

# Hybrid Search for Dynamically Changing CSPs

Roie Zivan<sup>(3)</sup>, Uri Shapen<sup>(1)</sup>, Amnon Meisels<sup>(1)</sup>, and Meir Kalech<sup>(2)</sup>

<sup>(1)</sup>Department of Computer Science

<sup>(2)</sup>Department of Information System Engineering

<sup>(3)</sup>Department of Industrial Engineering and Management

Ben-Gurion University of the Negev

Beer-Sheva, 84105, Israel

<sup>(1)</sup>{shapenko, am}@cs.bgu.ac.il

<sup>(2)</sup>kalechm@bgu.ac.il

<sup>(3)</sup>zivanr@bgu.ac.il

**Abstract.** Constraints satisfaction problems are often used for modeling and solving problems in an environment which is subject to changes. When some of the constraints are changed after a solution was obtained it is often desired to search for a solution to the derived problem where the solution to the changed problem is as similar as possible to the previous (e.g., current) solution. This requirement is relevant for many timetabling and scheduling scenarios.

Previous approaches to this problem include the method of Roos et. al. [5] which traverses the possible assignments according to their distance from the original solution, starting from the smallest distance. In a more recent paper by Hebrard et. al. [1] the problem of finding the most similar solution was shown to be NP-hard and a method based on a Branch and Bound scheme was proposed. This method enforces global arc-consistency on a global soft constraint of similarity.

The present paper proposes a new approach for finding the most similar solution to a changing constraints satisfaction problem. The proposed method exploits the fact that the goal is to find a complete assignment with two different properties. The first is that it contains as many as possible identical assignments to the previous solution. The second is that the assignment must be consistent, i.e. it must be a satisfying solution to the new CSP. According to these two different requirements our proposed solution method interleaves constraint optimization and constraint satisfaction techniques in order to find the most similar solution to the changed CSP. Preliminary experiments on random CSPs present the advantage of the proposed method over previous approaches.

## 1 Introduction

Constraints satisfaction is an elegant model for representing many real world combinatorial problems. In many realistic scenarios some of the constraints of a constraint satisfaction problem (CSP) might change after a solution was obtained. Some examples are a school timetable where a teacher might call in sick, a schedule of tasks in an industrial environment where a machine can break down or a new urgent order might arrive, etc. In many of these scenarios it is not enough just to find any other solution which satisfies the constraints of the newly derived problem. In the school example, teachers and students have their schedule and plan their actions according to it. Thus, it

is desired to find a new solution which is as similar as possible to the solution that was obtained before the problem changed.

[8, 10] study the problem of acquiring solutions for Dynamic CSPs. They define a Dynamic CSP as a series of CSPs which defer one from the other in some of their constraints (either added constraints, removed constraints or both). The main focus of these studies is to find a solution to each CSP in the series as fast as possible. Thus, a number of methods for acquiring a new solution to a changed CSP were proposed. One of the proposed methods reuses inconsistent assignments (*Nogoods*) which were found while solving the original CSP, when searching for a solution to the revised CSP. Another method uses dynamic backtracking and local search techniques starting with the solution which was obtained for the original problem. However, none of the proposed methods is guaranteed to find the *most similar solution* to the previous solution.

In [5] a method was proposed which given a revised CSP and a solution to the original problem, finds a solution to the new CSP which is the most similar to the previous solution. The proposed method traverses the assignments of the CSP according to their distance from the former solution and enforces arc consistency after each change. Roos et. al. found that this method is feasible only for unary constraints and that it causes a significant slow down in comparison with an algorithm which searches for any solution to the new problem (our experimental study validates the inefficiency of this method). In a later study they offer approximation methods which are apparently feasible but do not guarantee to find the most similar solution [6].

Hebrard et. al. study different aspects of finding similar and diverse solutions to a CSP. Their goals represent different motivations, like the elicitation of preferences, interactive constraint definitions, and the stability of solutions in a dynamic environment. They differentiate between an off-line problem in which a set of similar (or diverse) solutions is required and an online problem in which given a CSP and a set of assignments, the goal is to find a solution to the CSP which is as close to (or as distant from) the given set of assignments. In [1] the authors prove that finding a solution to a CSP which is as close to (or as distant from) a given set of assignments is *NP-hard* (actually they prove it is  $FP^{NP[\log n]}$  - *complete*). They propose an algorithm for finding the closest solution to a given set of assignments. The proposed algorithm is based on a Branch and Bound scheme and enforces global arc consistency (*GAC*) on a soft global distance constraint in each step of the algorithm. In [2] the authors propose an algebra that enables a combination of similarity constraints to a set of ideal partial assignments and distance constraints from non-ideal partial assignments.

The present paper focuses on scenarios where a solution to a revised CSP is required which is as similar to a given solution to the former problem. To this end we propose an algorithm which (like the methods presented in [5, 1]) given a changed CSP and a solution to the previous CSP guarantees to find a solution to the derived CSP which is the closest (most similar) to the solution of the former CSP. Our proposed method exploits the fact that its reported solution must satisfy two requirements. First, that it is as close as possible to the solution of the former CSP and second, that it is a solution to the newly generated CSP. A first difference between these two requirements is that the first is an optimization requirement and the second is a satisfaction requirement. The second (and most important) difference is that the search space which needs to be

scanned in order to fulfill the first requirement includes only the value assignments of the former solution while the search space which needs to be scanned in order to fulfill the satisfiability requirement includes the entire CSP.

In formerly proposed solutions to the above problem [5, 1] the algorithms attempt to fulfill both requirements in a single phase. We propose a hybrid search method which exploits the very different properties of the two requirements of the problem. Our proposed method searches the problem in two phases. In the first, a Branch and Bound (*B&B*) algorithm is used to find a consistent partial assignment which includes only value assignments of the former solution. In this phase only constraints between the assignments of the former solution are being checked. An admissible heuristic which counts possible conflicts between these value assignments generates a significant speedup. When a partial assignment which includes only value assignments of the former solution is found by the first phase of the algorithm, a second phase is performed to validate that this partial assignment can be extended to a solution to the new CSP. In this phase all values of unassigned variables must be taken into consideration. In case it is a solution the distance between the solution found and the solution to the former CSP is updated as the new upper bound and the solution is stored as the most similar which was found by the algorithm. Then the algorithm resumes with the first phase of search for an assignment which contains a larger number of value assignments which were included in the original solution once again.

Our experimental study shows a clear advantage of the approaches which follow a *B&B* scheme over the systematic traversal of the search space proposed by [5] except for the cases in which there exists a solution with a very small difference. Furthermore, the proposed hybrid search outperforms the *B&B* with *GAC* enforcing version of [1] by a large factor.

## 2 Constraint Satisfaction Problems

A *Constraint Satisfaction Problem (CSP)* is composed of a set of  $n$  variables  $V_1, V_2, \dots, V_n$ . Each variable can be assigned a single value from a discrete finite domain. Constraints or **relations**  $R$  are subsets of the Cartesian product of the domains of constrained variables. For a set of constrained variables  $V_{i_k}, V_{j_1}, \dots, V_{m_n}$ , with domains of values for each variable  $D_{i_k}, D_{j_1}, \dots, D_{m_n}$ , the constraint is defined as  $R \subseteq D_{i_k} \times D_{j_1} \times \dots \times D_{m_n}$ . A binary constraint  $R_{ij}$  between any two variables  $V_j$  and  $V_i$  is a subset of the Cartesian product of their domains;  $R_{ij} \subseteq D_j \times D_i$ .

An assignment (or a label) is a pair  $\langle var, val \rangle$ , where  $var$  is a variable and  $val$  is a value from  $var$ 's domain that is assigned to it. A *partial solution* is a consistent set of assignments of values to a set of variables. A **solution** to a *CSP* is a partial solution that includes all variables.

For simplicity of presentation we consider CSPs which include only binary constraints. The distance function between two solutions is the *Hamming distance* (the number of variables that are not assigned the same value in both solutions).

### 3 Finding a solution to a changed CSP

The problem we seek to solve in this paper is defined as follows: Given a constraints satisfaction problem  $CSP_{origin}$  a solution to  $CSP_{origin}$ ,  $S_{origin}$ , and a constraint satisfaction problem  $CSP_{changed}$  where  $CSP_{changed}$  is different from  $CSP_{origin}$  only in the set of constraints  $R$ , we are looking for the solution to  $CSP_{changed}$  which has the most value assignments that are included in  $S_{origin}$ . In the rest of the paper we will refer to the value assignments of the former solution  $S_{origin}$  as *Solution Value Assignments* (SVAs).

The proposed hybrid search algorithm for dynamic CSPs (*HS\_DynCSP*) includes two phases which are interleaved throughout the search. In the first phase the algorithm assigns SVAs to variables attempting to generate a partial assignment which includes only variables which have already assigned their SVAs. This phase of the algorithm implements a branch and bound scheme where the upper bound is the smallest Hamming distance among the solutions to  $CSP_{changed}$  which were found so far by the algorithm. If the algorithm detects that the partial assignment cannot be extended to a solution with a smaller distance from  $S_{origin}$  than the upper bound, it backtracks. In order to detect the need to backtrack as early as possible, the algorithm uses an admissible heuristic function described in Section 3.1. If no more SVAs can be assigned to any of the unassigned variables and the number of the unassigned variables is smaller than the upper bound, the second phase of the algorithm is performed. In the second phase an arc-consistency maintenance (*MAC*) algorithm is performed in order to validate that the partial assignment of SVAs which was found can be extended to a solution. If the second phase ends successfully and a solution is found, then it is recorded as the best solution so far and the upper bound is set to its distance from  $S_{origin}$ . The other option is that the satisfaction algorithm finds that the partial assignment of SVAs cannot be extended to a solution to  $CPA_{changed}$ . In both cases after this phase is completed the algorithm resumes the first phase by removing the last SVA assignment (e.g., backtracking).

Figures 1 and 2 present the code of the proposed Hybrid algorithm for Dynamic CSPs (*Hyb\_DynCSP*). The main loop of the algorithm calls the first phase in order to find an assignment of SVAs which is longer than the number of SVAs found in the best solution stored by the algorithm (i.e., that the number of variables which are not in this current assignment is smaller than the upper bound). When such an assignment is found, the second phase is called in order to validate that the assignment can be extended to a solution to the new CSP ( $CSP_{changed}$ ).

Function **phase1** tries to assign an SVA by calling function **assign\_SVA**. After each successful assignment of an SVA, the function checks that the current assignment can still lead to a solution with a distance from the former solution  $S_{origin}$  which is smaller than the upper bound (line 9). If not, the function backtracks. When there are no more SVAs to assign, the function **assign\_SVA** will return false and a check will be made if the difference between the length of the obtained assignment of SVAs and the number of variables in the system is smaller than the upper bound. This means that if it would be extended to a solution this solution will be nearer to  $S_{origin}$  than the best solution found so far. Thus, the function returns true. Otherwise, the function backtracks. When the function fails to assign an SVA and the current assignment is empty it means that

```

HS_DynCSP:
1. upper_bound  $\leftarrow n$ 
2. solution  $\leftarrow \text{null}$ 
3. current_assignment  $\leftarrow \phi$ 
4. unassigned_variables  $\leftarrow CSP_{\text{changed.variables}}$ 
5. while (phase1)
6.   phase2
7.   return solution

phase1:
8. while(assign_SVA)
9.   if ( $n - (\|current\_assignment\| + \text{heuristic}) \geq upper\_bound$ )
10.    backtrack
11.  if ( $n - \|current\_assignment\| < upper\_bound$ )
12.    return true
13.  else if ( $\|current\_assignment\| > 0$ )
14.    backtrack
15.  else
16.    return false

phase2:
17. current_solution  $\leftarrow \mathbf{AC}(current\_assignment)$ 
18. if(current_solution)
19.   upper_bound  $\leftarrow n - \|current\_assignment\|$ 
20.   solution  $\leftarrow current\_solution$ 
21. backtrack

```

**Fig. 1.** Main part of the Hybrid Dynamic CSP algorithm

all combinations of SVAs were explored and the algorithm terminates after the function returns false.

In function **phase2** a standard arc consistency maintenance (MAC) algorithm is used to check whether the current assignment which was found in **phase1** can be extended to a solution (line 17). If it does, this solution is stored and its distance from  $s_i$  is saved as the new upper bound. If not, the function backtracks (lines 18 - 21).

Function **backtrack** removes the SVA assignment that was performed last. All SVAs which were removed from the domains of unassigned variables when the last assignment was performed are returned back to their domains (lines 23,25).

Function **assign\_SVA** makes an attempt to assign an unassigned variable with its SVA (find an unassigned variable whose SVA is still in its domain). In case it succeeds, the function removes all conflicting SVAs from the domains of all unassigned variables (lines 27-29). The function returns false if none of the unassigned variables has an SVA in its domain.

```

backtrack:
22. current_assignment.last.remove_SVA
23. for each val ∈ {removed conflicting SVAs}
24.   val.move_back_to_domain
25. unassigned_variables.add(current_assignment.last)
26. current_assignment.remove_Last

assign_SVA:
27. var ← select_SVA_var
28. if (var)
29.   for each v ∈ unassigned_variables
30.     if (v.SVA ∈ v.domain and
           v.SVA conflicts with var.SVA)
31.       remove_from_domain(v, SVA)
32.   current_assignment.add(var)
33.   unassigned_variables.remove(var)
34.   current_assignment.remove_Last
35.   return true
36. else
37.   return false

```

**Fig. 2.** Functions used by the Hybrid Dynamic CSP algorithm

### 3.1 An admissible heuristic for the first phase

The first phase of the *Hyb\_DynCSP* algorithm is an optimization algorithm that uses a branch and bound scheme. As commonly used in optimization search methods, in order to speed up the algorithm one can use an admissible heuristic which will increase the lower bound of the algorithm [7]. As a result the algorithm will detect early a need to backtrack.

In the first phase the algorithm is searching for a partial solution of SVAs which is longer than the number of SVAs in the best solution found so far. The upper bound is the number of non SVA assignments in the best solution found. Thus, a first step in order to check whether the current assignment can be extended to a partial assignment of SVAs of the required length would be to check whether there are enough SVAs left in the domains of unassigned variables. More specifically, if we denote by  $n$  the total number of variables, by  $k$  the length of the current assignment and by  $s$  the number of unassigned variables that include an SVA in their domain then the current assignment can be extended to a solution which is more similar to  $S_{origin}$  only if  $n - (k + s) > upper\_bound$ . A better lower bound can be achieved if we take into consideration that every pair of conflicting SVAs cannot be assigned in the same solution. Thus, instead of adding both of them to the length of the current assignment only one of each such pair is added. More formally, if among the  $s$  unassigned variables which include SVAs in their domains there are  $s_{pairs}$  distinct pairs whose SVAs conflict. If in addition there are other  $s_{singles}$  variables which are not included in pairs and do not conflict among themselves, then if  $s_{pairs} > 0$  then  $n - (k + s) < n - (k + s_{pairs} + s_{singles})$ . By

**heuristic:**

1.  $count \leftarrow 0$
2.  $temp\_set \leftarrow unassigned\_variables$
3. **for each**  $v \in temp\_set$
4.   **if**  $(SVA \notin v.domain)$
5.      $temp\_set \leftarrow temp\_set \setminus v$
6.   **else if**  $(\exists v_1 \in temp\_set, v_1.SVA \text{ conflicts with } v.SVA)$
7.      $count ++$
8.      $temp\_set \setminus \{v, v_1\}$
9.   **else**
10.     $temp\_set \leftarrow temp\_set \setminus v$
11.     $count ++$
12. **return**  $count$

**Fig. 3.** An admissible heuristic for Hyb\_DynCSP

using the above admissible heuristic one increases the probability of detecting a need to backtrack earlier. This idea can be extended to cliques of any size.

## 4 Correctness of Hyb\_DynCSP

In order to prove that the proposed algorithm is correct one needs to prove that it is sound (a solution reported is a solution to the new CSP) and complete (the solution reported is the most similar solution to the former solution). Soundness is immediate. In phase one, since after each SVA assignment all conflicting SVAs are removed from the domains of unassigned variables (lines 29 - 31 in Figure 1), two conflicting SVAs cannot be assigned in this phase. The soundness of the second phase is dependent upon the soundness of the standard *MAC* algorithm.

In order to prove that the algorithm is complete we need to prove that the solution reported for the changed CSP, that is the most similar solution to the former solution. To this end we first prove the following Lemma:

**Lemma 1** *In phase 1 the algorithm backtracks only when the current assignment cannot be extended to a solution with more SVA assignments than in the best solution found so far (in other words, the heuristic used is admissible).*

**proof:** The upper bound represents the number of variables in the most similar solution found so far which are not assigned an SVA. There are only two places in phase one where a backtrack is performed. The second is simple to justify since there are no more SVAs in the domains of unassigned variables and therefore if the current assignment does not contain more SVAs than the most similar solution found by the algorithm there is no point in checking whether it can be extended to a solution. The first backtrack occurs after an SVA was assigned and conflicting SVAs were removed from domains of unassigned variables. If the difference between the number of variables in the CSP and the number of SVAs in the current assignment of the algorithm and in the domains

of unassigned variables is not smaller than this upper bound then even if there exists a solution with all the remaining SVAs it would not contain more SVAs than that of the best solution found so far. In addition, none of the conflicting pairs of SVAs in the domains of unassigned variables can be both included in a solution to the new CSP. Thus, by counting each of these pairs as a single potential SVA assignment the heuristic maintains its admissibility.  $\square$

The completeness of the algorithm derives from Lemma 1. If phase 1 ends successfully, the partial assignment of SVAs it found includes more SVAs than the most similar solution found so far (lines 11, 12 in Figure 1). The correctness of the standard *MAC* algorithm insures that if this partial assignment can be extended to a solution it will be found and stored as the new most similar solution. From Lemma 1 we know that the algorithm backtracks only if the current assignment cannot be extended to a consistent partial assignment of SVAs which includes more SVAs than in the stored solution. Thus, all relevant partial assignments are explored.  $\square$

## 5 Experimental Evaluation

We present preliminary experiments on random CSPs which demonstrate the advantages of the proposed algorithm in comparison with existing algorithms for finding the most similar solution to a changed CSP (e.g., *Dynamic CSP*).

Random *CSPs* are parametrized by  $n$  variables,  $k$  values in each domain, a constraints density of  $p_1$  and a tightness  $p_2$  which are commonly used in experimental evaluations of CSP algorithms [9].

The common approach in evaluating the performance of *CSP* algorithms is to measure time in logical steps to eliminate implementation and technical parameters from affecting the results. Thus, in the results below the number of constraints checks are reported [4, 3].

Two sets of experiments were performed on random problems. Both were conducted on *CSPs* with 20 variables ( $n = 20$ ) and 10 values in the domain of each variable ( $k = 10$ ). The value of constraints density is  $p_1 = 0.3$ . The tightness value  $p_2$ , was varied between 0.1 and 0.9, in order to cover all ranges of problem difficulty. For each of the pairs of fixed density and tightness ( $p_1, p_2$ ), 50 different random problems were generated. After solving the problem by using a standard CSP algorithm, 0.1 percent of the constraints were changed and the Dynamic CSP algorithms were used to find the solution to the resulting problem. The solution which has the largest number of assignments that are similar to the solution which was found to the original problem. The results presented are an average of the 50 runs of each dynamic CSP algorithm.

Figure 4 presents the results of the Hybrid Search algorithm and the algorithm of Roos et. al. As the tightness of the problem increases, the closest solution is expected to be with a larger difference to the original one. It is clear that the performance of Roos's algorithm drastically decreases when the problems become tight, in comparison with the Hybrid algorithm. One way to validate the dependency of the distance of the optimal solution from  $S_{origin}$  and the performance of the algorithm is the following. Consider the average results of the algorithm for problems with the same difference between  $S_{origin}$  and the optimal solution. These are presented in Figure 5.

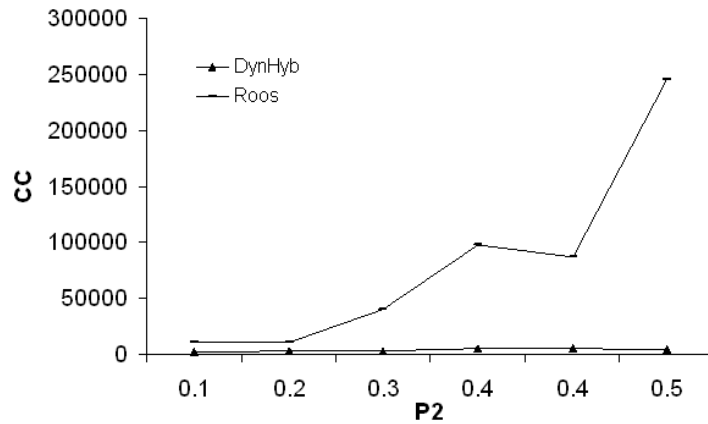


Fig. 4. Hybrid Search vs. Roos.

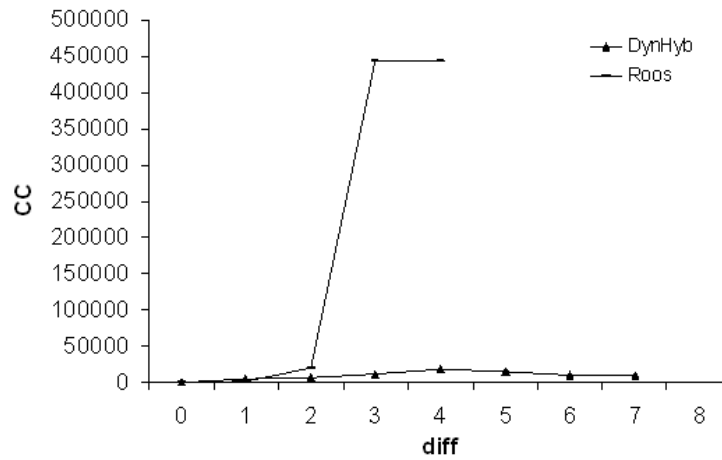
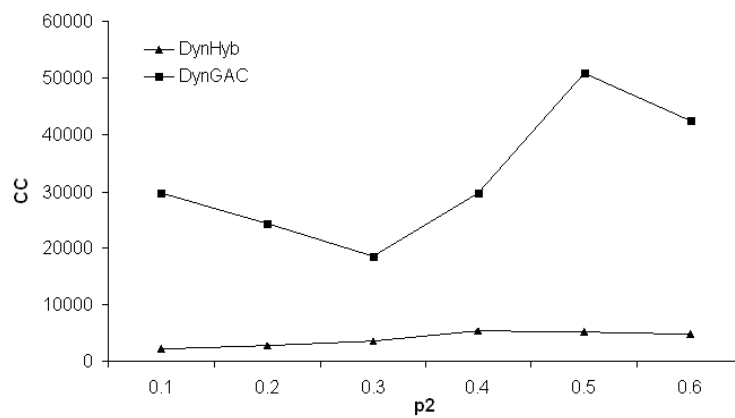


Fig. 5. Hybrid Search vs. Roos, solution distance.

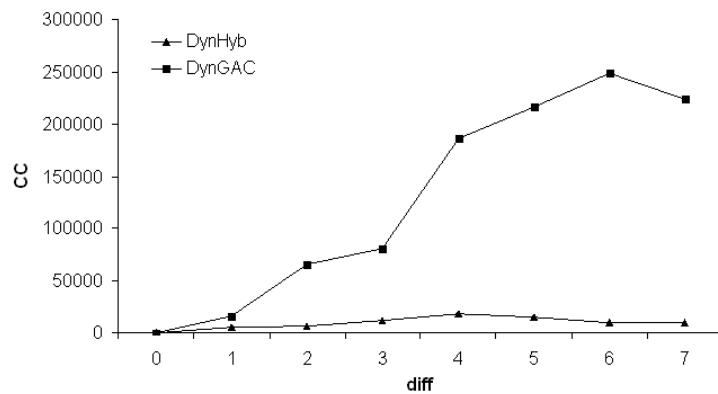
Figures 6 and 7 present the results of the Hybrid Search algorithm and the global arc consistency algorithm of Hebrard et. al [1]. It is obvious that the large advantage of the hybrid algorithm proposed in the present paper is independent of the constraints tightness or the difference between the original and the closest solution.

## 6 Discussion

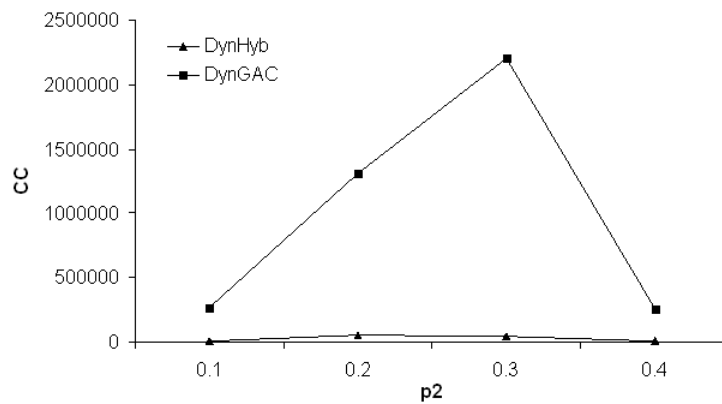
The presented experiments compare between three very different algorithms for finding the most similar solution to a changed CSP. The first, proposed by Roos et. al, searches the assignments by their distance from the original solution. Thus, the first solution to be found to the changed CSP is the optimal solution. This approach was found to be effective only when the tightness of the problem (and the resulting difference between the original and the new solution) is very small. The second algorithm is based on



**Fig. 6.** Hybrid Search vs. GAC.



**Fig. 7.** Hybrid Search vs. GAC., solution distance.



**Fig. 8.** Hybrid Search vs. GAC solving dense CSPs.

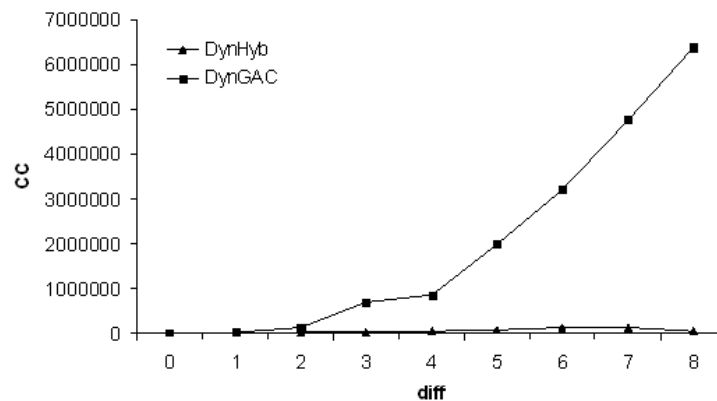


Fig. 9. Hybrid Search vs. GAC., solution distance, solving dense CSPs.

branch and bound and enforces global arc consistency on a similarity soft constraint. The branch and bound scheme algorithm is evidently much faster than the best first search of Roos et. al. However, forcing global arc consistency can be an overhead in terms of computation since the optimality of a new solution is determined only by the value assignments of the former solution (for the original CSP). The third algorithm, which is proposed in this paper, avoids the overhead in enforcing global arc consistency on the entire CSP. It separates the search to an optimization phase and a satisfaction phase. Thus, the optimization phase can include only one value for each variable (the value assignment in the solution for the original problem). The search space of the first phase (the optimization phase) in the proposed algorithm is defined by the number of consistent combinations of SVAs. The second phase which considers the entire domain of variables is performed only when a relevant partial solution is found by the first phase. It needs to consider only values in domains of unassigned variables. In addition, it solves a constraint satisfaction problem which is easier than a constraint optimization problem. This results in a significant improvement of performance.

## 7 Conclusion

A hybrid search algorithm for finding the most similar solution to a changed CSP was presented. The proposed algorithm performs optimization and satisfaction phases alternately. The optimization phase is performed on a much smaller search space. The satisfaction phase is used to determine if an assignment with a potential to be optimal leads to a solution to the changed problem. Our preliminary experiments show the high potential of this approach. In the future we intend to test this approach on realistic scheduling scenarios.

## References

- [1] E. Hebrard, B. Hnich, Barry O’Sullivan, and Toby Walsh. Finding diverse and similar solutions in constraint programming. In *AAAI-2005*, Pittsburgh, PA, USA, July 2005.

- [2] E. Hebrard, Barry O’Sullivan, and Toby Walsh. Distance constraints in constraint satisfaction. In *IJCAI-2007*, Hyderabad, India, January 2007.
- [3] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 21:365–387, 1997.
- [4] P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1996.
- [5] N. Roos, Y. Ran, and H. J. van den Herik. Combining local search and constraint propagation to find a minimal change solution for a dynamic csp. In *Artificial Intelligence: Methodology, Systems, Applications*, pages 272–282, 2000.
- [6] N. Roos, Y. Ran, and H. J. van den Herik. Approaches to find a near-minimal change solution for dynamic csps. In *Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems*, pages 373–387, 2002.
- [7] S. Russell and P. Norvig. *Artificial Intelligence, A Modern Approach, Second Edition*. Prentice Hall, 2005.
- [8] Thomas Schiex and Gérard Verfaillie. Nogood recording for static and dynamic constraint satisfaction problem. *International Journal on Artificial Intelligence Tools (IJAIT)*, 3(2):187–207, 1994.
- [9] B. M. Smith. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:155 – 181, 1996.
- [10] Gerard Verfaillie and Thomas Schiex. Solution reuse in dynamic constraint satisfaction problems. In *National Conference on Artificial Intelligence*, pages 307–312, 1994.