CSCI 4511W Project: Creating an Algorithm for Complex Dynamic CSPs

Omar Abdelfatah
abdel127@umn.edu

December 18, 2019

Abstract

Constraint satisfaction problems can vary from simple map coloring problems, to more complex problem that require intensive algorithms. There have been many algorithms and heuristics created for all of simple, complex, static, and dynamic CSPs. This paper will utilize those sources in order to tackle a complex version of a dynamic CSP. The problem solved here is of a DCSP whose set of variables and domain set, of possible values per variable, are the same. Additionally, each variable is required to have two values in order for the solution to be complete.
I may have attempted to tackle a problem bigger than myself as the complications of the shared variable and domain set seem to be requiring more attention than I had originally though. I suspect plenty of room for growth and improvement to this initial attempt at solving this problem. The experimental results of the presented algorithm show that, by utilizing the scholarly algorithms, it is possible to efficiently solve the complex CSP problem at hand. Hopefully this algorithm gets more generalized in the future to be applicable to other areas as well.

Introduction

Constraint satisfaction problem (CSP) refers to a problem that has a set of variables $X = \{X_1, ..., X_n\}$ whose value (picked from a set $D = \{D_1, ..., D_n\}$ of respective domain of values) must satisfy constraints and limitations found in the set C. CSPs can be represented by a constraint graph consisting of a node for each variable and a vertex connecting each pair of variables that are contained within a constraint. The most common CSPs are map coloring problems and the N-queens puzzle. Here I will be looking at a CSP whose variables are to be assigned a pair of values, which are a part of the variables. More specifically, the problem is inspired by my ballroom dance team which has 56 dancers (28 leaders/males, 28 followers/females), and the goal is to give each dancer two partners from the opposite group. Each dancer will be paired with exactly two dancers, and the complexity comes from the next two constraints: each dancer has their own preferences/constraints of who they want to dance with, and who they don't want to dance with; each dancer dances at a specific skill level, which is taken into account to not have a skill gap bigger than two levels in one partnership. A partnership is a two way relationship, meaning if $\{X_1\}$ is partnered with (has a value of) $\{X_2\}$, then $\{X_2\}$ is also partnered with $\{X_1\}$. With the knowledge of the nature of this problem, this paper investigates improvements to the standard CSP algorithm and any different versions of CSP algorithms that could be applicable to this problem.

As already mentioned, one of the more important things about this problem is that the set of variables is the same as the set of domains; this then opens the possibility of applying this to other complex CSPs which require multiple values per variable (e.g. project team assignment). As far as my research has gone, I was unable to find any related works that explored this concept of CSPs, and so it would be interesting to see where this goes.

Related Work

This section consists of short literature reviews on related work that has already been done. It will focus primarily on the algorithms that are applicable and related to the problem at hand.

Hybrid Algorithms for CSPs [5]

Tree-search algorithms try to construct a solution to a CSP by sequentially assigning values to the variables of the problem. [5] claimed that there are five basic tree search algorithms for CSP: naive backtracking (BT), backjumping (BJ), conflict-directed backjumping (CBJ), backmarking (BM), and forward checking (FC). It grouped BT, BJ, and CBJ as different styles of a backward move up the tree (backtracking) [5]. It also combined BT, BM, and FC as different styles of a forward move down the tree (assignment of variables) [5]. The paper presented an approach that allows these 5 basic algorithms to be combined, resulting in new hybrids [5]. Since it has described the base algorithms explicitly in terms of a forward move and a backward move, it then showed that the forward move of one algorithm may be combined with the backward move of another, resulting in a new hybrid algorithm [5]. In total, four hybrids are presented: backmarking with backjumping (BMJ), backmarking with conflict-directed backjumping (BM-CBJ), forward checking with backjumping (FC-BJ), and forward checking with conflict-directed backjumping (FC-CBJ) [5]. The performances of the nine algorithms (the original five plus the additional four hybrids) was compared over 450 instances of the same problem, resulting in data that presented FC-CBJ as by far the best of the algorithms examined [5].

Dynamic Variable Ordering [1]

The order in which a search algorithm instantiates the variables is known to have a potentially profound effect on its efficiency, and various heuristics have been investigated for choosing good orderings. Static variable ordering is when the order does not change between branches, and this is usually calculated once before the search algorithm begins. Dynamic variable ordering (DVO) is where the variables are instantiated during tree-search can vary from branch to branch in the search tree [1]. The paper studied DVO, in the context of binary CSP. Binary CSPs are ones whose set C only consists of constraints between two variables. The paper investigated the DVO technique commonly used in conjunction with tree-search algorithms for solving CSP [1]. The paper provided an implementation method for adding DVO to arbitrary tree-search algorithms (including those that maintain complicated information about the search history) [1]. It also investigated the most popular DVO reordering heuristic known as the minimum remaining values (MRV) [1]. MRV instantiates next the variable that has the fewest values compatible with the previous instantiations [1]. It also investigated the performance of 12 different algorithms with and without DVO, demonstrating that FC equipped with DVO is a very good algorithm for solving CSPs [1].

Best Constraint Satisfaction Search [3]

This reviewed paper presented the results of an empirical study of several CSP search algorithms and heuristics. The search algorithms studied were BT, BM, CBJ, and FC; those were matched with two variable ordering heuristics, min-width (MW) and DVO [3]. MW, a static variable ordering heuristic, orders the variables from last to first by repeatedly selecting a variable in the constraint graph that connects to the minimal number of variables that have not yet been selected [3]. Their selected DVO instantiates the variable with the smallest remaining domain size, after updating it based on already instantiated variables [3]. The paper used a random problem generator that created instances with given characteristics; it related the performance of the search methods with the number of variables, the tightness of the constraints (what fraction of the domain is disallowed by the constraint), and the sparseness of the constraint graph. CBJ with their implemented DVO version was shown to be extremely effective on a wide range of problems [3].

Heuristics for over-constrained CSPs [8]

The next paper proposed a heuristic search approach to solving large-scale CSPs [8]. The procedure and approach is simple to understand. Inconsistently assign all the variables some values, and then choose a variable that is in conflict (two variables are in conflict if their values violate a constraint [8]), and change its value in order to decrease the number of conflicts [8]. The search can be guided by a value-ordering heuristic (the min-conflicts heuristic), which attempts to minimize the number of constraint violations after each step [8]. Additionally, [8] showed that the heuristic can be used with a variety of different search methods. A theoretical analysis was presented in [8], which shows how this method works well on certain types of problems and also predicts when it is the most effective [8]. The simplicity of this approach allows it to be combined with other search methods and also allow it to be programmed efficiently [8].

Dynamic CSPs [4]

The next article reviewed here talked about CSPs whose sets (variables, domains, constraints) change dynamically in response to decisions made during the course of problem solving [4]. The article formalized this notion as a dynamic constraint satisfaction problem (DCSP) [4]. Then it described an implemented algorithm that that efficiently finds minimal (non-redundant) solutions to DCSP [4]. The utility of this approach is demonstrated for configuration and model composition tasks [4]. The paper also discussed the effect of changing which variables are needed in a solution and how to tackle that [4], but this is not relevant to our problem as our necessary variables do not change, only their constraints and domains.

Nogood Recording for CSPs and DCSPs [6]

The following article is focused on tackling the solution maintenance (finding a new solution to the constraint network after each modification) in DCSPs [6]. A new class of constraint recording called nogood recording is introduced in [6]. A nogood is a pair (α, J) where α is an assignment of values to a subset of the variables X, and $J \subset C$, such that no solution of the CSP (X, J) contains α [6]. The set J is called the nogood justification [6] (why α is an invalid solution). Nogood recording is able to solve regular CSPs and is also able to build an approximate representation of the solution space of DCSPs; the extra information collected through the nogood recording was shown to save a significant amount of time when solving CSPs[6] .

Solution Reuse in DCSPs [6]

Similar to [6], the following paper looked at an effective way to solve DCSPs [7]. In [7], they proposed the new method *local changes* (LC), which would reuse any old solutions to produce a new one. LC works as follows: let A be a consistent assignment V (a subset of X), and let $v \notin V$; a consistent assignment of $V \cup \{v\}$ can be achieved if and only if there exists a value val of v where we can assign val to v, remove all assignments (v', val') which are inconsistent with (v, val), and assign these variables again sequentially without modifying (v, val) [7]. It is also shown how a solution can be reached from no assignments, or from any previous assignments, and how it can be improved using filtering or learning methods (such as the nogood recording discussed in [6]) [7].

CSP Inconsistency Checking [2]

Finally, in [2] we will be looking at consistency checking in CSPs. [2] claims that there are two families of methods for CSP consistency checking: complete methods which make an exhaustive search on the solution space, and incomplete methods that make a local search on the solution space. The complete methods have the advantage to prove CSP inconsistency; however, their complexity grows exponentially when the problem size increases[2]. The incomplete methods have been used to find solutions for large size consistent CSPs that complete methods can not solve; however, they are unable to prove CSP inconsistency [2]. The paper attempted to provide an incomplete method that can deal with CSP inconsistency. It introduced a new incomplete method that is based on both a new notion of dominance between CSPs and a coloration of the CSP micro-structure [2]. The experimentation results of comparing the inconsistency detection rate of their method with other incomplete methods and one complete method showed that their method outperformed all the other incomplete methods, and its curve was superposed on the complete method, meaning that it succeeded to detect almost all of the inconsistent CSPs [2].

Application to Ballroom Problem

Considering that each dancer is to be assigned exactly two dancers, then every time a person is assigned as a value to a dancer, then their values are also updated. This makes the problem a DCSP, which makes [4], [6], and [7], very useful and applicable to our problem. Considering the tightness of this CSP, the experimental results from [3] in addition to methods mentioned in [5] would be useful search algorithms. These can of course be expanded on using static or dynamic variable ordering methods. The complexity of this problem is more than what meets the eye, as the direct relationship between the variables and values causes some obstacles.

Approach

A method of simplifying the complexity of this problem would be to separate the variables from their domains. This seems to be fairly achievable since leaders can only dance with followers and vice versa; all that needed to be done was choose a dance role as the variables and the other as the values, and assign accordingly. However, this would limit the usability and general applicability of the algorithm and solution to this problem; You would no longer be able to apply it to assigning classmates lab partners (assuming there is no gender segregation). For that reason, my approach was to keep both leaders and followers in the same set of variables as well as the same set of domains. After studying the aforementioned papers and algorithms, I had decided to spend the majority of my time implementing [5]'s FC-CBJ algorithm. Since that paper's data resulted in FC-CBJ's showing as most efficient, I had decided to begin my solution from there, with specific modifications that would allow it to work on my problem.

Additionally, [1] and [3] demonstrated how FC and CBJ, respectively, are effective when used with a DVO. This resulted in my software allowing for the FC-CBJ to be run using either a DVO or a static variable ordering. Comparison and results between the two will be shown later.

Modifications on [5]'s FC-CBJ

As the unmodified version ran and went through debugging phases, it was noted that some of the variables would be already solved when the algorithm attempts to label them. This was due to the variables being called as values twice before their need as a variable. This causes an incomplete solution to the CSP since to pair n dancers with two partners each, there needs to be n calls to the label function. To tackle this problem, a modification to the *bcssp* procedure introduced in [5] was made where line 12 would be replaced with these commands:

```
if skipped-dancers = 0 then
    status ← "Solutionfound";
else
    if not complete then
        while skipped-dancers $\neq$ 0 do
            bcssp-for-skipped();
        end
    else
        status = "Solution found";
    end
end
```

This piece of code is reached after the iterating value i is bigger than the amount of variables. Each time *label* attempts to pair a complete dancer, it will skipped them and the *skipped-dancer* variable is incremented by one. This procedure would then allow us to continue calling the *label* function until we have a complete solution. The following is the first few lines of the procedure for *bcssp-for-skipped*:

```
for dancer in team do ————— //where team is the set of variables
    if dancer is NOT complete then
        sub-team ← add(sub-team, dancer);
    end
endfor
skipped-dancers ← 0;
```

The remainder of that procedure is the complete and modified version of *bcssp*, with the exception to line 14 in [5]'s version, the status would be changed to "more backtracking required". This creates a subset of variables from the original set, to include only those who do not have two partners. It will then proceed call its own *subset-label* function that loops only through said subset, in order to fill those out. Resetting the skipped dancers counter allows it to track any new skips in the subset and then recursively call itself to create another subset of incomplete variables. The introduction of the new status of "more backtracking required" makes it so that when *bcssp-for-skipped* returns to *bcssp*, it does not have an "unsolvable" status simply because it could not complete the subset. The following code was added to the end of the while loop in *bcssp* (between lines 14 and 15 for [5]'s original version):

```
if status = "more backtracking required" then
    bcssp(start-pos, status, false);
end
```

start-pos is the index position of the first variable in *team*. *false* here the consistent flag of the procedure. Passing these arguments allows the next iteration of *bcssp* to start immediately at the incomplete variable and call *unlable* on it (due to the inconsistency flag) which would perform a backwards jump and update constraints. These presented pieces allow the algorithm to return a complete solution to the CSP. The introduction of tracking the amount of skipped dancers allows the fixing of unassigned partners, and the new status of "more backtracking required" results in the resuming of the conflict-directed backwards jumping.

Implementation of DVO

Through implementing the minimum remaining values DVO, the result was a similar algorithm to [7]'s local changes method. Mine is more simpler and can do less than the LC. The approach was to have an initial set of variables that will be used to build the new dynamic set. *dynamic-bcssp* is the same as the modified version explained above, except it uses *dynamic-label* and *dynamic-unlabel*. These functions perform the same as their static counterparts, with the exception of them iterating through the dynamic variable set as opposed to the initial static one, V. Before each call to *dynamic-label* a function (*build-next-dynamic-variable*) is called and removes the variable in V with the smallest domain size, and adds it to DV (the dynamic variables set). At the end of the whole process, V is empty and DV has all the dancers in it; this allows the iterator i to still be a valid tracker of the status and place of our search. Whenever *dynamic-unlabel* unlabels a set of variables, they are removed from DV, and returned back to V. Each dancer retains its original index value, which then allows a successful unlabelling to occur. There are also minor and trivial modifications made to *undo-reductions* and *check-forward* in order to allow them to traverse between the two sets DV and V.

Experiment Design and Results

In order to test the effectiveness between FC-CBJ with a DVO and without a DVO, 20 different scenarios were created using 3 sets of settings. Each scenario was ran 100 times, and the results were averaged out to calculate the number of consistency checks made (same parameter [5] used). The three variant settings were: V's original ordering type, usage of DVO, and tightness/difficulty of constraints. V's original ordering can be one of two things: a list of alternating leads and follows, or a list of all the leads followed by all the follows. The DVO can either be used or not used. Finally, the tightness of constraints is a measure of the binary constraints (both level-wise, and who each dancer specifically said they do not want), and this has 5 levels: trivial, easy, moderate, medium, and difficult. With increasing difficulty, there are more constraints per dancer (at maximum difficulty there is $D/2$ constraints), resulting in a smaller domain and also less correct solutions for the algorithm to find. The reason that V's original ordering matters and can affect both DVO and non-DVO, is due to the fact that both the variables and the domains are the same set. Meaning, even with DVO, that setting determines what order their domains are in. The upcoming results show the effect of V's original order on the effectiveness of DVO.
A list of this year's 50 (this semester's team was slightly smaller) dancers was created and modified to fit the criteria mentioned above. The only consistent aspect between the 20 different scenarios was each dancer's name and role. All other fields were varied as mentioned. The following table and graph show the amount of checks each scenario produced:

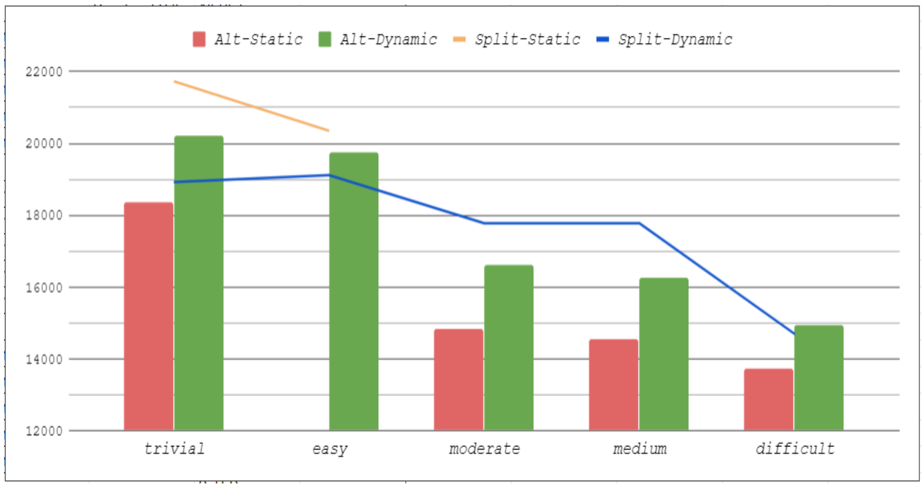
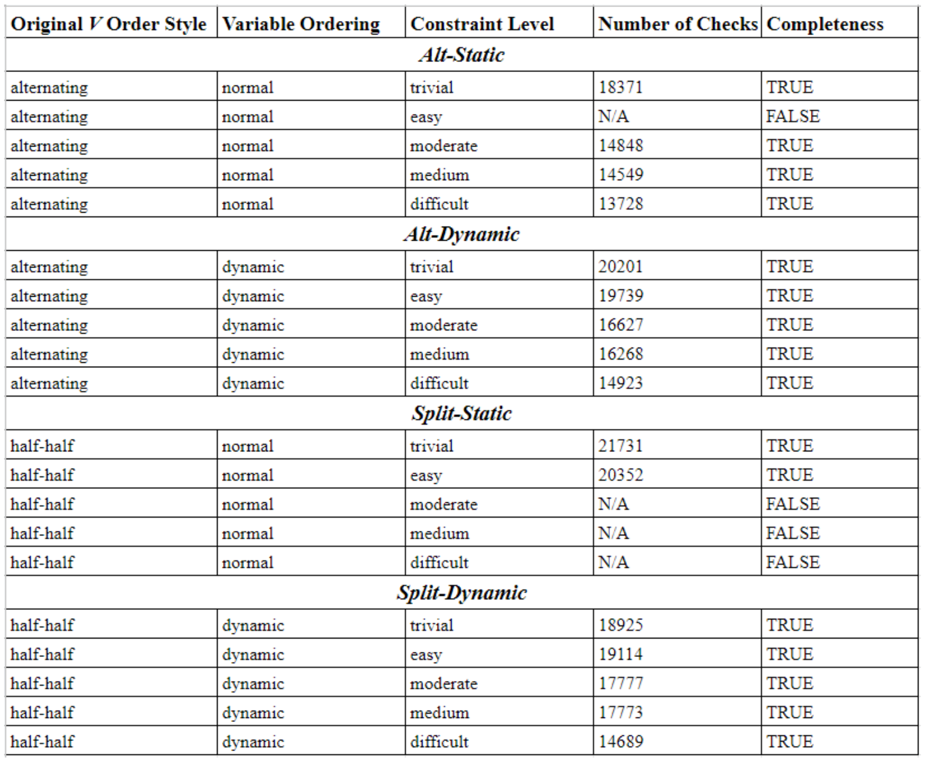


Figure 1: Graphic visualization of *Figure 2*

The naming convention used here is V-DVO, where the first half is V's original order, and the second half is whether there was dynamic or static variable ordering. The N/A values are due to the majority of the 100 runs not returning a complete solution or getting stuck in an infinite loop. The previous graph visualized this data for an easier analyses:



Original *V* Order Style	**Variable Ordering**	**Constraint Level**	**Number of Checks**	**Completeness**
Alt-Static				
alternating	normal	trivial	18371	TRUE
alternating	normal	easy	N/A	FALSE
alternating	normal	moderate	14848	TRUE
alternating	normal	medium	14549	TRUE
alternating	normal	difficult	13728	TRUE
Alt-Dynamic				
alternating	dynamic	trivial	20201	TRUE
alternating	dynamic	easy	19739	TRUE
alternating	dynamic	moderate	16627	TRUE
alternating	dynamic	medium	16268	TRUE
alternating	dynamic	difficult	14923	TRUE
Split-Static				
half-half	normal	trivial	21731	TRUE
half-half	normal	easy	20352	TRUE
half-half	normal	moderate	N/A	FALSE
half-half	normal	medium	N/A	FALSE
half-half	normal	difficult	N/A	FALSE
Split-Dynamic				
half-half	dynamic	trivial	18925	TRUE
half-half	dynamic	easy	19114	TRUE
half-half	dynamic	moderate	17777	TRUE
half-half	dynamic	medium	17773	TRUE
half-half	dynamic	difficult	14689	TRUE

Figure 2: Number of checks per scenario

Analysis

From Figure 2 it is clear that the only incomplete scenarios are the ones with a static variable ordering. The more interesting result, however, is that *Alt-Static* is consistently better than *Alt-Dynamic*, with the exception of its incomplete results in the easy constraints. At first I had expected any implementation of DVO to be superior to static variable ordering; however, the results proved me wrong. Upon further inspection of the algorithm, it became clear why *Alt-Static* would beat *Alt-Dynamic.* The nature of an alternating pattern would benefit a naive algorithm for this specific problem as it would allow each dancer to add the immediate dancer afterwards as a value for itself, since their roles are always compatible. The less constraints available, the more likely the next dancer is a legal option, which is why we see the difference between *Alt-Static* and *Alt-Dynamic* decrease as the constraints increase in difficulty. *Split-Dynamic* also outperforms *Alt-Dynamic*, which makes *Alt-Dynamic* seem to be a not so wise option. The reasoning behind *Split-Dynamic*'s advantage here is that when it makes a check call, it runs it through its own role first, and then runs it through the other role. The chances of finding a conflict with a dancer from your own role are higher (due to sharing partners or level differences between both of the partners) and that then results in finding a conflict sooner, thus making less check calls. Finally, a global trend is noticed here, and that is that all of the scenarios increase their effectiveness as the constraints become more difficult. This is a consistent result with both [1] and [3]'s which shows how both CBJ and DVO become more effective as the problems get more difficult. The ideal algorithm to work with would be *Alt-Static* if one is able to guarantee a perfectly alternated pattern of the set of variables.

Conclusion and Future Work

My work here has successfully created an algorithm that can solve a complex CSP that needs a dual assignment of values per variable, and also each variables domain is the set of variables themselves. Experimental results show that, given the appropriate initial static variable ordering, the *Alt-Static* would be the most effective at solving this problem. The question still stands on how to tackle similar problems that do not have a divide or segragation between their variables. The complexity of this problem leads me to believe that there is more work to be done with complex CSPs with a similar nature to this ballroom dancing problem. There is plenty of potential here to explore and expand on what is just starting. I expect some sort of implementation of the *nogood* class mentioned in [6] to prove some usefulness, as well as a better and more intentional implementation of the *local changes* class mentioned in [7] to also improve efficiency.

References

[1] F. Bacchus and P. Van Run. Dynamic variable ordering in csps. In *International Conference on Principles and Practice of Constraint Programming*, pages 258–275. Springer, 1995.

[2] B. Benhamou and M. R. Saïdi. A new incomplete method for csp inconsistency checking. In *AAAI*, pages 229–234, 2008.

[3] D. Frost and R. Dechter. In search of the best constraint satisfaction search. In *AAAI*, volume 94, pages 301–306, 1994.

[4] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction. In *Proceedings eighth national conference on artificial intelligence*, pages 25–32, 1990.

[5] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence*, 9(3):268–299, 1993.

[6] T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal on Artificial Intelligence Tools*, 3(02):187–207, 1994.

[7] G. Verfaillie and T. Schiex. Solution reuse in dynamic constraint satisfaction problems. In *AAAI*, volume 94, pages 307–312, 1994.

[8] R. J. Wallace and E. C. Freuder. Heuristic methods for over-constrained constraint satisfaction problems. In *International Workshop on Over-Constrained Systems*, pages 207–216. Springer, 1995.