Removing Redundant Conflict Value Assignments in Resolvent Based Nogood Learning

Jimmy H.M. Lee and Yuxiang Shi
Department of Computer Science & Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
{jlee,yshi}@cse.cuhk.edu.hk

ABSTRACT

Taking advantages of popular Resolvent-based (Rslv) and Minimum conflict set (MCS) nogood learning, we propose two new techniques: Unique nogood First Resolvent-based (UFRslv) and Redundant conflict value assignment Free Resolvent-based (RFRslv) nogood learning. By removing conflict value assignments that are redundant, these two new nogood learning techniques can obtain shorter and more efficient nogoods than Rslv nogood learning, and consume less computation effort to generate nogoods than MCS nogood learning.

We implement the new techniques in two modern distributed constraint satisfaction algorithms, nogood based asynchronous forward checking (AFCng) and dynamic ordering for asynchronous backtracking with nogood-triggered heuristic (ABT-DOng). Comparing against Rslv and MCS on random distributed constraint satisfaction problems and distributed Langford's problems, UFRslv and RFRslv are favourable in number of messages and NCCCSOs (nonconcurrent constraint checks and set operations) as metrics.

1. INTRODUCTION

In distributed reasoning, the variables, domains and constraints are distributed among a set of agents. To solve the problem, the agents communicate with each other by sending messages. Unlike centralized constraint solving, the efficiency of a distributed constraint satisfaction algorithm depends on not only the computation cost in each agent, but also the communication cost between agents [5]. Usually the additional communication cost of sending one message is much more than one internal constraint check in an agent [9]. Yeoh et al. [11] simulate multi-agent systems with the assumption that the cost of sending a message in fast communication is 0 and the cost of sending one message in slow communication is 1000 times of the cost of one constraint check in terms of runtime. It is thus desirable to pay more attention to communication cost in solving algorithms, especially in slow communications.

Many distributed systematic search-based distributed constraint satisfaction algorithms are nogood-based, such as *asynchronous weak-commitment search algorithm* (AWC) [12], *nogood-based asynchronous forward checking* (AFCng) [8] and *dynamic ordering for asynchronous backtracking with nogood-triggered heuristic* (ABT-DOng) [13]. A nogood is made of a subset of the inconsistent current partial assignments obtained by an agent. It indicates the value assignments in this nogood cannot be involved in any solution simultaneously. We can regard a nogood as a new constraint generated during search that is not explicitly stated in original problem. Thus, we refer to making nogoods as *nogood learning* [4].

Modern nogood learning techniques include *resolvent-based* (Rslv) nogood learning [4, 1] and *minimum conflict set* (MCS) nogood learning [6]. The first is also called the *highest possible lowest variable* (HPLV) heuristic [1] which is a heuristic used mainly to select an appropriate nogood of each removed value for generating combined nogoods. This technique ensures that the generated nogood is short each time a wipe-out occurs, but cannot guarantee this nogood is the shortest. At the same time, the variables related to the value assignments in the nogood can be relatively high according to the variable order [1]. Then, the algorithm can do earlier backtrack to reduce search. Rslv nogood learning is widely used in distributed constraint satisfaction algorithms [12, 8, 13].

MCS nogood learning generates as nogood a *minimum conflict set*, which is the smallest subset of *current partial assignments* that cause a deadend. The shorter the nogood, the more search space it can prune. However, the cost of identifying such a set is usually very high [6, 4]. Usually, distributed satisfaction algorithms would not choose it as the technique to do nogood learning.

By doing more internal reasoning in agents, we may get more efficient nogoods which can prune more search space, and the large search space pruning can cut down communication cost substantially. However, too much internal reasoning may slow down the whole solving process. There is a trade-off between the computation effort and the quality of nogoods.

We observe that the nogoods generated by Rslv nogood learning may have redundant conflict value assignments, which are not needed. Thus, we propose efficient ways to remove some redundant conflict value assignments from nogoods generated by Rslv nogood learning using properties of a special kind of nogoods, unique nogoods, in our *Unique nogood First Resolvent-based (UFRslv) nogood learning*. Continuing to remove all redundant conflict value assignments using generate-and-test, we get *Redundant conflict value assignment Free Resolvent-based (RFRslv) nogood learning*. We implement these two techniques in AFCng [8] and ABT-DOng [13] on the DisChoco platform [2], and show formally that both UFRslv and RFRslv generate shorter nogood than Rslv does, and have less computational complexity than MCS. We compare our two techniques against Rslv and MCS nogood learning on uniform random distributed constraint satisfaction problems and distributed Langford's problems using #msg (number of messages) and NCCCSOs (Non-concurrent constraint checks and Set Operations) as metrics. Experiments demonstrate that the two new techniques can save substantial number of messages and NCCCSOs comparing to Rslv nogood learning. And in most cases, they can

Appears in: *Alessio Lomuscio, Paul Scerri, Ana Bazzan, and Michael Huhns (eds.), Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2014), May 5-9, 2014, Paris, France.*

also beat MCS nogood learning in terms of number of messages and save tremendous NCCCSOs.

2. PRELIMINARIES

In this section, we introduce the definition of distributed constraint satisfaction problems, the idea of nogoods and two nogood learning techniques (MCS and Rslv nogood learning).

2.1 Basic Definitions

A *distributed constraint satisfaction problem* (DisCSP) $\mathcal{P}$ is a tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \phi)$ where $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of variables, $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a set of finite domains for each of the variables in $\mathcal{X}$, $\mathcal{C}$ is a set of constraints and $\mathcal{A} = \{a_1, \ldots, a_k\}$ is a set of (distributed) agents and ϕ is a mapping: $\mathcal{X} \mapsto \mathcal{A}$ which maps each variable in $\mathcal{X}$ to an agent in $\mathcal{A}$. Usually, the searching algorithm needs to define a variable order, such as lexicographic order, among $\mathcal{X}$. We use *priority* ($prio(x_i)$) to indicate one variable's (x_i's) order. If variable x_i has higher (lower) priority than x_j in the variable order, i.e., x_i is assigned before (after) x_j, then we have $prio(x_i) > prio(x_j)$ ($prio(x_i) < prio(x_j)$). We denote $x_i = v_i$ a *value assignment* assigning value $v_i \in D_i$ to variable x_i, and the value assignments $l \equiv x_1 = v_1 \wedge x_2 = v_2 \wedge \cdots \wedge x_n = v_n$ is a *complete assignment* on variables in $\mathcal{X}$. *Partial assignments* $l(S)$ is a projection of l onto variables in $S \subseteq \mathcal{X}$. Each constraint $C_S \in \mathcal{C}$ is a constraint over a set $S = \{x_{s_1}, x_{s_2}, \ldots, x_{s_k}\}$ of variables. It is specified by a set of tuples (partial value assignments $l(S)$) allowed to be assigned to S or an expression which states the condition whether a tuple is allowed.

Note that in DisCSPs, variables and constraints are distributed among agents. An agent a_j holds a variable x_i iff $\phi(x_i) = a_j$, and a_j holds a constraint C_S iff $\exists x_i \in S \; and \; \phi(x_i) = a_j$. In this paper, we suppose each agent controls one variable and each variable belongs to one agent without losing any generality. In most distributed constraint satisfaction algorithms, each agent maintains a store to hold the partial assignments it gets to know. And in nogood based distributed constraint satisfaction algorithms, each agent a_i also has a *nogoodstore* N_i to hold the nogoods found during search.

2.2 Nogood Learning

Chronological backtrack tree search is a basic systematic search for solving distributed constraint satisfaction problems. It traverses the search tree of possible assignments in a depth-first left-to-right manner. In general, the search space is exponential. So, we usually use some techniques to prune the search space to speed up the search procedure. Nogood learning is a kind of technique to create explicit disallowed partial assignments and prune search space.

2.2.1 Nogood

A *nogood* is the negation of a conjunction of value assignments which are disallowed by the problem constraints. The *length* of a nogood is the number of value assignments contained in this nogood. We also call each value assignment contained in a nogood a *conflict value assignment*. For example, the nogood, $\neg[x_i = v_i \wedge x_j = v_j \wedge \cdots \wedge x_k = v_k]$, means that the assignments it contains are not simultaneously allowed since they cause unsatisfiability. This is the most direct and intuitive representation of nogood. Suppose we have a nogood $ng \equiv \neg[l(S)]$ and a value assignment $x_i = v_i$, if $x_i = v_i$ is in $l(S)$, then we say value assignment $x_i = v_i$ is *contained* in nogood ng. A *higher nogood* [4] of x_i is the nogood $\neg[l(S)]$ where $\forall x_k \in S - \{x_i\}$, $prio(x_k) > prio(x_i)$.

We can also represent nogoods with equivalent implication representation [10]. A *directed nogood* ng ruling out value v_k from the initial domain of variable x_k is an implication of the form $x_{s_1} = v_{s_1} \wedge \cdots \wedge x_{s_m} = v_{s_m} \rightarrow x_k \neq v_k$, meaning that the assignment $x_k = v_k$ is inconsistent with $x_{s_1} = v_{s_1} \wedge \cdots \wedge x_{s_m} = v_{s_m}$. When a nogood, ng, is represented as an implication, the *left hand side* ($lhs(ng) \equiv x_{s_1} = v_{s_1} \wedge \cdots \wedge x_{s_m} = v_{s_m}$) and the *right hand side* ($rhs(ng) \equiv x_k \neq v_k$) are defined with respect to the position of $\rightarrow$. Many look-back distributed constraint satisfaction algorithms, such as AFCng [8], store directed nogoods in each agent a_i's nogoodstore N_i as justifications of value removals.

Suppose we have two directed nogoods $ng_1 \equiv x_{s_1} = v_{s_1} \wedge \cdots \wedge x_{s_m} = v_{s_m} \rightarrow x_k \neq v_k$ and $ng_2 \equiv x_{t_1} = v_{t_1} \wedge \cdots \wedge x_{t_m} = v_{t_n} \rightarrow x_k \neq v_k$, ng_1 is *higher than* ng_2, if $\exists w \; such \; that \; \forall i > w, prio(x_{s_i}) = prio(x_{t_i}) \; and \; prio(x_{s_w}) > prio(x_{t_w})$, with the assumption that $prio(x_{s_1}) > prio(x_{s_2}) > \ldots > prio(x_{s_m})$ and $prio(x_{t_1}) > prio(x_{t_2}) > \ldots > prio(x_{t_n})$. Suppose an agent has got partial assignments $l(S)$, a directed nogood ng_i is *consistent*, if $\forall x_t = v_t \in lhs(ng_i)$ is contained in $l(S)$ or $l(S)$ does not contain value assignment of x_t. And we say a nogood $ng \equiv \neg[l(S)]$ *involves* a directed nogood ng_i, if $lhs(ng_i) \subseteq l(S)$. When an agent generates or resolves stored nogoods to generate combined nogoods, these stored nogoods should be consistent.

2.2.2 Nogood Learning Techniques

Nogood learning techniques can help agents to generate nogoods to eliminate inconsistent values. When an agent detects a *deadend* which means a variable's domain is wiped out by current value assignments, it will generate a *combined nogood* using a nogood learning technique by resolving all or a subset of consistent nogoods (with directed representation) which represent the removals of values in nogoodstore together. We *resolve* a set of directed nogoods, $\{ng_1, ng_2, \ldots, ng_k\}$, to get a new implication $lhs(ng_1) \wedge \cdots \wedge lhs(ng_k) \rightarrow rhs(ng_1) \wedge \cdots \wedge rhs(ng_k)$. When we generate combined nogoods, we must guarantee that the combined nogood can represent the deadend. If we have a combined nogood, $ng \equiv \neg[l(S)]$, and $\forall v_t \in D_i$ (D_i is the domain of x_i), $x_i = v_t$ conflicts with the value assignments $l(S)$, then we call ng a *complete nogood* for D_i's wiping out.

Suppose variable x_i has met a deadend with all domain elements pruned, and the nogoodstore N_i contains at least $|D_i|$ directed nogoods to explain the removal of values in D_i. The most intuitive way to generate a complete nogood is to pick an arbitrary directed nogood ng_k for each $v_k \in D_i$ and resolve these nogoods $\{ng_k | v_k \in D_i\}$ together. Then, the agent can get an implication $lhs(ng_1) \wedge \ldots \wedge lhs(ng_{|D_i|}) \rightarrow rhs(ng_1) \wedge \ldots \wedge rhs(ng_{|D_i|})$. The right hand side of this implication is always *false*, because the domain is wiped out. We get $lhs(ng_1) \wedge \ldots \wedge lhs(ng_{|D_i|}) \rightarrow false$, which is equivalent to $\neg[lhs(ng_1) \wedge \ldots \wedge lhs(ng_{|D_i|}) \wedge true] \equiv \neg[lhs(ng_1) \wedge \ldots \wedge lhs(ng_{|D_i|})]$. The new combined nogood is thus $\neg[lhs(ng_1) \wedge \ldots \wedge lhs(ng_{|D_i|})]$. Theorem 1 guarantees that the combined nogood generated in this way is complete.

THEOREM 1. *Suppose variable x_i has met a deadend with all domain elements pruned, and the nogoodstore N_i contains at least $|D_i|$ directed nogoods explaining the removal of values in D_i. Suppose we pick an arbitrary directed nogood ng_k for each $v_k \in D_i$. Then $ng \equiv \neg[lhs(ng_1) \wedge \ldots \wedge lhs(ng_{|D_i|})]$ is a complete combined nogood for x_i's deadend.*

PROOF. When we resolve all the arbitrarily chosen directed nogoods, $\{ng_k | v_k \in D_i\}$, together, we can get an implication $lhs(ng_1) \wedge \ldots \wedge lhs(ng_{|D_i|}) \rightarrow rhs(ng_1) \wedge \ldots \wedge rhs(ng_{|D_i|})$. We can see that the left hand side of this implication implies the removal of all the values in D_i. So, $ng \equiv \neg[lhs(ng_1) \wedge \ldots \wedge lhs(ng_{|D_i|})]$ is a complete combined nogood for x_i's deadend. □

Although resolving arbitrarily chosen nogoods for each value

can make a complete combined nogood, the quality of this combined nogood varies. A method to guide us in choosing proper nogoods is needed. We focus on Rslv and MCS nogood learning.

Rslv Nogood Learning.

Rslv Nogood Learning is also called the *Highest Possible Lowest Variable* (HPLV) heuristic [1], which is widely used in many DisCSP algorithms. Rslv nogood learning can be considered as a heuristic to guide us choosing an appropriate nogood to represent the removal of a particular value, if this value is pruned by more than one nogood. The nogoods made with this technique is equivalent to a *resolvent* in propositional logic [4]. Suppose an agent a_i has a variable x_i with a domain D_i, and every possible value in D_i violates some higher nogoods consistent with the current partial assignments. Algorithm 1 describes the procedure of Rslv nogood learning. The inputs of function HPLV are variable x_i's domain D_i and the nogoodstore N_i in agent a_i. The agent selects one nogood for each value $v \in D_i$ (HPLV line 5) following the rules below [4]:

1. Select the smallest nogood among the higher nogoods of x_i pruning each v.
2. Ties are broken by selecting the one which is the highest.

After choosing a nogood for each $v \in D_i$, the agent puts the left side of this nogood into the combined nogood (HPLV line 6).

Algorithm 1 HPLV Heuristic

```
1: procedure HPLV(D_i, N_i)
2:     combinedNogood ← ∅
3:     for all v_j ∈ D_i do
4:         nogoods ← nogoods of v_j in N_i;
5:         ng ← shortest and highest nogood in nogoods
6:         combinedNogood ← combinedNogood ∪ lhs(ng)
7:     end for
8: return combinedNogood
9: end procedure
```

An agent selects the smallest nogood since it wants to prune more search space. Agent chooses the highest nogood if there are ties for the smallest nogood because higher nogood includes variables with higher priority according to the variable order. Upon backtracking, with this shorter and higher nogood, the agent may prune more search space and backtrack to higher place.

This is a greedy algorithm to generate nogoods. With this technique, we cannot guarantee the nogoods to be the shortest, i.e., the nogoods are not made of minimum subsets of the inconsistent value assignments which can lead to the deadend. Rslv nogood learning cannot guarantee the nogoods to be the highest.

In fact, if we use this nogood learning technique in a nogood based distributed constraint satisfaction algorithm, we only need to store one nogood for each removed value. When an agent gets a new nogood for one removed value, it will choose the more appropriate nogood between the newly received nogood and the nogoods already stored in the agent according to the HPLV heuristic. This can be an online algorithm with linear time complexity according to the number of nogoods generated during search. The space complexity is also linear according to the domain size. When an agent detects a deadend, it just needs to resolve the nogoods held in nogoodstore to generate a combined nogood. This procedure will take linear time according to the size of variable's domain.

EXAMPLE 1. *In a DisCSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \phi)$, there are eight variables, $\{x_1, x_2, \ldots, x_8\}$. The variable order is $prio(x_1) > prio(x_2) > \cdots > prio(x_8)$. Agent a_8 controls variable x_8. And there are 6 values in the variable's domain, $\{1, 2, 3, 4, 5, 6\}$. When agent a_8 meets a deadend, i.e., all the values in the domain are pruned by the current partial value assignments. Suppose the nogoods in the nogoodstore is as: $ng_1 \equiv x_2 = v_2 \rightarrow x_8 \neq 1$, $ng_2 \equiv x_3 = v_3 \rightarrow x_8 \neq 2$, $ng_3 \equiv x_1 = v_1 \wedge x_2 = v_2 \rightarrow x_8 \neq 3$, $ng_4 \equiv x_2 = v_2 \wedge x_3 = v_3 \rightarrow x_8 \neq 3$, $ng_5 \equiv x_4 = v_4 \rightarrow x_8 \neq 4$, $ng_6 \equiv x_2 = v_2 \wedge x_5 = v_5 \rightarrow x_8 \neq 4$, $ng_7 \equiv x_3 = v_3 \wedge x_7 = v_7 \rightarrow x_8 \neq 4$, $ng_8 \equiv x_2 = v_2 \wedge x_5 = v_5 \rightarrow x_8 \neq 5$, $ng_9 \equiv x_2 = v_2 \wedge x_3 = v_3 \wedge x_7 = v_7 \rightarrow x_8 \neq 5$, $ng_{10} \equiv x_6 = v_6 \rightarrow x_8 \neq 6$, $ng_{11} \equiv x_3 = v_3 \wedge x_7 = v_7 \rightarrow x_8 \neq 6$. When an agent generates new combined nogood using Rslv nogood learning, it will choose ng_1 (ng_2) for removed value 1 (2), because there is no other nogood to prune value 1 (2). And it chooses ng_3 for removed value 3, because ng_3 is higher than ng_4. This agent uses ng_5 (ng_8, ng_{10}) to represent the removal of value 4 (5, 6), because ng_5 (ng_8, ng_{10}) is shorter than ng_6, ng_7 (ng_9, ng_{11}). Then, we can combine the left hands of ng_1, ng_2, ng_3, ng_5, ng_8, and ng_{10} together to get new combined nogood $ng \equiv \neg[x_1 = v_1 \wedge x_2 = v_2 \wedge x_3 = v_3 \wedge x_4 = v_4 \wedge x_5 = v_5 \wedge x_6 = v_6]$.*

MCS Nogood Learning.

MCS nogood learning is proposed by Mammen et al. [6]. An agent identifies the minimum conflict set from its stored current partial assignments and makes a nogood with it. The minimum conflict set is the smallest subset of the stored current partial assignments that causes a deadend. Mammen et al. claim this nogood can be the most effective one because it can prune the largest portion of the search space. The idea of Hirayama et al. [4] is to generate a big nogood first and then test whether a subset of the nogood is enough to generate deadend or not from larger subsets to smaller subsets. It will explore all the combinations of each removed value's nogoods and find the minimum conflict set. However, the complexity of generating the minimum conflict set is exponential. The benefit we can get from stronger pruning may not cover the cost we spend in finding the minimum conflict set. There is a trade-off between the computation effort and the quality of nogood. The smaller the nogoods, the more computation effort they cost, but the more powerful they are in problem solving.

3. UFRSLV NOGOOD LEARNING

Before we introduce our algorithm, some concepts and terms should be defined. When we solve a DisCSP $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \phi)$ using a nogood-based algorithm and there is a nogood $ng \equiv \neg[x_{s_1} = v_{s_1} \wedge \cdots \wedge x_{s_{i-1}} = v_{s_{i-1}} \wedge x_{s_i} = v_{s_i} \wedge x_{s_{i+1}} = v_{s_{i+1}} \wedge \cdots \wedge x_{s_m} = v_{s_m}]$. The value assignment $x_{s_i} = v_{s_i}$ is a *redundant* conflict value assignment in nogood ng, if $x_{s_1} = v_{s_1} \wedge \cdots \wedge x_{s_{i-1}} = v_{s_{i-1}} \wedge x_{s_{i+1}} = v_{s_{i+1}} \wedge \cdots \wedge x_{s_m} = v_{s_m}$ is inconsistent. That is to say $\neg[x_{s_1} = v_{s_1} \wedge \cdots \wedge x_{s_{i-1}} = v_{s_{i-1}} \wedge x_{s_{i+1}} = v_{s_{i+1}} \wedge \cdots \wedge x_{s_m} = v_{s_m}]$, obtained by just removing $x_{s_i} = v_{s_i}$ from ng, is still a nogood. From the definition, we can easily get the following two theorems.

THEOREM 2. *The nogoods generated by MCS nogood learning technique contains no redundant conflict value assignments.*

PROOF. Suppose there is a redundant conflict value assignment $x_i = v_i$ in a nogood ng which is generated by MCS nogood learning. Since this value assignment is redundant, we can remove it from ng and the result is also a nogood. Obviously this nogood is also complete to represent the deadend, but has a smaller number of conflict value assignments than ng has. We have a contradiction and ng is the shortest nogood. □

THEOREM 3. *The length of a nogood containing no redundant conflict value assignments is not necessarily minimum.*

PROOF. We can continue with Example 1. Nogood $\neg[x_2 = v_2 \wedge x_3 = v_3 \wedge x_5 = v_5 \wedge x_6 = v_6]$ contains no redundant conflict value assignments. However, the minimum conflict set is $x_2 = v_2 \wedge x_3 = v_3 \wedge x_7 = v_7$. Result follows. □

An important observation is that some nogoods must be involved in combined nogoods to represent the deadend met by some variable, if we want the combined nogoods to be complete. We can put this kind of nogoods into the combined nogoods early and directly. We suppose an agent a_i detects a deadend π. Directed nogood ng_i in a_i's nogoodstore is a *necessary nogood*, if all complete nogoods which can represent the deadend π must involve ng_i. And a *unique nogood* of a removed value v is the only nogood which can justify the removal of v, i.e., value v can be only removed by the *unique nogood*. We continue with Example 1. We can see that ng_1, ng_2 and ng_4 are *necessary nogoods*. Since ng_1 and ng_2 are *unique nogoods*, i.e., they are the only nogoods to prune values 1 and 2 respectively. If we remove ng_1 (ng_2) from any complete nogood ng, value 1 (value 2) cannot be pruned, i.e., the resultant combined nogood is not complete. So, ng_1 and ng_2 are *necessary nogoods*. Since ng_4's left hand side is just made of the left hand side of necessary nogoods ng_1 and ng_2, ng_4 should also be involved in every complete combined nogood implicitly.

In Theorem 4, we give the relation between *necessary nogoods* and *unique nogoods*. We can get Theorems 4 easily from the definitions of necessary nogoods and unique nogoods.

THEOREM 4. *A unique nogood must be a necessary nogood.*

Necessary nogoods have such a good property that they should be involved in any complete combined nogoods. From Theorem 4, we can easily obtain the fact that unique nogoods must be involved in any complete combined nogood. And these unique nogoods are easy to find, i.e., an agent just needs to scan the removed values and find the only nogood of a particular value.

From Example 1, we can find that the combined nogood generated by Rslv nogood learning may contain some redundant conflict value assignments. For example, we will choose ng_3 rather than ng_4, because ng_3 is higher than ng_4. However, because ng_1 and ng_2 are unique nogoods, they must be in the combined nogood. When the combined nogood involves ng_1 and ng_2, the combined nogood involves ng_4 implicitly, i.e., we do not need to involve ng_3 in the combined nogood any more. Thus, with Rslv nogood learning, we may involve redundant conflict value assignments in a combined nogood. If we can spend some effort to remove some of these redundant value assignments, the new combined nogood will be shorter and pruning power can become stronger. Based on this idea, we propose a new nogood learning technique, called *Unique First Resolvent Based* (UFRslv) nogood learning, which removes some redundant conflict value assignments with little effort.

Algorithm 2 gives UFRslv nogood learning. The inputs are the original domain D_i of variable x_i, and the nogoodstore N_i in agent a_i. Initially empty, the $combinedNG$ is the new combined nogood we want to generate (`UFRslv` line 2). And $Values$ contains the removed values which we need to consider, initially the original domain (`UFRslv` line 3).

In the first step, agent finds all unique nogoods and puts the left hand side of these unique nogoods into an intermediate combined nogood (`UFRslv` line 4 to 7). We will not consider these unique nogoods and their pruning values in remaining steps. Here, we just remove these pruned values from $Values$. Continuing to explain

Algorithm 2 UFRslv Nogood Learning

```
1: procedure UFRsLv(D_i, N_i)
2:     combinedNG ← ∅;
3:     Values ← D_i;
4:     for all v_i ∈ Values has unique nogood ng do
5:         combinedNG ← combinedNG ∪ lhs(ng);
6:         Values ← Values − v_i;
7:     end for

8:     for all v_i ∈ Values do
9:         if v_i is pruned by l(T) in combinedNG then
10:            Values ← Values − v_i;
11:        end if
12:    end for

13:    combinedNG ← combinedNG ∪ HPLV(Values, N_i);
14:    return combinedNG;
15: end procedure
```

with Example 1, when we find the unique nogood ng_1 and ng_2, we will combine the left hand side of ng_1 and ng_2 to get an intermediate combined nogood $\neg[x_3 = v_3 \wedge x_4 = v_4]$

In the second step, we try to prune values contained in $Values$ using the value assignments $l(T)$ in the intermediate combined nogood just generated. If a value is pruned by $l(T)$, the value and the nogoods corresponding to this value will not be considered in remaining steps (`UFRslv` line 8 to 12). For example, we can use $x_3 = v_3 \wedge x_4 = v_4$ to prune value 3 of x_8. Then, we remove value 3 from $Values$ and do not consider ng_3 and ng_4 in remaining steps.

After these two steps, we have already constructed some parts of the combined nogood. Then, the agent will use Rslv nogood learning to choose nogoods for the remaining values in $Values$ (`UFRslv` line 13) and combine them into $combinedNG$. In Example 1, with UFRslv nogood learning, we can get the combined nogood as $\neg[x_2 = v_2 \wedge x_3 = v_3 \wedge x_4 = v_4 \wedge x_5 = v_5 \wedge x_6 = v_6]$. We can see that this nogood is shorter than the nogood generated by Rslv nogood learning, i.e., it does not contain $x_1 = v_1$.

From the procedure of UFRslv nogood learning, we can obtain Theorem 5.

THEOREM 5. *The length of combined nogoods generated by UFRslv nogood learning is shorter than or equal to the length of the combined nogoods generated by Rslv nogood learning.*

PROOF. The difference between the UFRslv and Rslv nogood learning techniques is that we do two additional steps before we do Rslv nogood learning in UFRslv nogood learning. In the first step of UFRslv nogood learning, it will involve all unique nogoods in the intermediate combined nogood. From Theorem 4 and the definition of necessary nogoods, these unique nogoods should be involved in the combined nogood generated by Rslv nogood learning as well. In addition, when we do the second step in UFRslv nogood learning, we will try to ignore some removed values, which can be already pruned by the value assignments in the intermediate combined nogood, in following Rslv nogood learning. Thus, we have removed some unnecessary nogoods in the second step, and the nogoods generated by UFRslv nogood learning should be shorter than or equal to those generated by Rslv nogood learning. □

Unlike Rslv nogood learning, UFRslv nogood learning needs to store all consistent nogoods of each removed value in the nogoodstore since we do not know which nogoods we will choose to represent the removal of values in UFRslv nogood learning.

From the procedure of UFRslv nogood learning, we can obtain Theorems 6 and 7. Completeness is easy to check since UFRslv removes only redundant conflict value assignments.

THEOREM 6. *The nogood generated by UFRslv nogood learning is complete.*

THEOREM 7. *Suppose variable x_i has met a deadend with its domain D_i being wiped out, and there are N consistent directed nogoods stored in nogoodstore. The time complexity of UFRslv nogood learning is $O(|D_i| + N)$.*

PROOF. In the first step of UFRslv nogood learning, we need to scan the original domain to find unique nogoods and put the left hand side of these unique nogood into an intermediate combined nogood. This takes linear time according to the size of domain. Then, we try to use the value assignments in the implication to prune domain values. In order to prune one value, we just need to test whether there exists a nogood pruning the value involved in the implication. This step takes linear time according to the number of consistent nogoods in nogoodstore. And in the following step, our method just do Rslv nogood learning. We need to scan the nogoodstore to choose one nogood to represent the removal of each value. This takes linear time according to the number of nogoods in nogoodstore. Finally, we will resolve these nogoods to get the final combined nogood. This also costs linear time according to domain size. So, the time complexity of UFRslv nogood learning is $O(|D_i| + N)$. □

Compared with MCS nogood learning which has exponential complexity on the number of consistent nogoods in nogoodstore, our technique is expected to save a lot of computation effort. Compared with Rslv nogood learning, UFRslv is more attractive especially in slow communication since the additional computation is only internal reasoning and the resultant large search space pruning can cut down communication cost substantially.

4. RFRSLV NOGOOD LEARNING

Although nogoods from UFRslv nogood learning contain less redundant conflict value assignments than those generated by Rslv nogood learning, the resultant nogoods can still contain redundant conflict value assignments. We propose *Redundant Conflict Value Assignment Free Resolvent Based* (RFRslv) nogood learning. The combined nogood generated by this technique is guaranteed not to contain any redundant conflict value assignments. Then, RFRslv nogood learning may get shorter combined nogoods than UFRslv nogood learning by spending some more effort to remove redundant conflict value assignments. We expect this technique can gain more benefits from the shorter nogoods.

The idea of RFRslv nogood learning consists of two steps: (a) generate a combined nogood using UFRslv nogood learning and (b) check whether each conflict value assignment in the combined nogood is redundant, and if so, we will remove this value assignment from the combined nogood.

The RFRslv nogood learning procedure is given in Algorithm 3. The inputs are the original domain D_i of variable x_i, and the nogoodstore N_i in agent a_i.

In the first step, we generate a combined nogood using UFRslv nogood learning (`RFRslv` line 2). And then, we need to remove all the redundant conflict value assignments. The most intuitive way is to remove each conflict value assignment temporally from the combined nogood and check whether the resultant nogood is still complete. If it is the case, the removed conflict value assignment is redundant and we can leave it out.

Algorithm 3 RFRslv Nogood Learning

```
1:  procedure RFRslv(D_i, N_i)
2:      combinedNG ← UFRslv(Values, N_i);
3:      Values ← D(x_i);
4:      shouldIn ← ∅;
5:      for all v_i ∈ Values has unique nogood ng do
6:          shouldIn ← shouldIn ∪ lhs(ng)
7:          Values ← Values − v_i;
8:      end for
9:      for all x_m = v_m in combinedNG do
10:         if (x_m = v_m) ∈ shouldIn then
11:             continue
12:         end if
13:         tempNG ← combinedNG − (x_m = v_m)
14:         if tempNG can wipe out x_i's domain then
15:             combinedNG ← tempNG;
16:         end if
17:     end for
18:     return combinedNG;
19: end procedure
```

We speed up the procedure with the idea of necessary and unique nogoods. We can see that unique nogoods should be involved in the combined nogood, and the value assignments contained in unique nogoods must be contained in the combined nogood. Thus, we do not need to check these conflict value assignments. At the same time, when we check whether the resultant nogood is complete, there is no need to check whether the values with unique nogoods can be pruned by the value assignments in the intermediate combined nogood. The reason is that the unique nogoods must be involved in the combined nogood.

In our algorithm, we check each remaining conflict value assignment starting with the value assignment of the variable with the lowest priority, because we want to get as high a nogood as possible. With higher nogood, an agent can backtrack to a higher variable and the search procedure can be more efficient.

In the second step, we first remove the values with unique nogoods from the copy of original domain $Values$ and put the left hand side of these unique nogoods into $shouldIn$ (`RFRslv` line 5 to 8). Then, we continue to check whether a conflict value assignment ($x_m = v_m$) is redundant or not from lower variables to higher variables (`RFRslv` line 9 to 17). If $x_m = v_m$ is involved in $shouldIn$, it means this value assignment should be in the combined nogood. Then, we continue to check other conflict value assignments (`UFRslv` line 10). Otherwise, we will remove (denoted by "−") this value assignment from the combined nogood tentatively and test whether the remaining conflict value assignments can prune all the values in $Values$ or not (`UFRslv` line 14). If they can prune all the values in $Values$, it means the conflict value assignment ($x_m = v_m$) is redundant. We will remove this value assignment and continue to check other value assignments. Otherwise, this value assignment is not redundant. We will restore this value assignment back into the combined nogood.

Continuing with Example 1, we get combined nogood $ng \equiv \neg[x_2 = v_2 \wedge x_3 = v_3 \wedge x_4 = v_4 \wedge x_5 = v_5 \wedge x_6 = v_6]$ using UFRslv nogood learning. Then, we continue to test whether a conflict value assignment in ng is redundant. We start to check from right to left (the variables are lexicographically ordered). When we check value assignment $x_4 = v_4$, we can find the remaining value assignments can also represent the deadend. Thus, $x_4 = v_4$ is redundant and can

be removed from ng. After we check all the value assignments, we can get the final combined nogood $\neg[x_2 = v_2 \wedge x_3 = v_3 \wedge x_5 = v_5 \wedge x_6 = v_6]$.

From the procedure of RFRslv nogood learning, we can obtain the following theorems. Completeness and nogood length comparison are straightforward to check.

THEOREM 8. *The nogoods generated by RFRslv nogood learning is complete.*

THEOREM 9. *The nogoods generated by RFRslv nogood learning is shorter than or equal to those generated by UFRslv nogood learning.*

THEOREM 10. *Suppose there are n variables, variable x_i has met a deadend with its domain D_i being wiped out, and N consistent directed nogoods stored in nogoodstore. The time complexity of RFRslv nogood learning is $O(|D_i| + nN)$.*

PROOF. In the first step of RFRslv nogood learning, we will generate a combined nogood using UFRslv nogood learning with $O(|D_i| + N)$ time complexity. And in the worst case, all value assignments of variables except x_i are contained in the combined nogood. We need to check whether each value assignment in this nogood is redundant. And in each check, we need to test whether there exists a nogood involved in the intermediate nogood pruning each value. So, this will take $O(nN)$ time complexity. Thus, the complexity of RFRslv nogood learning is $O(|D_i| + nN)$. □

This new technique is expected to save much computation effort as compared with MCS nogood learning and generate more powerful nogoods than Rslv nogood learning.

5. EXPERIMENT

We have implemented UFRslv and RFRslv nogood learning in AFCng [8] and ABT-DOng [13] and compared these two new techniques with Rslv and MCS nogood learning based on uniform random constraint satisfaction problems and distributed Langford's problems. We evaluate the performance of the algorithms by communication load and runtime. Communication load is measured by the total number of messages (#msg) exchanged among agents during algorithm execution. We use modified version of non-concurrent constraint checks [7], non-concurrent constraint checks and set operations (NCCCSOs), as the runtime metric, since our nogood learning techniques consist of many set operations, such as combining two sets (nogoods) together and checking whether one set (nogood) is a subset of another (nogood). When we compare different algorithms in real distributed multi-agent systems, these operations may take considerable time. To be fair in our comparison of various techniques in the simulator, we consider also the number of set operations in the runtime metric.

We assume each set operation taking similar time as a constraint check. We verified our assumption by testing set operations and constraint checks (only considering table constraint) in Java. DisChoco 2.0 deals with table constraints using the Java BitSet class. When checking whether two value assignments satisfy a constraint, we need to calculate the offset from these assignments and get the corresponding truth value from the BitSet. In our testing implementation, we also use the BitSet class to represent sets and do set operations with builtin methods. We tested three set operations: combining two sets, subtracting one set from another, and checking whether one set is a subset of another. We performed each kind of operations 100000 times. On average, 100000 set operations and 100000 constraint checks took about 23 ms and 20 ms respectively.

We consider two kinds of communication: fast and slow. For fast communication, we suppose the communication cost of sending one message takes the same time as doing one constraint check. For slow communication, we assume the communication cost is 1000 times of that doing one constraint check.

All experiments are performed using DisChoco 2.0 [2]. Each problem instance is solved by each algorithm 10 times. Computation timeouts if the algorithm consumes more than 10^8 NCCCSOs with fast communication (since NCCCSOs are higher in slow communication). We report average of all measures in separate graphs. Since we cannot report all results due to space limitation, we provide all data in the following link:

`http://www.cse.cuhk.edu.hk/~jlee/rand.pdf`
`http://www.cse.cuhk.edu.hk/~jlee/lang.pdf`

5.1 Uniform Binary Random DisCSPs

Uniform binary random DisCSPs [10] are characterized by $\langle n, d, p_1, p_2\rangle$, where n is the number of agents and variables (each agent holds one and only one variable), d is the number of values in each domain, p_1 is the density defined as the ratio of existing binary constraints and p_2 is the constraint tightness defined as the ratio of forbidden value pairs. In the experiment, we fix the number of agents to 15 and the size of each variable's domain to 20. We vary the tightness from 0.1 to 0.9 by steps of 0.1. Density is varied from 0.2 to 0.8 by steps of 0.1. For each pair of fixed density and tightness (p_1, p_2), we generate 25 instances.

Because of the lack of space, we only give some results which are the most representative. Figure 2(a) - 2(c) and Figure 2(d) - 2(f) give results of AFCng and ABT-DOng respectively. We present results for constraint tightness of 0.6 and vary density from 0.2 to 0.8. In the figures, we use Rslv, MCS, UFRslv and RFRslv to represent corresponding nogood learning techniques. NCCCSO-1 and NCCCSO-1k denote the NCCCSOs taken by the algorithm in fast and slow communication respectively. The analysis we give in the following is based on *all* tightness used in the experiments.

Overall speaking, UFRslv and RFRslv exhibit similar patterns, outperforming Rslv in most cases, especially when the problem is difficult. And these two techniques can beat MCS in terms of number of messages sometimes and always save substantial NCCCSOs.

With AFCng, UFRslv and RFRslv can save up to 60% and 61% of number of messages and up to 55% and 44% of NCCCSOs in fast network when compared with Rslv. With slow communication, UFRslv and RFRslv can save up to 59% and 60% of NCCCSOs against Rslv. Since our new nogood learning takes more local reasoning, it can get shorter and more efficient nogoods, thus resulting in less messages. In a slow network, sending a message is expensive, making our methods useful and significant.

Compared with MCS, UFRslv and RFRslv are competitive in terms of number of messages, but we cannot draw a solid conclusion on which is better. Although our nogoods are not the shortest, we try to make higher nogoods. These higher nogoods allow search to backtrack to a higher node. This may help us do searching in a more efficient way. On the other hand, MCS consumes many more NCCCSOs than our two methods. In some difficult cases, such as tightness being 0.6 and density being 0.3, MCS can consume 7 times more NCCCSOs than UFRslv and RFRslv in a fast network. In slow network, however, the advantages of UFRslv and RFRslv over MCS are narrower.

With ABT-DOng, when compared with Rslv, UFRslv and RFRslv can save up to 55% and 61% of number of messages respectively. They both save up to 44% of NCCCSOs in fast network. In slow network, they can save 55% and 63% of NCCCSOs respectively. The improvements obtained by our two methods with ABT-DOng

are more stable than those obtained with AFCng. When compared with MCS, the situation is similar to the last discussion. While the number of messages consumed by MCS is less sometimes, our techniques can save up to 50% of NCCCSOs over MCS in fast network, but the advantage gap gets smaller or even reverse sometimes in slow network.

5.2 Distributed Langford's Problem

Recall the Langford's problem (prob024 in CSPLib [3]), which is parametrized by a pair (m, n) and aims at finding an $m \times n$ digit sequence consisting of digits 1 to n, each occurring m times, such that any two consecutive occurrences of digit i are separated by i other digits. One way to model Langford's problem is to use $m \times n$ variables to represent each digit in the sequence. The value in each domain represents the place in sequence. The domain of each variable is $\{1, \ldots, mn\}$. For example, in the $(2, 4)$-Langford's problem, x_1 and x_2 represent the positions of the first 1 and the second 1, and $x_2 - x_1 = 2$. x_3 and x_4 represent the positions of the first 2 and the second 2, and $x_4 - x_3 = 3$; similarly for x_5, x_6 and x_7, x_8.

We bring the Langford's problem into the distributed context. In the Distributed (m, n)-Langford's Problem, we assume each variable is controlled by one agent so that we have $m \times n$ agents to control the variables.

Figure 3(a) - 3(c) and Figure 3(d) - 3(f) show the results of AFCng and ABT-DOng respectively on various (m, n) instances of the distributed Langford's problem using the four nogood learning techniques we have discussed. From left to right on the x-axis, (m, n) pairs are $(2, 6)$, $(2, 7)$, $(3, 6)$, $(3, 7)$, $(3, 8)$, $(3, 9)$, $(3, 10)$ and $(4, 7)$. In our AFCng implementations, MCS performs the best but by only *very thin margin* in number of messages for small instances, but RFRslv prevails in the large ones by a *considerable margin* when MCS even timeouts on all these instances. Timeouts are represented as the highest values in the graphs. In fast network, UFRslv is the best among the four techniques across all instances in terms of NCCCSOs, while UFRslv and RFRslv top the list in slow network. Rslv is in general the worst in performance in all metrics, but is relatively more robust than MCS. We also note that the performances of UFRslv and RFRslv are actually very close to each other in the Langford's instances, so that their lines overlap in our graphs, which have a wide spread in scale in the vertical axis.

In our ABT-DOng implementations, again MCS performs well in small instances in number of messages, but RFRslv takes over in the largest two instances. In terms of NCCCSOs, both UFRslv and RFRslv, which have very close performance, outperform MCS and Rslv substantially. Rslv is the worst in all metrics.

Savings in number of messages and runtime (NCCCSOs) is dramatic. With AFCng, UFRslv and RFRslv can save around 68% of messages over Rslv in some difficult problems, such as $(3, 8)$ and $(3, 10)$-Langford's problem. In fast and slow communication, our methods can save NCCCSOs up to 78% and 70% respectively. MCS timeouts in the largest four instances. Among the non-timeout instances, UFRslv and RFRslv have almost identical number of messages as MCS, but can save up to 82% of NCCCSOs in fast network. The savings in slow network become almost minimal.

With ABT-DOng, UFRslv and RFRslv can save up to 87% of number of messages over Rslv. In fast and slow communication, our methods can save NCCCSOs up to 85% and 88% respectively comparing with Rslv. MCS are better only in the five smaller instances in number of messages. Among the three largest instances, UFRslv and RFRslv can save up to 82% of messages over MCS. In terms of NCCCSOs, UFRslv and RFRslv are always better and can improve up to 88% and 93% for fast and slow network respectively.

6. CONCLUDING REMARKS

In this paper, we present UFRslv and RFRslv nogood learning. These new techniques were incorporated in AFCng and ABT-DOng. In both proposals, each agent will do more internal reasoning to get shorter nogoods than Rslv nogood learning. Computing shorter nogoods costs internal computation in an agent. If we take communication costs into the evaluation metric, however, the stronger pruning we get from the shorter nogood can give us more benefits by pruning larger parts of the search space. Our methods outperform Rslv nogood learning in terms of number of messages and NCCCSOs. In most cases, our new techniques consume around similar or even less number of messages as MCS nogood learning does. However, our methods use much less NCCCSOs.

Between the two new methods, RFRslv nogood learning is slightly better than UFRslv nogood learning in general in terms of messages. In fast network, UFRslv nogood learning consumes less NCCCSOs. In slow network, however, RFRslv nogood learning becomes better in terms of NCCCSOs. We can choose different nogood learning techniques for specific environments.

A possible future direction of work is to continue the development of new nogood learning techniques. Doing that, we should note the possible trade-off between the effort we spend on generating nogoods and the effectiveness of the nogoods in pruning search space.

We observe that our nogood learning techniques works better with ABT-DOng over AFCng, i.e. the ratio of improvement is larger and more stable with ABT-DOng. Another possibility is to investigate efficient dynamic variable and value ordering algorithms to make good use of the properties of nogoods.

Acknowledgement

We are grateful to the anonymous referees for their constructive comments. The work was generously supported by GRF grants CUHK413808, CUHK413710, CUHK413713 and CSIC/RGC Joint Research Scheme grant S-HK003/12 from the Research Grants Council of Hong Kong SAR.

7. REFERENCES

[1] C. Bessière, A. Maestre, I. Brito, and P. Meseguer. Asynchronous backtracking without adding links: a new member in the ABT family. *Artificial Intelligence*, 161(1):7–24, 2005.

[2] R. Ezzahir, C. Bessière, M. Belaissaoui, and E. H. Bouyakhf. DisChoco: a platform for distributed constraint programming. In *Proc. Workshop on Distributed Constraint Reasoning*, pages 16–27, 2007.

[3] I. P. Gent and T. Walsh. CSPLib: a benchmark library for constraints. In *Proc. CP'99*, pages 480–481, 1999.

[4] K. Hirayama and M. Yokoo. The effect of nogood learning in distributed constraint satisfaction. In *Proc. ICDCS'00*, pages 169–177, 2000.

[5] K. Hirayama, M. Yokoo, and K. Sycara. The phase transition in distributed constraint satisfaction problems: first results. In *Proc. CP'00*, pages 515–519, 2000.

[6] D. Mammen and V. R. Lesser. Problem structure and subproblem sharing in multi-agent systems. In *Proc. ICMAS'98*, pages 174–181, 1998.

[7] A. Meisels, E. Kaplansky, I. Razgon, and R. Zivan. Comparing performance of distributed constraints processing algorithms. In *Proc. Workshop on Distributed Constraint Reasoning*, pages 86–93, 2002.

Figure 1: Experiment Result on Random DisCSPs (tightness = 0.6)

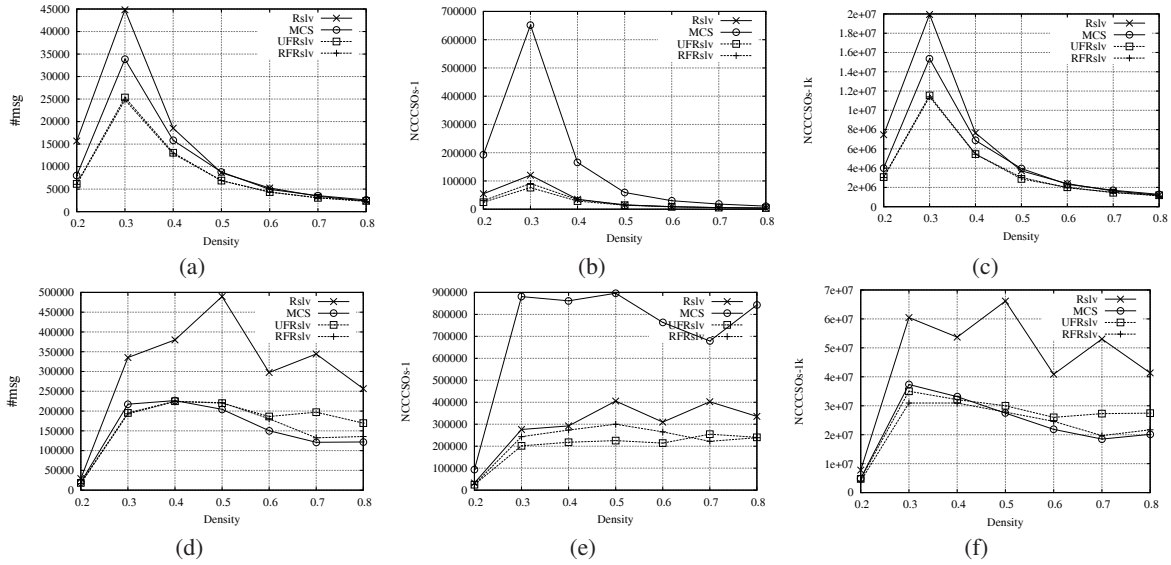
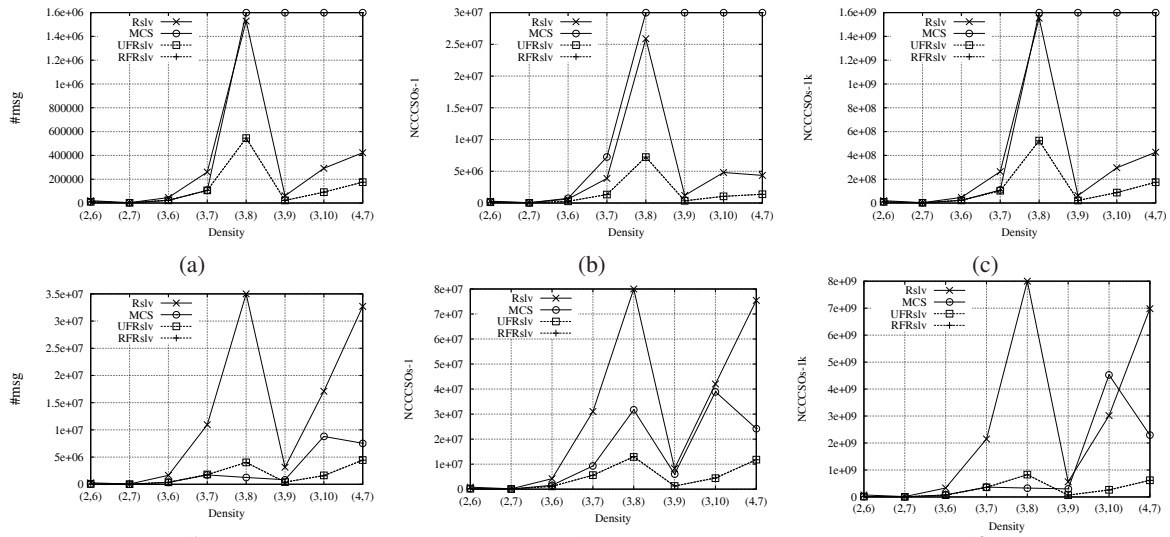


Figure 2: Experiment Result on Langford's Problem



[8] A. Meisels and R. Zivan. Asynchronous forward-checking for DisCSPs. *Constraints*, 12(1):131–150, 2007.

[9] K. Sycara, S. F. Roth, N. Sadeh, and M. S. Fox. Distributed constrained heuristic search. *IEEE Transactions on Systems, Man and Cybernetics*, 21(6):1446–1461, 1991.

[10] M. Wahbi. *Algorithms and ordering heuristics for distributed constraint satisfaction problems*. PhD thesis, Université Montpellier II-Sciences et Techniques du Languedoc, 2012.

[11] W. Yeoh, A. Felner, and S. Koenig. BnB-ADOPT: an asynchronous branch-and-bound DCOP algorithm. *Journal of AI Research*, 38:85–133, 2010.

[12] M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Proc. CP'95*, pages 88–102, 1995.

[13] R. Zivan, M. Zazone, and A. Meisels. Min-domain retroactive ordering for asynchronous backtracking. *Constraints*, 14(2):177–198, 2009.