added to the set of constraints stored by the individual.

The objective function of the CTLEA uses the tabu list by first checking if a conflict is in the tabu list before performing the conflict check on the CSP-instance. If the conflict is found to exist in the CSP-instance, it is added to the tabu list.

3.3 Conflict Tabu List

The CTLEA maintains a tabu list of conflicts implemented as a hash set, indexed over the constraint they are in. All conflicts found during the run of the CTLEA are added to the conflict tabu list. Only conflicts not already in the the tabu list are added, the list does not contain double entries.

The tabu list is used in only two ways, adding a conflict (insertion), and checking if a conflict is in the tabu list (look-up). Since there is no need to alter or remove a conflict once it has been added, both insertion and look-up can be done in constant time ($O(1)$) depending on the quality of the hash-function, and given adequate size of the hash table.

3.4 Move-Operator

The move-operator of the CTLEA takes a single individual (parent) to produce a single child (offspring). The basic premise of the move-operator is simple: select a variable to change, and change it in such a way that a child with fewer violated constraints is created. As such, there are two choices to be made: which variable to change; and what value to change the variable to.

The move-operator selects which variable to change by selecting one uniform randomly from a multi-set of variables created in the following way. First, all variables relevant to constraints in the set of constraints stored by the individual are added. Then, all variables transitively dependent to the variables already in the multi-set are added. A variable is *transitive dependent* to another variable, if it is relevant to a constraint which the other variable is relevant to. Take, for example, constraint c_1, with its two relevant variables x_1 and x_2. If there is another constraint c_2, with relevant variables x_1 and x_3, then x_3 is transitive dependent to x_1 and c_1. A multi-set is used so that variables that are relevant or transitive dependent to more than one constraint in the set stored by the individual have a higher probability of being selected.

Value selection follows the same idea as variable selection, in that a value is uniform randomly chosen from a set of values. The set of values is created by checking for each value in the domain if it violates a relevant constraint. If it does not, it is added to the set.

The move-operator uses the tabu list by first checking the tabu list if a value is tabu, before checking the CSP-instance. If a value violates a relevant constraint, the conflict is added to the tabu list as well. A simple example of this use of the tabu list goes as follows. Supposed variable x_1 is selected for change. A random value for x_1 is now chosen from the set of values V_1, say v_3. The tabu list is now used to check if this value is in the tabu list. Since the tabu-list stored value-pairs, all variables relevant through a constraint have are now selected. Say, only variable x_5 is relevant to x_1, and in the current individual it has value v_7. The tabu list is now checked for the occurance of value pair: $(\langle x_1, v_3\rangle, \langle v_5, v_7\rangle)$. If the value pair is on the tabu list, another value is selected for v_1, if not the move-operator ends. If in the object operator the tried value pair turns out to be a conflict, it is added to the tabu list.

The move-operator iteratively selects different variables and tries to select values to them. No variable is selected twice, and a new variable is selected only when no value can be found that does not violate its relevant constraints. Selecting a variable twice is prevented by not adding a variable to the variable multi-set if it has been selected earlier.

It is possible that all variables relevant, or transitively dependent have been selected by the move-operator already. At this point, the remaining variables are selected uniform randomly. If all possible variables have been selected, a new individual is initialised and inserted in the offspring population. At this point, the move-operator acts as a gradual restart strategy, by starting a new, randomly chosen, search-path for the CTLEA to explore.

4 Experimental Setup

The test-set introduced in [2] was used for experimentation with CTLEA (see section 2). Success rate (*SR*), and the average number of conflict checks to solution ($ACCS$), are used to measure the performance of the algorithm.

The *SR* measure is used to measure the effectiveness of an algorithm, and is calculated by dividing the number of successful runs performed by the algorithm by the total number of runs performed. A successful run is a run in which the algorithm solves the CSP-instance. Usually given as a real number between 0.0 and 1.0, the *SR* can also be expressed as a percentage. A *SR* of 1.0 or 100%, or perfect *SR*, means all runs solved their CSP-instance. Since the algorithm's primary task is to solve CSP-instances, the *SR* is perceived as the most important performance measure to compare algorithms on. Accuracy of the *SR* measure is affected by the total number of runs.

The $ACCS$ measure is used to measure the efficiency of our algorithm, and is calculated by averaging the number of conflict checks needed by an algorithm over several successful runs. A conflict check is defined as the check made to see if a conflict is in a constraint. Note that the $ACCS$ measure includes all conflict checks made by the algorithm, in the case of the CTLEA, this does also include those made in the move-operator. Conflict checks made during unsuccessful runs of an algorithm are discarded, and if all considered runs of an algorithm are unsuccessful, the $ACCS$ measure is undefined. Used as a secondary performance measure for comparing algorithms, the accuracy is $ACCS$ affected by the number of successful runs and the total number of runs of an algorithm (the ratio of which is the *SR* measure), i.e., $ACCS$ is more accurate when *SR* is higher.

Efficiency performance measures have to take into account the computation effort expended by an algorithm. The $ACCS$ uses the number of conflict checks as the atomic measure to quantify the expended computational effort however. The CTLEA also expends computational effort on maintaining the tabu list (see section 3). While comparing the effort spent on performing conflict checks and maintaining the tabu list, it was found that the latter was negligible in comparison to the former when the CSP-instance was sufficiently hard to solve. Given that the CSP-instances used in the test-set are all taken from the mushy region in the density-tightness parameter space (see section 2), complexity of the CSP-instances is sufficient to regard the computation effort of maintaining the tabu list as negligible compared to that of performing the conflict checks.

The CTLEA is blessed with relatively few parameters to fine-tune: the population size (*popsize*); the maximum number of conflict checks allowed (*maxCC*); the *bias* of the biased linear ranking parent selection operator; and the size of the parent population. Although it is possible to vary the size of the parent population, as in [3], we keep it equal to the number of individuals in the population (*popsize*), with no noticeable effects on performance in preliminary experiments. From [3], as well as other studies ([1,2]), we took 1.5 as bias for the biased linear ranking selection operator. This leaves us with just two parameters to fine-tune: *popsize*, and *maxCC*.

Although in [1] and [2] small population sizes were advocated, extensive experimentation in [3] shows that larger populations were more appropriate, mostly because of the beneficial effects on the population diversity. There is a trade-off to consider though. With small populations, more computational effort can be spend on increasing the fitness of the individuals over more generations. Although a relatively small number of search-paths can be followed in parallel, they can be followed to more depth. The drawback is that small populations have the tendency to lose population diversity, thus increasing the risk of getting the algorithm stuck in a local optimum from which it can not escape. On the other hand, larger populations allow for more search-paths to be followed in parallel, but to a lesser depth, while maintaining a higher population diversity. It is not possible to predict where in the *popsize*-*maxCC* parameter space the optimum parameter setting lies, and as such, we experimented with a large number of parameter combinations to find it. This also allows us to identify the optimum parameter settings for each density-tightness combination, in case this differs.

The experimental setup of the CTLEA is then as follows: for each CSP-instance in the test-set (of which there are 225), we run the algorithm 10 times. Varying combinations of population size (*popsize*) and maximum number of conflict checks allowed (*maxCC*) are used. The *popsize* parameter is taken from the following set: $\{10\} \cup \{50, 100, 150, \ldots, 2000\}$ (41 elements). The *maxCC* parameter is taken from the following set: $\{100000, 200000, \ldots, 2000000\}$ (20 elements). In total $225 \times 10 \times 41 \times 20 = 1,845,000$ runs were performed.

5 Results

Figure 1 summarises the results of the experiments described in the previous section. It consists of 9 graphs, each showing the result for one density-tightness combination in the test-set. The top row of graphs show 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 of *maxCC* on the *SR* for different values of *popsize*. Along the x-axis of each graph in

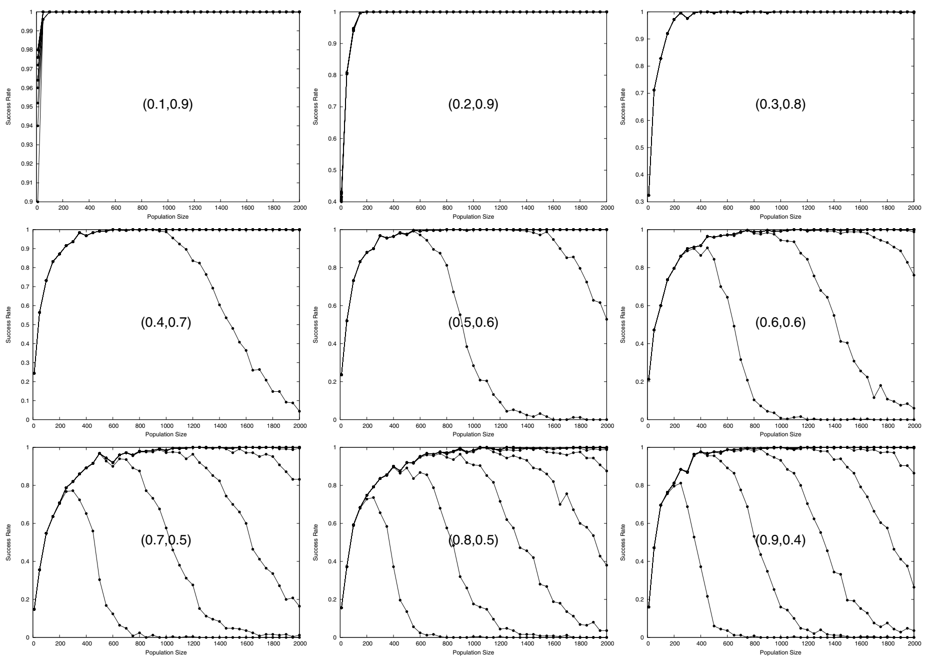

Fig. 1. The relationship between the population size (x-axis) and the success rate (y-axis) of the CTLEA for different maximum number of conflict checks allowed

figure 1 the *popsize* is shown, the y-axis shows the SR, while each curve in the graph was found for different values of $maxCC$.

The same trend is noticeable for SR in all graphs: the SR first increases when larger values for *popsize* and $maxCC$ are used, but drops off sharply when the *popsize* gets too large relative to the available $maxCC$. At the point where the CTLEA solves all CSP-instances ($SR = 1.0$), just enough $maxCC$ is available for the *popsize* but not more. Beyond this point, SR decreases for increased *popsize* but equal $maxCC$. Each curve therefore describes an arc with increasing SR for larger values of *popsize*, until *popsize* is increased to the maximum value able to be successfully maintained by the available $maxCC$, after which SR decreases again. Differences between the different graphs in figure 1 can partially be explained by differences in complexity between the different density-tightness combinations. CSP-instances in density-tightness combination 1, for example, are known to be easier to solve than those in density-tightness combination 9, and the number of conflict checks needed to sustain the population while reaching a perfect SR reflect that.

Table 1 shows, for each density-tightness combination, the first parameter combination for reaching a perfect SR, *popsize* minimised before $maxCC$, as well as the $ACCS$ used to find the solutions. Note the increasing size of *popsize* and $maxCC$ needed for reaching a perfect SR for the different density-tightness combinations. Because the CSP-instances for the different density-tightness combinations

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.

	SR	**ACCS**	**popsize**	**maxCC**
1	1.0	1313	100	100000
2	1.0	4670	200	100000
3	1.0	20283	400	100000
4	1.0	50745	600	100000
5	1.0	94931	800	200000
6	1.0	167627	1100	300000
7	1.0	239106	1200	400000
8	1.0	254902	1050	700000
9	1.0	240046	950	500000

Table 2. Comparing the success rate and average conflict checks to solution of the CTLEA, the STLEA, Hill-climbing algorithm with Restart (HCAWR), Chronological Backtracking Algorithm (CBA), and Forward Checking with Conflict-Directed Backjumping Algorithm (FCCDBA)

	CTLEA		**STLEA**		**HCAWR**		**CBA**		**FCCDBA**	
	SR	**ACCS**	**SR**	**ACCS**	**SR**	**ACCS**	**SR**	**ACCS**	**SR**	**ACCS**
1	1.0	1313	1.0	2576	1.0	234242	1.0	3800605	1.0	930
2	1.0	4670	1.0	67443	1.0	1267015	1.0	335166	1.0	3913
3	1.0	20283	1.0	313431	1.0	2087947	1.0	33117	1.0	2186
4	1.0	50745	1.0	397636	1.0	2260634	1.0	42559	1.0	4772
5	1.0	94931	1.0	319212	1.0	2237419	1.0	23625	1.0	3503
6	1.0	167627	1.0	469876	1.0	2741567	1.0	44615	1.0	5287
7	1.0	239106	1.0	692888	1.0	3640630	1.0	35607	1.0	4822
8	1.0	254902	1.0	774929	1.0	2722763	1.0	28895	1.0	5121
9	1.0	240046	1.0	442323	1.0	2465975	1.0	15248	1.0	3439

(perhaps with the exception of density-tightness combination 1) were selected to minimise complexity variance, the increasing *popsize* and *maxCC* needed to solve the higher density-tightness combinations thus seems to reflect an aptitude of the algorithm to solve CSP-instances with a lower tightness, i.e., fewer average conflicts per constraint.

Table 2 shows a comparison of the performance of the CTLEA with the STLEA from [3], and benchmark algorithms from [2]. Table 2 shows that the CTLEA outperforms STLEA on all CSP-instances with density-tightness combinations. As the STLEA, the CTLEA compares favourably with the Hill-climbing with Restart Algorithm (HCAWR), with efficiency measured in *ACCS* several magnitudes better. Compared with the Chronological Backtracking Algorithm (CBA), the CTLEA outperforms it on CSP-instances with density-tightness combinations 1, 2, and 3, but is outperformed on all others. This shows that in the area where CTLEA has shown good performance, CSP-instances with lower tightness, it can outperform a classical algorithm. Compared with the more sophisticated Forward Checking with Conflict-Directed Back-jumping Algorithm (FCCDBA) (a combination of forward-checking [13] and conflict-directed back-jumping [14]) however, the CTLEA can only approach performance on CSP-instances the first two density-tightness combinations, but is outperformed in all others. Overall, the CTLEA raised the performance bar a little higher for EAs, but remains unable to beat sophisticated deterministic algorithms on efficiency.

6 Conclusions

This paper introduced the Conflict Tabu Search Evolutionary Algorithm (CTLEA) for solving binary constraint satisfaction problems. The CTLEA is a hybrid algorithm, incorporating elements of an evolutionary and the tabu search meta-heuristic. The CTLEA is based on the Simple Tabu Search Evolutionary Algorithm (STLEA), introduced in [3], substituting its compound label tabu list with a tabu list limiting the search space by storing of conflicts. The rational behind choosing conflicts for the CTLEA tabu list is the comparatively limited number of conflicts and their usefulness in the new move-operator. Like the STLEA, the CTLEA maintains the basic structure of an evolutionary algorithm, but merges the crossover and mutation operator in one 'move-operator'. Further efficiency improvements were achieved by using the same representation as was used in the STLEA.

A large number of parameter tuning experiments were performed for different density-tightness combinations of a commonly used test-set. The performance of the CTLEA with the best parameter settings was found to outperform the STLEA, making it the best performing EA for solving the binary CSP found thus far.

Although comparable in performance to the Chronological Backtracking Algorithm on CSP-instances with lower tightness, the CTLEA continues to be outperformed by the more sophisticated Forward Checking with Conflict Directed Back-jumping Algorithm.

Future research will focus on comparing the relative behaviour of the CTLEA to other algorithms when size of the CSP-instances is increased and the effects of using different types of tabu lists on performance.

References

1. Craenen, B., Eiben, A., van Hemert, J.: Comparing evolutionary algorithms on binary constraint satisfaction problems. IEEE Transactions on Evolutionary Computing 7(5), 424–445 (2003)
2. Craenen, B.: Solving Constraint Satisfaction Problems with Evolutionary Algorithms. Doctoral dissertation, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands (November 2005)

3. Craenen, B., Paechter, B.: A tabu search evolutionary algorithm for solving constraint satisfaction problems. In: Runarsson, T.P., et al. (eds.) Parallel Problem Solving from Nature – PPSN IX. Lecture Notes on Computer Science, vol. 4192, pp. 152–161. Springer, Berlin (2006)
4. Glover, F., Laguna, M.: Tabu search. In: Reeves, C. (ed.) Modern Heuristic Techniques for Combinatorial Problems, pp. 70–141. Blackwell Scientific Publishing, Oxford, England (1993)
5. 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)
6. 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. Applications IV, vol. 1, pp. 187–194 (1998)
7. 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, pp. 213–229. Kluwer Academic Publishers, Boston, MA (1998)
8. Kratica, J.: Improving performances of the genetic algorithm by caching. Computer and Artificial Intelligence 18(3), 271–283 (1999)
9. 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 1990), Stockholm, Pitman, pp. 550–556 (1990)
10. Tsang, E.: Foundations of Constraint Satisfaction. Academic Press, London (1993)
11. MacIntyre, E., Prosser, P., Smith, B., Walsh, T.: Random constraint satisfaction: theory meets practice. In: Maher, M.J., Puget, J.-F. (eds.) CP 1998. LNCS, vol. 1520, pp. 325–339. Springer, Heidelberg (1998)
12. 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, pp. 116–123. Morgan Kaufmann Publisher, Inc., San Francisco (1989)
13. Haralick, R., Elliot, G.: Increasing tree search efficiency for constraint-satisfaction problems. Artificial Intelligence 14(3), 263–313 (1980)
14. Prosser, P.: Hybrid algorithms for the constraint satisfaction problem. Computational Intelligence 9(3), 268–299 (1993)