Partial Domain Search Tree for Constraint-Satisfaction Problems

Guni Sharon
ISE Department
Ben-Gurion University
Israel
gunisharon@gmail.com

Abstract

The traditional approach for solving Constraint satisfaction Problems (CSPs) is searching the *Assignment Space* in which each state represents an assignment to some variables. This paper suggests a new search space formalization for CSPs, the *Partial Domain Search Tree* (PDST). In each PDST node a unique subset of the original domain is considered, values are excluded from the domains in each node to insure that a given set of constraints is satisfied. We provide theoretical analysis of this new approach showing that searching the PDST is beneficial for loosely constrained problems. Experimental results show that this new formalization is a promising direction for future research. In some cases searching the PDST outperforms the traditional approach by an order of magnitude. Furthermore, PDST can enhance Local Search techniques resulting in solutions that violate up to 30% less constraints.

Introduction

CSPs are defined by a set of variables V, a set of domains D and a set of constraints C. The task is to assign each variable $var_i \in V$, a value from its domain $D_i \in D$, so that no constraint $c \in C$, is violated. A constraint prohibits an assignment to a subset of V. One way to solve a CSP is by systematically iterating over all possible assignments until a solution is encountered. This is known as searching the *assignment space*. The assignment space can be formulated as a tree, denoted as the *Partial Assignment Search Tree* (PAST). Each node in PAST represents a unique *partial assignment*. In contrast to a *complete assignment*, in a partial assignment values are only assigned to a subset of V. In each PAST node, one new variable is assigned a value. Consequently, the leaf nodes of PAST contain all complete assignments.

In this paper we introduce a new search space formalization: the *Partial Domain Space* (PDS). Each state in the PDS represents a set of *partial domains* (D'). Each partial domain $D'_i \in D'$ is a subset of $D_i \in D$. Similar to the assignment space, the PDS can be formulated as a tree, denoted as *Partial Domain Search Tree* (PDST). A node N in PDST consists of a set of partial domains and a complete assignment. If its assignment violates a constraint c, N is split to two successors. The partial domain in each successor is reduced in such a way that any chosen assignment will not violate c. Consequently, c will never be violated again in the subtree beneath N. PDST is directed towards satisfying constraints instead of iterating through all assignments. Nevertheless, we show that a systemic search of PDST is complete.

Experimental results show that searching PDST can outperform searching PAST by a factor of 19 when both are enhanced by AC. PDST can also perform as a framework for Local Search (LS) algorithms. We show that LS enhanced by PDST finds better solutions (less violated constraints) compared to the traditional Random Restart framework.

Definitions and Background

In CSP, given a set of variables $V = \{var_1, \ldots, var_n\}$, a set of domains $D = \{D_1, \ldots, D_n\}$ and a set of constraints C, the goal is to assign each variable var_i, a value from its domain D_i, so that no constraint is violated. In this work we only relate to binary constraints. We define a binary constraint by a tuple $(var_1, val_1, var_2, val_2)$ where the assignment $var_1 \leftarrow val_1, var_2 \leftarrow val_2$ is **illegal**.

The following is a summary of the notation used throughout this paper:

- **Partial assignment** - Assignment to a subset of V.
- **Complete assignment** - Assignment to all variables in V.
- **Consistent assignment** - An assignment that does not violate any constraint.
- **Solution** - A consistent complete assignment.
- **Set of full domains (D)** - The original set of domains: for each variable var_i a domain D_i.
- **Set of partial domains (D')** - A set of partial domains: for each variable var_i a domain D'_i that is a subset of D_i.

A *complete* CSP solver will halt and return a solution if one exists or else it will halt and return false. Most complete CSP solvers search the *Partial Assignment Search Tree* (PAST) (Kumar 1992).

The Partial Assignment Search Tree

PAST is a search tree which spans all the possible assignments. Each node in PAST corresponds to a unique partial assignment. The root node of PAST consists an empty partial assignment, i.e., no variable is assigned. In each node N, one variable (var_i) is chosen for assignment, as a consequence,

it has D_i unique successors. In each successor of N, var_i is assigned a unique value from D_i. Since each node assigns one variable, the depth of PAST is exactly $|V|$. All nodes at depth $|V|$ (the leaf nodes) consist of a complete assignment that is potentially a solution.

Different CSP solvers that search PAST may use different ordering of variables or values and different pruning techniques but they are all based on the backtrack (BT) algorithm (AKA Depth-First Search) (Kumar 1992).

Many enhancements were presented to BT over the years, for a comprehensive survey see (Ghdira 2013). Some of these enhancements may also be applicable to other state space representation. For instance, AC (Zhang and Yap 2001; Bessière and Régin 2001) can also be used to detect infeasible partial domains, which are the focus of this paper.

The Partial-Domain Search Tree

The *partial-domain search tree* (PDST) is a novel search space formalization for CSPs that is a generalization of the Conflict-Based Search algorithm for multi-agent pathfinding (Sharon *et al.* 2012; 2015).

Nodes in PDST

Each node in PDST consists of: **(1)** A set of partial domains (D') and **(2)** A complete assignment (not necessarily consistent). In the root node, D' is equivalent to the original set of domains, D, (i.e., $\forall i : D'_i = D_i$). The complete assignment for each PDST node is chosen from D'. In other words, every variable var_i takes on a value from its current partial domain D'_i. In a *goal node* the chosen complete assignment is consistent, i.e., a solution. There are several ways that a complete assignment can be chosen such as choosing an arbitrary assignment or using local search techniques that return a complete assignment. We say that a PDST node N, *permits* an assignment s, if for each variable var_i, the value it is given in s is part of D'_i. The root PDST node, for example, permits all possible assignments.

Generating successors

Let N be a non goal PDST node. Since it is not a goal node, the assignment of N violates at least one constraint $(var_1, val_1, var_2, val_2)$ denoted as c. All solutions must satisfy c and so in any solution either var_1 or var_2 or both must take a different assignment. In other words, all solutions must belong to one of the following types: **(Type 1)** var_1 is not assigned val_1 and var_2 is assigned val_2. **(Type 2)** var_2 is not assigned val_2 and var_1 is assigned val_1. **(Type 3)** Both var_1 and var_2 are not assigned val_1 and val_2 respectively. The different solution types are disjoint, i.e., any solution belongs to one and only one type. Node N generates two successors, N_1 and N_2, for covering all possible solutions. At N_1, val_1 is removed from D'_1. This permits all solutions of type 1 and 3. At N_2, val_2 is removed from D'_2 and all values except val_1 are removed from D'_1, forcing var_1 to take val_1. This permits all solutions of type 2.

Observation 1 - The set of complete assignments that is permitted by one immediate successor is disjoint from the set permitted by the other.

Algorithm 1: Partial-Domain Backtrack

```
   Input: Set of domains D'
1  A ← get_assignment(D')
2  if A is consistent then
3      Return TRUE
4  c ← get_constraint(A)
5  //Successor N_1
6  if remove(D', c.var_1, c.val_1) then
7      if AC(D') then
8          if PDBT(D') then
9              return TRUE
10     restore(D')
11 //Successor N_2
12 if remove(D', c.var_2, c.val_2) then
13     D_{c.var_1} = {c.val_1}
14     if AC(D') then
15         if PDBT(D') then
16             return TRUE
17     restore(D')
18 Return FALSE
```

Observation 2 - If k is the set of leaf nodes that are descendants of N, each solution permitted by N must be permitted in exactly one node from k.

Observation 3 - A given node cannot permit a complete assignment that is not permitted by all of its ancestors.

Theorem 1 *PDST contains no duplicates.*

Proof: Let N_i and N_j be two different nodes in PDST.

Case 1 - N_i and N_j are not on the same branch: Let N_a be the first common ancestor of N_i and N_j. Each successor of N_a permits a set of complete assignments that is disjoint from the set of the other (observation 1). N_i and N_j cannot permit a complete assignment that is not permitted by all their ancestors (observation 3). As a result the set of assignments permitted by N_i and the set permitted by N_j are disjoint and so each must have a unique partial domain.

Case 2 - One node (N_i) is an ancestor of the other (N_j): The size of the domain of at least one variable is reduced between successive nodes. Consequently, at least one variable has a bigger domain in N_i compared to N_j. As a result of both cases, the set of partial domains of any two nodes must be unique meaning no duplicates can exist. □

Searching PDST can be done using a variant of the backtrack (BT) algorithm denoted as *Partial-Domain Backtrack*.

The Partial-Domain Backtrack Algorithm

Algorithm 1 presents a recursive variant of the Partial-Domain Backtrack (PDBT) for searching PDST. We cover PDBT line by line. Initially PDBT is called with $D' = D$ as a parameter. This call corresponds to the root PDST node. A complete assignment is chosen and assigned to variable A (Line 1). Next, a goal test checking if A is consistent

is performed (Line 2). Assume A violates the constraint $(var_1, val_1, var_2, val_2)$ which is stored in c (Line 4). Two successors are generated from the extracted constraint N_1 and N_2. In successor N_1, the value val_1 is removed from D'_1 (Line 6). If val_1 is the only remaining value in D'_1, the $remove()$ function will return $FALSE$, val_1 will not be removed and successor N_1 will not be generated. AC is optional and can be called in order to further reduce the set of domains or find it infeasible (Lines 7,14). If the recursive call of N_1 returns $FALSE$, PDBT turns to generate N_2. At N_2, val_2 is removed from D'_2 (Line 12) and all variables but val_1 are removed from var_1 (Line 13). If no solution is found under N_2, $FALSE$ is returned.

Theoretical Analysis

Assume a CSP with n variables where the domain size of each variable is a constant d and there are t constraints.[1] We start by examining the size of PDST for this general problem. Each node in PDST enforces one constraint that will never be violated in the subtree beneath it. Consequently, the maximal depth of PDST is t. Since it is a binary tree, the size of PDST is $O(2^t)$. Since there can be a constraint between any pair of values, $t = O([dn]^2)$. Hence, the size of PDST is exponential in $(dn)^2$.

For the same problem a PAST has a depth of n as each level in the tree assigns one more variable. The branching factor of PAST is d as each value is considered for each variable. Consequently, the size of PAST is $O(d^n)$. In the worst case scenario where the problem is dense with constraints and $t \approx (dn)^2$, $n \ll t$ and the size of PAST is smaller compared to that of PDST. In practise, PDST is considerably smaller compared to PAST in many cases since t can be very small compared to $(dn)^2$. Moreover, regardless of the size of t, the root node of PDST may (fortunately) be assigned a solution. And so, the size of the PDST in the best case is 1 for any solvable problem. In fact, spanning a PDST of size 2^t requires a set of 2^t bad assignment choices, in most cases this is highly improbable. By contrast, the size of the PAST in the best case, for a solvable problem, is d since any solution must be at depth d. Nevertheless, PDBT is a complete algorithm.

Theorem 2 *PDBT is complete.*

Proof: (1) Assume a general CSP instance with at least one solution. Since the root PDST node permits all solutions any solution must be permitted by one PDST leaf, N_l (Observation 2). The complete assignment in N_l must be a solution otherwise N_l is not a leaf or doesn't permit a solution. As a systemic search PDBT must examine all PDST nodes and N_l in particular.

(2) The size of PDST is finite (bounded by $O(2^t)$). Since PDST contains no duplicate nodes (Theorem 1) any systemic search algorithm, and PDBT as a special case, will examine a finite number of nodes. □

[1]The domain sizes of all variables are chosen to be constant ($= d$) for the sake of discussion but in fact d can be viewed as the size of the maximal domain over all variables.

Local Search

So far we discussed complete algorithms that systematically search the state space. Unfortunately, systematic search is impractical for many CSP instances since the search space is too big. These infeasible problems are commonly solved using local search techniques (LS) which are usually very fast but incomplete.

LS is initialized with a randomly selected complete assignment. Then, it attempts to iteratively decrease the number of violated constraints by changing a single assignment at a time. If LS gets stuck in a local minima or a time bound has expired a *random restart* procedure is performed. In random restart the assignment is initialized randomly or according to a heuristic that guesses a promising initial assignment (Martí *et al.* 2013). Different LS variants may take different approaches towards choosing the next assignment in a given iteration while balancing between exploiting promising direction and exploring new areas of the search space (Hao and Pannier 1998). PDBT can utilise LS as described next.

Using Local Search in PDBT

At each recursive call PDBT chooses a complete assignment out of the given set of partial domains (D') by calling $get\_assignment(D')$ (Line 1 of Algorithm 1). It is possible to implement this function using a LS solver in order to retrieve a complete assignment. The chosen LS solver must halt after a finite amount of time. Using LS will increase the probability of finding a solution in each PDST node at the cost of extra computational effort per node. When the PDST is very large, using LS to try and find a solution at early stages of the search is worthwhile. Note that unlike the random restart procedure, the PDBT framework is complete and can identify unsolvable problems. The benefits of using PDBT with LS are visible in the experimental evaluation section presented next.

Experimental Evaluation

Each problem setting is represented by a tuple (n, d, p_1, p_2) where n is the number of variables, d is the domain size of all variables, $p1$ is the probability that any pair of agents will be in conflict (AKA density) and $p2$ is the probability that any two values belonging to conflicting variables will be in conflict (AKA tightness). For each problem setting 100 instances were generated. We report the average number of *constraint checks* (CC) which is the number of times an assignment was verified to be allowed by a constraint. The BT algorithm for PAST is denoted as PABT as opposed to the BT algorithm for PDST (PDBT).

PDBT Vs PABT

The first experiment compares basic PDBT and PABT (without AC). n was set to 15, d to 5 and the number of constraint varied (different p_1 and p_2 values). Figure 1 presents CC measurements (x-axis) as a function of different p_2 values (y-axis). Each frame presents results for a different p_1 value (0.1, 0.3, 0.5, 0.7).

Recall that the size of PDST is exponential in the number of constraints while the size of PAST is exponential in

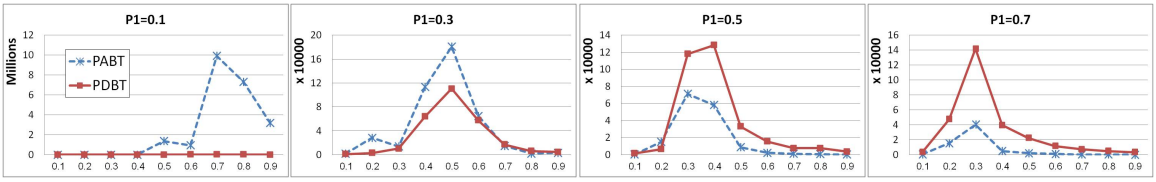

Figure 1: PDBT Vs PABT. The average number of constraint checks performed (y-axis) for different p_2 values (x-axis).

n	d	$\frac{n}{d}$	Constraint Checks		
			PABT	PDBT	Ratio
22	11	2.0	**106,643**	236,146	0.45
34	7	4.9	**193,718**	282,912	0.68
40	6	6.7	**690,066**	955,283	0.72
48	5	9.6	**362,424**	441,732	0.82
60	4	15.0	157,343	**102,447**	1.54
80	3	26.7	241,005	**25,358**	9.50
120	2	60.0	672,133	**34,636**	19.41

Table 1: Constraint checks for PABT+AC and PDBT+AC.

the number of variables. This translates to the fact that the relative performance of PDBT compared to PABT degrades as the number of constraints increases and other parameters are fixed. For low p_1 values (0.1, 0.3), where variables are loosely constrained, PDBT outperforms PABT. On the other hand, when variables are tightly constrained ($p_1 > 0.4$) PABT prevails.

Using Arc-Consistency

The second experiment compared PDBT and PABT when both use AC3.1 (Zhang and Yap 2001) to detect and prune invalid partial assignments/domains. The values of n and d varied while for each pair of (n, d) all p_1 and p_2 combinations were used where $p_1, p_2 \in \{0.1, 0.2, ..., 0.9\}$. Though n and d varied, the value of (dn) was set to 240 in order to keep a controlled environment with regard to the growth in the size of PDST (exponential in $[dn]^2$). Each row presents the average of over 81 (p_1, p_2) combinations and 8100 instances. In this experiment we considered only instances that are solvable since they are Arc Consistent. Both algorithms will identify an instance that is not Arc Consistent immediately, making those instances irrelevant.

Table 1 presents the CC measurements for both algorithms (best value in each line is given in bold). The ratio n/d is also given and presents a positive correlation with the relative performance of PDBT. As n/d increases so does the CC ratio between PABT and PDBT. $n/d = 10$ is a crossover point after which PDBT outperforms PABT.

Local Search

The third experiment shows the benefits of using PDBT with LS. Two LS solvers were used: the *Hill Climbing* algorithm (HC) (Selman *et al.* 1992) and the more advanced *Simulated Annealing* algorithm (Hao and Pannier 1998).[2] Each of the two algorithms was used once under the random restart procedure (RR) and second under the PDBT framework.[3]

P_1	Violations			
	Hill Climbing		Simulated A.	
	RR	PDBT	RR	PDBT
0.1	84	62	82	**58**
0.2	242	**204**	247	208
0.3	422	**369**	436	381
0.4	613	**524**	637	546
0.5	812	**722**	846	751
0.6	1,015	**901**	1,058	938
0.7	1,223	**1,094**	1,274	1,141
0.8	1,434	**1,138**	1,494	1,209
0.9	1,646	**1,317**	1,715	1,386
Total	832	**703**	865	735

Table 2: Number of violations for PDBT and RR.

Table 2 presents the number of constraints violations in the solutions returned by each of the algorithms within a time bound of five seconds. In this experiment we set $n = 100$ and $d = 20$. p_1 values are reported at the leftmost column. For each p_1 value 900 instances were generated, 100 for each p_2 value.

In total, PDBT finds better solutions (with less violated constraints) compared to RR across all the P_1 range.

Discussion and future work

This paper presents a new formalization and a corresponding search tree for CSPs denoted as the *Partial Domain Search Tree* (PDST) that is based on the CBS algorithm (Sharon *et al.* 2015). The traditional *Partial Assignment Search Tree* (PAST) was the focus of extensive research and many enhancements were introduced to it over the years. PDBT in its current/basic form is not competitive with state-of-the-art PABT solvers. The empirical results suggest that when both PDBT and PABT use the same enhancement (AC), the first can outperform the later by more than an order of magnitude in some cases. For other cases, where variables are tightly constrained, the traditional PABT approach prevails. Adapting more enhancements as well as coming up with new special enhancements for PDBT is left for future work.

PDBT shows promising results as a Local Search framework where it is able to find better solutions (less violated constraints) compared to the Random Restart approach.

[2] Simulated Annealing was set using The Metropolis acceptance criterion (Metropolis *et al.* 1953) with a temperature parameter = 1 and was bounded to 1,000 steps.

[3] In our experiments PDBT used AC3.1 (Zhang and Yap 2001) to detect and prune infeasible sets of partial domains.

Acknowledgments

We would like to thank: Ariel Felner, Roie Ziven and Malte Helmert for their advice and help on this issue.

References

Christian Bessière and Jean-Charles Régin. Refining the basic constraint propagation algorithm. In *IJCAI*, pages 309–315, 2001.

Khaled Ghdira. *Constraint Satisfaction Problems: CSP Formalisms and Techniques*. Wiley, 2013.

Jin-kao Hao and Jerome Pannier. Simulated annealing and tabu search for constraint solving. In *Symposium on Artificial Intelligence and Mathematics*. Citeseer, 1998.

Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI magazine*, 13(1):32, 1992.

Rafael Martí, Mauricio GC Resende, and Celso C Ribeiro. Multi-start methods for combinatorial optimization. *European Journal of Operational Research*, 226(1):1–8, 2013.

Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

Bart Selman, Hector J Levesque, and David G Mitchell. A new method for solving hard satisfiability problems. In *AAAI*, volume 92, pages 440–446, 1992.

Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent path finding. In *AAAI*, 2012.

Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artif. Intell.*, 219:40–66, 2015.

Yuanlin Zhang and Roland HC Yap. Making ac-3 an optimal algorithm. In *IJCAI*, pages 316–321, 2001.