A Simple Polynomial-Time Randomized Distributed Algorithm for Connected Row Convex Constraints

T. K. Satish Kumar*[†], Duc Thien Nguyen[‡], William Yeoh[††], and Sven Koenig[†]

[†]Department of Computer Science
University of Southern California
Los Angeles, CA 90089, USA
tkskwork@gmail.com, skoenig@usc.edu

[‡]School of Information Systems
Singapore Management University
Singapore 178902
dtnguyen.2011@smu.edu.sg

[††]Department of Computer Science
New Mexico State University
Las Cruces, NM 88003, USA
wyeoh@cs.nmsu.edu

Abstract. In this paper, we describe a simple randomized algorithm that runs in polynomial time and solves connected row convex (CRC) constraints in distributed settings. CRC constraints generalize many known tractable classes of constraints like 2-SAT and implicational constraints. They can model problems in many domains including temporal reasoning and geometric reasoning; and generally speaking, play the role of "Gaussians" in the logical world. Our simple randomized algorithm for solving them in distributed settings, therefore, has a number of important applications. We support our claims through empirical results. We also generalize our algorithm to tractable classes of tree convex constraints.

Keywords: Connected Row Convex Constraints; Distributed CSPs; Randomized Algorithms.

1 Introduction

Many combinatorial problems in Artificial Intelligence can be modeled as *constraint satisfaction problems* (CSPs). While the task of solving general CSPs is NP-hard, much work has been done on identifying tractable subclasses of CSPs. An important subclass of tractable constraints is that of *connected row convex* (CRC) constraints.

CRC constraints generalize many other known tractable classes of constraints like 2-SAT, implicational, and binary integer linear constraints [2]. In fact, they have been identified as the equivalent of "Gaussians" in the logical world [6]. A filtering algorithm,

* Alias: Satish Kumar Thittamaranahalli

for example, analogous to the Kalman filter in a logical world, is presented in [12] and makes use of CRC constraints. CRC constraints have also been used in the context of temporal reasoning [7, 10] and geometric reasoning [5] to identify tractable classes of temporal and spatial reasoning problems respectively.

Although CRC constraints arise in a wide range of real-world domains, centralized algorithms for solving them are not always useful. This is because often the variables and constraints are distributed among many agents introducing privacy issues and knowledge transfer costs between them [24, 23]. A plethora of real-world problems such as in networked systems, multi-agent planning and belief propagation, resource sharing and scheduling, and distributed spatio-temporal reasoning can therefore benefit from efficient distributed algorithms for CRC constraints. In this paper, we study the problem of solving CRC constraints in distributed settings. We do this using the formalism of *distributed* CSPs [25].

For the centralized version of solving CRC constraints, there are two classes of efficient algorithms. The first class of algorithms is based on the intuition that *path consistency* (PC) ensures global consistency for row convex constraints [21]. CRC constraints are known to be closed under the operations that establish PC and therefore solvable in polynomial time [2]. Additional properties of CRC constraints are used to establish PC faster and therefore solve them more efficiently [27]. A related algorithm also solves CRC constraints very efficiently using variable elimination [28].

The second class of algorithms is based on the idea of *smoothness*: a property of constraints which ensures that "at every infeasible point, there exist two directions such that with respect to any feasible point, moving along at least one of these two directions decreases a certain distance measure to it" [9]. Smooth constraints can be solved using very simple randomized algorithms that run in polynomial time [6, 8]. Moreover, it is easy to show that CRC constraints are a special case of smooth constraints when the distance metric used is the *manhattan distance* [8, 9]. This makes CRC constraints amenable to the above-mentioned simple but very powerful polynomial-time randomized algorithms.

In this paper, we will use the simplicity of the randomized algorithm for CRC constraints to solve them efficiently in distributed settings as well. This straightforward generalization stands in stark contrast to the difficulty of generalizing PC-based algorithms to distributed settings.

The primary problem associated with PC-based algorithms (including variable elimination) is that new constraints are added between pairs of variables X_i and X_j even if no direct constraint existed between them in the problem specification. In distributed settings, this means that new communications should be opened between pairs of agents which potentially have to go through a large number of other intermediate agents. In turn, this means that the communication complexity becomes prohibitively high. Translating knowledge between agents in an exchangeable format could also be very expensive. Furthermore, in many application domains, having agents exchange information about "inferred" constraints is highly undesirable for purposes of security and privacy.

In contrast, the randomized algorithm that works on smooth constraints follows a simple random walk strategy. It is very simple to implement and does not introduce any newly inferred constraints between the original variables. This power of randomiza-

tion is especially well-suited for generalization to distributed settings. In this paper, we exploit this intuition and present a simple polynomial-time randomized algorithm that solves CRC constraints in distributed settings. We show that the randomized distributed algorithm outperforms even the original centralized version of it. This means that our algorithm effectively factors in parallelism too.[1] Finally, we also show that it is easy to generalize our techniques to solve classes of tree convex constraints that do not have very efficient PC-based algorithms even for the centralized versions of the problems [9].

2 Preliminaries and Background

A CSP is defined by a triplet $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{X} = \{X_1, X_2 \ldots X_N\}$ is a set of *variables* and $\mathcal{C} = \{C_1, C_2 \ldots C_M\}$ is a set of *constraints* between subsets of them. Each variable X_i is associated with a discrete-valued *domain* $D_i \in \mathcal{D}$, and each constraint C_i is a pair $\langle S_i, R_i \rangle$ defined on a subset of variables $S_i \subseteq \mathcal{X}$, called the *scope* of C_i. $R_i \subseteq D_{S_i}$ ($D_{S_i} = \times_{X_j \in S_i} D_j$) denotes all compatible tuples of D_{S_i} allowed by the constraint. $|R_i|$ is referred to as the *arity* of the constraint C_i. A *solution* to a CSP is an assignment of values to all variables from their respective domains such that all constraints are satisfied. In a *binary* CSP, the maximum arity of any constraint is 2. Binary CSPs are representationally as powerful as CSPs and are NP-hard to solve in general.

A *distributed* CSP (DCSP) [25] is defined by the 5-tuplet $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \alpha \rangle$, where $\mathcal{X}, \mathcal{D}$ and $\mathcal{C}$ are the sets of variables, domains and constraints as defined for a regular CSP, respectively; $\mathcal{A} = \{A_1, A_2 \ldots A_P\}$ is a set of *agents*; and $\alpha : \mathcal{X} \rightarrow \mathcal{A}$ maps each variable to an agent.

A *constraint network (constraint graph)* is used to visualize a CSP or DCSP instance. Here, nodes in the graph correspond to variables in the DCSP, and edges connect pairs of variables appearing in the same constraint. We use K to refer to the size of the largest domain and $T(X_i)$ to refer to the set of neighbors of variable X_i in the constraint graph.

A network of binary constraints is said to be *arc consistent* if and only if for any two distinct variables X_i and X_j, and for each value of X_i, there exists a value of X_j that satisfies the direct constraint $C(X_i, X_j)$ between them. Similarly, a network of binary constraints is said to be *path consistent* if and only if for any three distinct variables X_i, X_j and X_k, and for each pair of consistent values of X_i and X_j that satisfies the constraint $C(X_i, X_j)$, there exists a value of X_k such that the constraints $C(X_i, X_k)$ and $C(X_j, X_k)$ are also satisfied.

Conceptually, algorithms that enforce PC work by iteratively "tightening" the binary constraints of a constraint network [11]. When binary constraints are represented as matrices, PC algorithms employ the three basic operations of *composition*, *intersection* and *transposition*. The *(0,1)-matrix representation* of a constraint $C(X_i, X_j)$ between the variables X_i and X_j consists of $|D_i|$ rows and $|D_j|$ columns when orderings on the domains of X_i and X_j are imposed. The '1's and '0's in the matrix respectively indicate

[1] It is important to note that the research motivation is fundamentally different from just exploiting parallelism. Unlike in distributed/parallel computing where we are free to choose the system architecture to increase efficiency, in a distributed CSP, the problem situation is already fixed [24, 23].

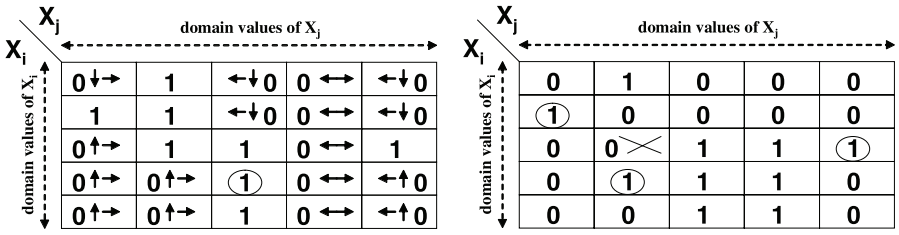

Fig. 1. Illustrates the idea of smoothness. The left side, (a), is a binary smooth constraint. It satisfies the property that at each '0', there exist two directions, as indicated, such that with respect to any '1', as encircled, moving along at least one of these two directions decreases the manhattan distance to it. The right side, (b), does not satisfy the property of smoothness. The required pair of directions does not exist at the '0' marked with a cross. In some sense, this is because the '1's lie on three or more sides of it, as shown encircled. Binary smooth constraints are the same as CRC constraints, as proved in [8]. (a) therefore satisfies the definitional conditions for a CRC constraint as stated in the running text as well.

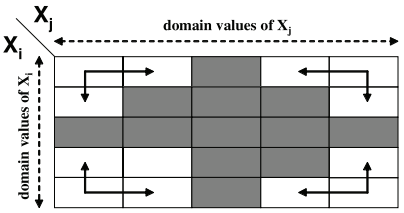

Fig. 2. Shows the general geometry of a CRC constraint and the pattern of the required pairs of directions at the infeasible points. Shaded areas represent feasible combinations in the constraint.

the allowed and disallowed tuples. Figure 1 presents examples of matrix notations for binary constraints.

A binary constraint represented as a (0,1)-matrix, is *row convex* if and only if, in each row, all of the '1's are consecutive. It has been shown in [21] that if there exist domain orderings for all the variables in a path consistent network of binary constraints such that all the constraints can be made row convex, then the network is *globally consistent*.[2] The orderings on the domain values of all the variables are critical to establishing row convexity in path consistent networks. In order to find these required domain orderings for row convexity, the well-known result of [1] is used.

Although row convexity implies global consistency in path consistent networks, the very process of achieving PC may destroy it. This means that row convexity of the original set of constraints does not necessarily imply global consistency. CRC constraints avoid this problem by imposing a few additional restrictions. A (0,1)-matrix is CRC if,

[2] A globally consistent network has the property that a solution can be found in a backtrack-free manner.

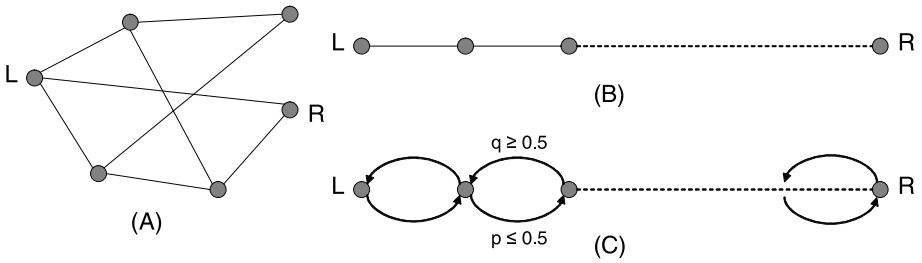

Fig. 3. Shows three scenarios in which random walks are performed. In an undirected graph, as in (a) and (b), for any two nodes L and R, $T(R, L)+T(L, R)$ is related to the "resistance" between them. In (c), $p \leq q$ at every node, and $T(R, L)$ is less than that in (b) because of an increased "attraction" towards L at every node.

after removing empty rows and columns,[3] it is row convex and connected, that is, the positions of the '1's in any two consecutive rows intersect, or are consecutive. Unlike row convex constraints, CRC constraints are closed under composition, intersection and transposition—the three basic operations employed by algorithms that enforce PC—hence establishing that enforcing PC over CRC constraints is sufficient to ensure global consistency [2]. Figures 1(a), 1(b) and 2 show examples to illustrate the properties of CRC constraints.

3 Random Walks and Expected Arrival Times

In this section, we will provide a quick overview of *random walks*, and some theoretical properties associated with them. Figure 3(a) shows an undirected graph (with unit weights assumed on all edges). A random walk on such a graph involves starting at a particular node; and at any stage, randomly moving to one of the neighboring positions of the current position. A property associated with such random walks on undirected graphs is that if we denote the expected time of arrival at some node (say L) starting at a particular node (say R) by $T(R, L)$, then $T(R, L) + T(L, R)$ is $O(m\mathcal{H}(L, R))$; m is the number of edges, and $\mathcal{H}(L, R)$ is the "resistance" between L and R when the unit weights on the edges are interpreted as electrical resistance values [3].

Figure 3(b) shows a particular case of the one in Figure 3(a) in which the nodes of the graph are connected in a linear fashion—i.e., the probabilities of moving to the left or to the right from a particular node are equal (except at the end-points). In this scenario, it is easy to note that by symmetry, $T(L, R) = T(R, L)$. Further, using the property of random walks stated above, if there are n nodes in the graph, then both $T(L, R)$ and $T(R, L)$ are $O(n^2)$.

Figure 3(c) shows a slightly modified version of that in Figure 3(b), where the graph is directed, although it is still linear. Moreover, there are weights associated with edges, which are interpreted as probabilities in the random walk; and the weight on $\langle s, s_{left}\rangle$

[3] rows or columns with only '0's

is, in general, not equal to that on $\langle s, s_{right} \rangle$. Here, s is some node in the graph, and s_{left} and s_{right} are, respectively, the nodes occurring immediately to the left and right of it. However, we are guaranteed that the probability of moving to the left at any node is greater than that of moving to the right (i.e., $p \leq q$). Given this scenario, it is easy to see that the expected time of arrival at the left end-point (L), starting at the right end-point (R), is also $O(n^2)$ (if there are n nodes in all). Informally, this is because at every node, there is an increased "attraction" to the left compared to that in Figure 3(b); and the expected arrival time can only be less than that in the latter.

4 A Simple Polynomial-Time Randomized Algorithm for CRC Constraints

In [8, 9], the theory of random walks is shown to be useful in identifying tractable classes of constraints.[4] In particular, simple randomized algorithms for solving CRC constraints are presented in [8].

Roughly, the idea is to start out with an initial arbitrary assignment to all the variables from their respective domains and use the violated constraints in every iteration to guide the search for the satisfying assignment E^* (if it exists). In particular, in every iteration, a violated constraint is chosen, and the rank of the assignment of one of the participating variables (according to the domain orderings) is either increased or decreased. Since we know that the true assignment E^* satisfies all constraints, and therefore the chosen one too, randomly moving along one of the two directions associated with the '0' corresponding to the current assignment E will reduce the manhattan distance to E^* with a probability ≥ 0.5.[5]

Much like the random walk in Figure 3(c), therefore, we can bound the convergence time to E^* by a quantity that is only quadratic in the maximum manhattan distance between any two complete assignments. (We also note that the manhattan distance between two complete assignments E_1 and E_2 is at most NK, and 0 if and only if $E_1 = E_2$.) See [8] for more details and further reductions in the time/space complexities. The expected running time of the randomized algorithm is $O(\gamma N^2 K^2)$ where γ is the maximum degree of any node in the constraint network.[6] This complexity matches that of the best algorithms for solving CRC problems with high treewidths [15].

---

[4] Also see [18] for similar randomized algorithms for 2-SAT.

[5] The manhattan distance between two complete assignments $E_1 = \langle X_1 = v_1, X_2 = v_2 \ldots X_N = v_N \rangle$ and $E_2 = \langle X_1 = v'_1, X_2 = v'_2 \ldots X_N = v'_N \rangle$ is $|r(v_1) - r(v'_1)| + |r(v_2) - r(v'_2)| \ldots |r(v_N) - r(v'_N)|$. Here, $r(v)$ is the rank of the domain-value v in the corresponding domain.

[6] Because of the *Amplification Lemma*, it suffices to analyze a randomized algorithm using the expected running time [16]. This does not mean that the algorithm works only on a fraction of the input instances. The algorithm works on *all* input instances with the same analysis. The probabilities are associated with the internal coin flips of the algorithm and are controlled using the *Amplification Lemma*.

Algorithm 1: DISTRIBUTED-CRC()

1 Each agent X_i calls INITIALIZE()

Procedure Initialize

2 $d_i = \textbf{ValInit}(X_i)$
3 $R_i = \{\langle X_k, \textbf{ValInit}(X_k)\rangle \mid X_k \in T(X_i)\}$
4 $t_i = 0$
5 $z_i = 0$
6 **for** $X_k \in T(X_i)$ **do**
7 $\quad r_{ik} =$ NULL
8 **end**
9 Send VALUE(X_i, d_i, t_i) to each $X_k \in T(X_i)$

5 A Simple Polynomial-Time Randomized Distributed Algorithm for CRC Constraints

In this section, we will generalize the polynomial-time randomized algorithm for solving CRC constraints to work in distributed settings. The distributed algorithm is described in Algorithm DISTRIBUTED-CRC and its accompanying 'Procedures'. The algorithm and procedures are mostly self-explanatory, but we provide a brief explanation below. Without loss of generality, we will assume that each agent A_i is associated with a unique variable X_i [23]. When there is no scope for confusion, we will use A_i and X_i interchangeably.

Each agent X_i maintains the following data structures:

- Its current value d_i, which is initialized by **ValInit**(X_i)—the *first* value in the ordered domain of X_i.
- Its current context R_i, which is initialized with all tuples of its neighbors and their initial values. R_i is X_i's belief about the current values of its neighbors.
- Its current time index t_i, which is initialized to 0. The time index is used to synchronize all the agents such that they progress in synchronized iterations.
- Its random variable z_i, which is initialized to 0. z_i is used to ensure that each iteration of the distributed algorithm has a valid serialization order of its steps in terms of the centralized random walk strategy.
- Its decision variable r_{ik} for each constraint $C(X_i, X_k)$, which is initialized to 'NULL'. r_{ik} indicates which agent should change its value if $C(X_i, X_k)$ is violated.
- Its recommended directions $\textit{possibleDirections}^i_{ik}(d_i, d_k)$ for changing its current value in accordance with the property of *smoothness* for $C(X_i, X_k)$. This is a function of the current values d_i of variable X_i and d_k of variable X_k. It is initialized in a pre-processing procedure [8].[7]
- Its priority $p(X_i)$, which is used to determine the agent responsible for "flipping the coin" in order to decide which variable should change its value when $C(X_i, X_k)$ is

[7] The procedure ELIMINATE-DIRECTIONS [8] removes the complement of this set.

Procedure When-Rcvd-Value(X_s, d_s, t_s)

```
10 Update (X_s, d'_s) ∈ R_i with (X_s, d_s)
11 if received VALUE(X_k, d_k, t_i) from all X_k ∈ T(X_i) then
12     z_i ~ U(0, 1)
13     for X_k ∈ T(X_i) do
14         if p(X_i) > p(X_k) and C(X_i, X_k) is unsatisfied then
15             possibleDirections^i_ik(d_i, d_k) = suggested directions for X_i in C(X_i ←
               d_i, X_k ← d_k)
16             if |possibleDirections^i_ik(d_i, d_k)| = 0 then
17                 r_ik = X_k
18             else if |possibleDirections^i_ik(d_i, d_k)| = 1 then
19                 r_ik = ChooseRandom{X_i, X_k}
20             else
21                 r_ik = X_i
22             end
23         else
24             r_ik = NULL
25         end
26         Send PROPOSAL(X_i, r_ik, z_i, t_i) to X_k
27     end
28 end
```

violated. These priorities can be randomly assigned in a pre-processing procedure. Moreover, no two agents have the same priority.

The algorithm starts by each agent making a call to the initialization procedure. In this procedure, each agent X_i initializes to its *first* domain value. Knowing that the other agents do similarly, X_i also initializes its belief about its neighbors in R_i. t_i, z_i and r_{ik} are initialized appropriately as well. X_i then sends a VALUE message to all its neighbors informing them of its current value appropriately time-stamped. This triggers further computation.

In Procedure WHEN-RCVD-VALUE, X_i first updates R_i. Then, if it has received VALUE messages from all neighbors for that iteration t_i, it picks a value for z_i—for imminent use—uniformly at random from the interval $(0, 1)$.[8] It then identifies those neighbors X_k that share a violated constraint $C(X_i, X_k)$ for which it is responsible for "flipping a coin".[9] After identifying such X_k, X_i looks at the recommended directions for change as dictated by the smooth constraint $C(X_i, X_k)$. If both these directions do not concern X_i, then we are in an empty column of the matrix representation of $C(X_i, X_k)$. In such a case, X_k should change its value, and r_{ik} is so assigned. Similarly, if both directions do not concern X_k, X_i should change its value, and $r_{ik} = X_i$. In the case that there is a direction for both X_i and X_k, the choice is made by flipping a fair

[8] For simplicity of argument, we assume that each sample from the continuous interval generates a different value.

[9] By convention, the higher priority agent flips the coin.

Procedure When-Rcvd-Proposal(X_s, r_{is}, z_s, t_s)

```
29 if received PROPOSAL(X_k, r_ik, z_k, t_i) from all X_k ∈ T(X_i) where p(X_k) > p(X_i) and
   C(X_i, X_k) is unsatisfied then
30     for X_k ∈ T(X_i) where C(X_i, X_k) is unsatisfied and r_ik = X_i do
31         if z_i > z_k then
32             possibleDirections^i_ik(d_i, d_k) = suggested directions for X_i in C(X_i ←
               d_i, X_k ← d_k)
33             if |possibleDirections^i_ik(d_i, d_k)| = 1 then
34                 d_i = new value as suggested by
35                 the unique possible direction for X_i
36             else
37                 d_i = new value as suggested by
38                 ChooseRandom(possibleDirections^i_ik(d_i, d_k)) for X_i
39             end
40             break
41         end
42     end
43 end
44 if received PROPOSAL(X_k, r_ik, z_k, t_i) from all X_k ∈ T(X_i) then
45     t_i = t_i + 1
46     Send VALUE(X_i, d_i, t_i) to each X_k ∈ T(X_i)
47 end
```

coin in 'ChooseRandom'. Finally, X_i sends a PROPOSAL message to X_k suggesting changes.

In Procedure WHEN-RCVD-PROPOSAL, X_i waits to receive PROPOSAL messages from each neighbor X_k that shares a violated constraint with it and is of higher priority. If the message from X_k recommends that X_i should change its value, it does so modulo checking that $z_i > z_k$.[10] If there is only one recommended direction of change for X_i, that is simply followed. If there are two directions, we are in an empty row or column, and a choice is made with the flip of a fair coin. The loop breaks after one change is affected for X_i, and VALUE messages are sent to all neighbors for the next iteration.

5.1 Correctness and Convergence

The distributed algorithm for solving CRC constraints employs randomness at various steps. To analyze the randomized distributed algorithm, we can draw intuitions from the randomized algorithm that solves CRC constraints in a centralized setting. In this centralized algorithm, although there might be multiple violated constraints in any iteration, it does not matter which violated constraint we choose to guide the random walk. In any iteration, exactly one variable changes its value. This means that all the violated constraints that do not contain this variable appear as violated constraints in the next iteration too.

[10] As explained later, this is required to prove the convergence of the algorithm by showing that there is a valid serialization order of the various steps in terms of the random walk strategy.

In the distributed algorithm, multiple variables can change their values in the same iteration. However, if we simulate each iteration in this algorithm using a valid sequence of iterations in the centralized algorithm, then the correctness of the former is proved. Let the set of violated constraints in iteration i of the distributed algorithm be $V_d(i)$.

We now present a simple graphical way of visualizing the behavior of the distributed algorithm. This facilitates the proofs of correctness and other properties of the algorithm. Consider the constraint network of the problem instance. In each iteration, some subset of the edges represents the violated constraints. Now let us analyze the subset of violated constraints that affect the algorithm in that iteration.

First, we note that an initial priority is assigned to each variable. These priorities are used to nominate for each constraint, which of the two end-point agents flips the coin to generate a random bit when that constraint is violated. By convention, we assume that the one with the higher priority does this task of coin flipping. Of course, the result of the coin flip is used to decide which agent would be recommended to change its current value. Because the symmetry is broken in this way, we can now represent each violated constraint in iteration i as a directed edge in the constraint graph. By convention, the tail of the directed edge represents the agent recommended to change its value.

From Procedure WHEN-RCVD-PROPOSAL, it is clear that in any iteration, a variable changes its value at most once as per one of the recommendations it receives. In our graphical visualization, this means that an agent heeds to at most one of its outgoing directed edges. Further, consider the subset of violated constraints and their associated recommendations that qualify for affecting the values of the variables. The use of randomly generated z-values in each iteration ensures that the graphical representation of this subset of violated constraints does not have a cycle. Put together, let us call this "effective" subset of violated constraints as $V_{eff}(i)$ and the corresponding DAG induced by it as $G_{eff}(i)$.

Now because $G_{eff}(i)$ is a DAG, there exists a topological ordering on its nodes, starting from a node that has no incoming edges. Further, since every node has at most one outgoing edge in $G_{eff}(i)$, a topological ordering on the nodes represents a topological ordering on the edges as well. In turn, this total ordering on the edges represents a consistent serialization order for simulating the behavior of the distributed algorithm in iteration i using multiple iterations of the centralized algorithm. Consider the first edge and the change of value it recommends for the tail variable. This recommendation does not change the violation status of the other constraints in $G_{eff}(i)$. Repeating this argument for the 2nd, 3rd, 4th edges, and so on, a consistent serialization is achieved. Of course, the simulation uses the same outcome of the random coin flips for the centralized algorithm. (It doesn't matter when the random bits are generated as far as they are generated independently.)

The correctness of the distributed algorithm is now established by its reducibility to a compressed version of the centralized algorithm. The analysis of the running time of the distributed algorithm, however, requires further arguments. In particular, it could be the case that $V_{eff}(i)$ is empty in spite of $V_d(i)$ being non-empty. This stagnates the random walk for that iteration. We will now show that stagnation can happen with a probability of at most 0.5 in any iteration. From the theory of random walks, it is easy to see that proving this would increase the expected convergence time by a factor

Number of Agents $\lvert\mathcal{X}\rvert$	CRC Runtime (s)	DPOP Runtime (s)	D-CRC Runtime (s)
15	76.24	241.60	37.26
16	75.92	374.04	39.14
17	76.20	579.26	39.24
18	69.74	1380.36	39.78
19	85.34	1985.18	39.24
20	96.84	19432.08	40.26
21	102.24	timeout	39.94
22	106.60	timeout	40.02
23	112.92	timeout	39.96
24	113.92	timeout	44.08
25	120.26	timeout	43.30

Table 1. Varying Number of Agents $|\mathcal{X}|$: 0.5 density; domain size = 3.

Domain Size $\lvert D_i\rvert$	CRC Runtime (s)	DPOP Runtime (s)	D-CRC Runtime (s)
3	102.04	38538.50	41.54
4	135.66	274822.40	41.80
5	166.38	timeout	42.66
6	181.15	timeout	45.53
7	201.24	timeout	43.92
8	225.58	timeout	45.24
9	234.65	timeout	47.10
10	243.09	timeout	48.80

Table 2. Varying Domain Size $|D_i|$: 21 agents; 0.5 density.

of at most 2. Put together, this would mean that the expected running time of the distributed algorithm is at most twice that of the centralized algorithm which is known to be polynomial. This analysis, of course, is conservative; and as our experiments show, the distributed algorithm runs much faster in practice.

To prove that stagnation can happen with a probability of at most 0.5 in any iteration, we make the observation that if $V_d(i)$ is non-empty, it must have at least one violated constraint $C(X_u, X_v)$. Let's say that X_u is chosen as the tail variable. For X_u to remain unchanged in that iteration, it must reject all recommendations from X_v (and potentially other variables) based on the z-values. But each such rejection happens with a probability of 0.5. Therefore, X_u must change its value and avoid stagnation with a probability of at least 0.5, hence proving our claim.

6 Experimental Results and Discussions

In this section, we compare our randomized distributed algorithm (D-CRC) with the centralized algorithm of [8][11] and DPOP [19] (an off-the-shelf state-of-the-art DCSP

[11] also abbreviated "CRC" in the tables for convenience

Graph Density p_1	CRC Runtime (s)	DPOP Runtime (s)	D-CRC Runtime (s)
0.1	53.54	52.16	31.72
0.2	64.24	204.38	39.66
0.3	77.12	946.36	44.68
0.4	86.64	4943.90	41.42
0.5	95.88	timeout	41.38

Table 3. Varying Graph Density p_1: 21 agents; domain size = 3.

algorithm).[12] We used a publicly available version of DPOP which is implemented on the FRODO framework [13]. We implemented D-CRC on the same framework to allow for fairer comparisons. We ran our experiments on an Intel Xeon 2.40GHz machine with 512GB of memory per run. We measured runtime using the simulated time metric [20], which is a common metric for simulating parallel runtimes in DCR algorithms [22, 4, 17]. We used the default latency of zero in our experiments.

We evaluated the algorithms on randomly generated feasible CRC problems where we varied the size of the problem instances by increasing the number of agents N from 15 to 25, the domain size of each agent $|D_i|$ from 3 to 10, the graph density p_1—defined as the ratio between the number of constraints and $\binom{N}{2}$—from 0.1 to 0.5, and a parameter that controls the constraint tightness.[13]

Tables 1-3 provide the results of our experiments. Each data point is the median over 50 instances. It is easy to observe the superior performance of D-CRC. DPOP is not competitive with D-CRC at all. The reason is that DPOP does not prune any of the search space and needs to evaluate all constraints and optimize over them even if a large portion of the search space is infeasible. Moreover, D-CRC outperforms even the centralized CRC algorithm by about a factor of 2 across all parameter settings. The reason for the speedup is that D-CRC allows multiple agents to concurrently change their respective values.

One could also possibly use other local search algorithms like MGM [14] or DSA [26] to solve CRC problems. However, unlike D-CRC, the convergence of these algorithms is not guaranteed.

7 Tree Convex Constraints

Tractable classes of tree convex constraints have been identified in [9] using the same notion of *smoothness* but with a *tree-based distance* metric instead of the manhattan distance. The randomized algorithms provided for tree convex constraints in [9] outperform the deterministic algorithms for solving them. Moreover, these randomized algorithms can be generalized to distributed settings in exactly the same way as illustrated in this paper.

[12] We chose to compare against DPOP as its implementation is publicly available.

[13] Varying the constraint tightness was mostly inconsequential for the algorithms.

8 Conclusions

In this paper, we presented a simple polynomial-time randomized algorithm for solving CRC constraints in distributed settings. Because CRC constraints encompass such diverse problems like 2-SAT, implicational constraints, simple temporal problems, restricted disjunctive temporal problems [7], metric temporal problems with domain rules [10], logical filtering and planning problems [12], and spatial reasoning problems [5], our distributed algorithm for solving them has a number of important applications in problem solving for multi-agent systems.

Our algorithm is simple to implement, does not compromise privacy by having agents exchange information about inferred constraints, and does not bear additional overhead for exchanging knowledge in different formats. Empirically, our algorithm outperforms DPOP by a large margin and also performs better than the centralized version. The arguments used in the proofs generalize to tree convex constraints. Further, they provide a handle for analyzing the effect of using randomization in solving DCSPs with the inkling that randomization often helps in multi-agent problem solving.

9 Acknowledgements

The research at USC was supported by NSF under grant number IIS-1319966 and ONR under grant number N00014-09-1-1031. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies or the U.S. government.

References

1. K. Booth and G. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *Journal of Computer and System Sciences*, 13(3):335–379, 1976.
2. Y. Deville, O. Barette, and P. V. Hentenryck. Constraint satisfaction over connected row convex constraints. *Artificial Intelligence*, 109(1–2):243–271, 1999.
3. P. Doyle and J. L. Snell. *Random Walks and Electrical Networks*, volume 22 of *Carus Mathematical Monographs*. Mathematical Association of America, 1984.
4. D. Hatano and K. Hirayama. DeQED: An efficient divide-and-coordinate algorithm for DCOP. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 566–572, 2013.
5. T. K. S. Kumar. On geometric CSPs with (near)-linear domains and max-distance constraints. In *Proceedings of the ECAI Workshop on Modeling and Solving Problems with Constraints*, 2004.
6. T. K. S. Kumar. *Contributions to Algorithmic Techniques in Automated Reasoning About Physical Systems*. PhD thesis, Stanford University, 2005.
7. T. K. S. Kumar. On the tractability of restricted disjunctive temporal problems. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 110–119, 2005.

8. T. K. S. Kumar. On the tractability of smooth constraint satisfaction problems. In *Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR)*, pages 304–319, 2005.
9. T. K. S. Kumar. Simple randomized algorithms for tractable row and tree convex constraints. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 74–79, 2006.
10. T. K. S. Kumar. Tractable classes of metric temporal problems with domain rules. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 847–852, 2006.
11. T. K. S. Kumar, L. Cohen, and S. Koenig. Incorrect lower bounds for path consistency and more. In *Proceedings of the International Symposium on Abstraction, Reformulation and Approximation (SARA)*, 2013.
12. T. K. S. Kumar and S. Russell. On some tractable cases of logical filtering. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 83–92, 2006.
13. T. Leaute, B. Ottens, and R. Szymanek. Frodo: An open-source framework for distributed constraint optimization. In *Proceedings of the International Workshop on Distributed Constraint Reasoning*, 2009.
14. R. Maheswaran, J. Pearce, and M. Tambe. Distributed algorithms for DCOP: A graphical game-based approach. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems (PDCS)*, pages 432–439, 2004.
15. S. Marisetti. Experimental study of algorithms for connected row convex constraints. Master's thesis, Texas Tech University, 2008.
16. R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
17. D. T. Nguyen, W. Yeoh, and H. C. Lau. Distributed Gibbs: A memory-bounded sampling-based DCOP algorithm. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 167–174, 2013.
18. C. Papadimitriou. On selecting a satisfying truth assignment (extended abstract). In *Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS)*, pages 163–169, 1991.
19. A. Petcu and B. Faltings. Dpop: A scalable method for multiagent constraint optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 2005.
20. E. Sultanik, R. Lass, and W. Regli. DCOPolis: A framework for simulating and deploying distributed constraint reasoning algorithms. In *Proceedings of the Workshop on Distributed Constraint Reasoning*, 2007.
21. P. van Beek and R. Dechter. On the minimality and global consistency of row convex constraint networks. *Journal of the ACM*, 42(3):543–561, 1995.
22. W. Yeoh, A. Felner, and S. Koenig. BnB-ADOPT: An asynchronous branch-and-bound DCOP algorithm. *Journal of Artificial Intelligence Research*, 38:85–133, 2010.
23. W. Yeoh and M. Yokoo. Distributed problem solving. *AI Magazine*, 33(3):53–65, 2012.
24. M. Yokoo, editor. *Distributed Constraint Satisfaction: Foundation of Cooperation in Multiagent Systems*. Springer, 2001.
25. M. Yokoo, E. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 614–621, 1992.
26. W. Zhang, G. Wang, Z. Xing, and L. Wittenberg. Distributed stochastic search and distributed breakout: Properties, comparison and applications to constraint optimization problems in sensor networks. *Artificial Intelligence*, 161(1–2):55–87, 2005.
27. Y. Zhang. Fast algorithm for connected row convex constraints. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 192–197, 2007.
28. Y. Zhang and S. Marisetti. Solving connected row convex constraints by variable elimination. *Artificial Intelligence*, 173:1204–1219, 2009.