Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence

Real-Time Solving of Quantified CSPs Based on Monte-Carlo Game Tree Search

Baba Satomi, Yongjoon Joe, Atsushi Iwasaki, and Makoto Yokoo
Department of ISEE, Kyushu University
Fukuoka, Japan
{s-baba@agent., yongjoon@agent., iwasaki@, yokoo@}inf.kyushu-u.ac.jp

Abstract

We develop a real-time algorithm based on a Monte-Carlo game tree search for solving a quantified constraint satisfaction problem (QCSP), which is a CSP where some variables are universally quantified. A universally quantified variable represents a choice of nature or an adversary. The goal of a QCSP is to make a robust plan against an adversary. However, obtaining a complete plan off-line is intractable when the size of the problem becomes large. Thus, we need to develop a real-time algorithm that sequentially selects a promising value at each deadline. Such a problem has been considered in the field of game tree search. In a standard game tree search algorithm, developing a good static evaluation function is crucial. However, developing a good static evaluation function for a QCSP is very difficult since it must estimate the possibility that a partially assigned QCSP is solvable. Thus, we apply a Monte-Carlo game tree search technique called UCT. However, the simple application of the UCT algorithm does not work since the player and the adversary are asymmetric, i.e., finding a game sequence where the player wins is very rare. We overcome this difficulty by introducing constraint propagation techniques. We experimentally compare the winning probability of our UCT-based algorithm and the state-of-the-art alpha-beta search algorithm. Our results show that our algorithm outperforms the state-of-the-art algorithm in large-scale problems.

1 Introduction

A constraint satisfaction problem (CSP) [Mackworth, 1992] is the problem of finding an assignment of values to variables that satisfies all constraints. Each variable takes a value from a discrete finite domain. A variety of AI problems can be formalized as CSPs. Consequently, CSP research has a long and distinguished history in AI literature. A quantified constraint satisfaction problem (QCSP) [Chen, 2004] is an extension of a CSP in which some variables are universally quantified. A universally quantified variable can be considered the choice of nature or an adversary. The goal of a QCSP is to make a robust plan against an adversary. A QCSP can formalize various application problems including planning under uncertainty and playing a game against an adversary.

While solving a CSP is generally NP-complete, solving a QCSP is generally PSPACE-complete. Thus, as the number of variables increases, obtaining a complete plan off-line becomes intractable quickly when the size of the problem becomes large. In an off-line planning, if a complete solution is not found before the agent actually plays against the adversary, it is a complete failure. However, if the adversary is not omniscient, the agent does not necessarily need a complete plan to defeat the adversary. The agent should make a reasonable choice, even if it is not guaranteed to be succeed, using the available time until each deadline. In this paper, we develop a real-time algorithm that sequentially selects a promising value for each variable at each deadline.

Most existing algorithms for solving a QCSP are off-line algorithms [Bacchus and Stergiou, 2007; Gent *et al.*, 2005]. One notable exception is [Stynes and Brown, 2009]. In [Stynes and Brown, 2009], a real-time algorithm for solving a QCSP is presented. This algorithm applies a standard game tree search technique to QCSP; it is a combination of a lookahead method based on an alpha-beta pruning and a static evaluation function. In [Stynes and Brown, 2009], several alternative strategies are evaluated. A strategy called Intelligent Alpha Beta (IAB) is shown to be most effective. In IAB, child nodes in a search tree are ordered from best to worst, and an alpha-beta search is executed. In this algorithm, the evaluation value is calculated based on a static evaluation function for a *leaf node* in a partially expanded game tree (which is not a terminal node of the fully expanded game tree).

In a standard game tree search algorithm including alpha-beta, developing a good static evaluation function is crucial. However, it is very unlikely that we can develop a good static evaluation function for a QCSP because it must estimate the possibility that a partially assigned QCSP is solvable. This task is difficult even for a standard CSP. The static evaluation function in [Stynes and Brown, 2009] (called Dynamic Geelen's Promise) uses the product of the sizes of future existential domains. This seems reasonable for a standard CSP, but it is not clear whether this function is really appropriate to a QCSP.

In this paper, we apply a Monte-Carlo game tree search technique that does not require a static evaluation function.

A Monte-Carlo method, which is an algorithm based on repeated random sampling, evaluates the node by the results of many playouts in which we play a game randomly until it is finished. Thus, the evaluation values are stochastic. In Computer game Go, a variation of the Monte-Carlo method called the UCT (UCB applied to Trees) algorithm [Kocsis and Szepesvári, 2006] has been very successful. One merit of UCT is that it can balance exploration and exploitation when selecting a node to start a playout. In this paper, we also use a UCT-based algorithm.

However, the player and the adversary are extremely asymmetric in a QCSP if we choose parameters, such as constraint tightness, similar to a standard CSP. A prevailing assumption in a CSP literature is that satisfying all constraints is difficult. For example, in the eight-queens problem (which is considered as a very easy CSP instance), if we place eight queens on the chess board at random, the chance that these queens do not threaten with each other is very small. Thus, if we simply apply UCT, finding a game sequence where the player wins is very rare. As a result, the UCT's decision is about the same as a random guess. To overcome this difficulty, we introduce constraint propagation techniques based on a concept called *arc-consistency* to allow the algorithm to concentrate on the part of the game tree where the player has some chance to win. We experimentally compare the winning probability of our UCT-based algorithm and the state-of-the-art alpha-beta search algorithm (IAB). Our results show that our algorithm outperforms IAB for large-scale problems.

The rest of this paper is organized as follows. In Section 2, we show the formalization of a QCSP, real-time online solving of a QCSP, and the UCT algorithm as a related research. In Section 3, we present real-time algorithms for solving a QCSP. Then, in Section 4, we show the experimental results. Finally, in Section 5, we conclude this paper.

2 Related Research

2.1 Quantified CSP

A constraint satisfaction problem (CSP) is a problem of finding an assignment of values to variables that satisfies constraints. A CSP is described with n variables $x_1, x_2, \cdots, x_n$ and m constraints $C_1, C_2, \cdots, C_m$. Each variable x_i takes a value from a discrete finite domain D_i.

A QCSP [Chen, 2004] is a generalization of a CSP in which variables are existentially ($\exists$) or universally ($\forall$) quantified. Solving a QCSP is PSPACE-complete. Each quantifier is defined by the sequence of quantified variables. Sequence Q consists of n pairs, where each pair consists of quantifier Q_i and variable x_i, as represented in (1):

$$Q_1 x_1 \cdots Q_n x_n. \tag{1}$$

Note that the sequence order matters, e.g., the meanings of $\forall x \exists y\ loves(x, y)$ and $\exists y \forall x\ loves(x, y)$ are quite different. $\forall x \exists y\ loves(x, y)$ means any x loves some y, where y can be different for each x. On the other hand, $\exists y \forall x\ loves\ (x, y)$ means particular person y is loved by everybody.

A QCSP has a form QC as in (2), where C is a conjunction of constraints and Q is a sequence of quantified variables:

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 (x_1 \neq x_3) \wedge (x_1 < x_4) \wedge (x_2 \neq x_3). \tag{2}$$

The semantics of a QCSP QC can be defined recursively as follows:

- If C is empty then the problem is true. If Q is of the form $\exists x_1 Q_2 x_2 \cdots Q_n x_n$, then QC is true iff there exists some value $a \in D_1$ such that $Q_2 x_2 \cdots Q_n x_n C[(x_1, a)]$ is true. If Q is of the form $\forall x_1 Q_2 x_2 \cdots Q_n x_n$, then QC is true iff for each value $a \in D_1$, $Q_2 x_2 \cdots Q_n x_n$ $C[(x_1, a)]$ is true. Here, $C[(x_1, a)]$ is a constraint C where x_1 is instantiated to value a.

2.2 Real-Time Online Solving of QCSP

In the real-time online solving of QCSP [Stynes and Brown, 2009], a QCSP is treated as a two-players game, in which the existential player assigns values to existentially quantified variables and the universal player assigns values to universally quantified variables. Each player must decide the value of each variable within a time limit. For the existential player, the goal is to reach a solution, but the universal player is trying to prevent a solution from being reached. Real-Time online solving of QCSP for the existential player is defined as follows:

- Given QCSP QC, increasing sequence of time points $t_1, t_2, \cdots, t_n$, and sequence of values $v_1, v_3, v_5, \cdots, v_{n-1}$ such that each value v_j is in D_j and is revealed at time t_j, generate at each time t_k for $k = 2, 4, 6, \cdots, n$ a value $v_k \in D_k$ such that the tuple $(v_1, v_2, \cdots, v_n)$ is a solution for QC.

Here, for simplicity, the sequence of quantifiers of QCSP QC is assumed to be a strictly alternating sequence of quantifiers, starting with $\forall$ and ending with $\exists$.

2.3 UCT

UCT (Upper Confidence bound applied to Trees) [Kocsis and Szepesvári, 2006] is a Monte-Carlo method combined with tree search. A Monte-Carlo method is a stochastic technique using random numbers, that evaluates nodes with the mean score given by the result of many playouts in which we play a game randomly until it is finished. It is effective when a lookahead search is difficult, e.g., the size of the game tree is huge, and/or designing a good static evaluation function is hard. For Computer Go, UCT-based algorithms are very effective. CrazyStone [Coulom, 2006] is one of the first computer Go programs that utilizes UCT. UCT is also successful in *General Game Playing* competitions, where an agent should accept declarative descriptions of an arbitrary game at runtime and be able to play the game effectively without human intervention [Finnsson and Björnsson, 2008].

UCT can be considered an application of Upper Confidence Bound (UCB) [Auer *et al.*, 2002] to tree search, which is a technique applied to the multi-armed bandit problem. The goal of the problem is to select a sequence of arms that maximizes the sum of rewards. The reward of each arm is given by a fixed probability distribution which is initially unknown. In this problem, achieving a good balance between exploration and exploitation, i.e., whether to use a trial for gathering new information, or collecting of rewards, is important. When UCB is applied for selecting an arm, it selects an arm whose UCB value is highest. A UCB value is defined in (3):

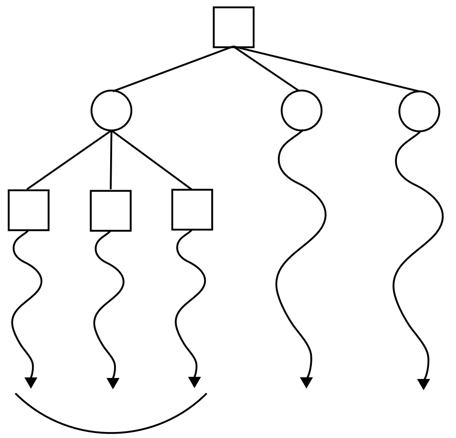

Figure 1: Node expansion by UCT

$$\bar{X}_j + \sqrt{\frac{2 \log t}{t_j}}. \tag{3}$$

$\bar{X}_j$ is the mean of rewards given by the plays of arm j, t_j is the number of plays of arm j, and t is the overall number of plays. The first term favors the arm with the best empirical reward while the second term prefers an arm with fewer plays.

In UCT, a partially expanded game tree (which is a subtree of a fully expanded game tree) is stored. Initially, this subtree has only one root node. UCT continue to select a node by the UCB value in this subtree, from the root node until it reaches one leaf node of the subtree (which is not a terminal node of the fully expanded game tree). A leaf node is expanded if the number of visits of the node reaches a threshold value. When UCT arrives at a leaf node, a random playout starts from the leaf node to a terminal node, where the result of the game is determined. The result of the playout is updated iteratively from the leaf node to the root node as an evaluation value.

Figure 1 represents the subtree used in UCT. Since UCT gives only two values that represents winning or losing as a reward, an evaluation value represents the winning probability. In Figure 1, the number of visits for the left node is largest since the winning probability of the left node is highest. So the left node is expanded first. On the other hand, the center and right nodes are also visited (although less frequently than the left node) due to the second term in (3).

Algorithm 1 illustrates the UCT algorithm. UCT searches a tree repeatedly until a time limit. One search trial includes three processes. The first process (i) in Algorithm 1 is a function that selects a node by the UCB value. The second process (ii) represents one playout, which returns the result of the playout. The last process (iii) is a function that updates the evaluation values of nodes. This function updates the evaluation values from a leaf node to the root node by the result of the playout.

Algorithm 1 UCT algorithm

```
while not timeout do
   search(rootNode);
end while

function search(rootNode)
node := rootNode;
while node is not leaf do
   node := selectByUCB(node); – (i)
end while
value := playout(node); – (ii)
updateValue(node, value); – (iii)
end function
```

3 Real-Time Algorithm based on Monte-Carlo for Solving QCSPs

In this section, we present a real-time algorithm for solving a QCSP, which is based on UCT. First, we show the basic real-time algorithm based on UCT (Section 3.1). However, since this basic algorithm fails to obtain useful information from playouts in a reasonable amount of time, we modified it by introducing constraint propagation techniques (Section 3.2).

3.1 Basic Monte-Carlo Algorithm for QCSPs

A UCT-based algorithm stores a subtree of a game tree. In a QCSP, since the values of the variables must be determined in a sequence of quantifiers, a QCSP's game tree can be represented as follows:

- A node in a game tree represents a partially assigned QCSP. More specifically, the root node represents a state where no variable's value has been assigned yet. A node whose depth is i represents a state where the values have been assigned from the first variable to i-th variable in the sequence.
- Each node has a variable that should be assigned next. The number of links from a node equals the domain size of the variable of the node, i.e., each link represents the choice of a value of the variable. A child node represents a state where the value of the parent node's variable is assigned according to the link.

Our algorithm continues to select a child node (which corresponds to selecting the value of a variable by the UCB value), from the root until it reaches one leaf node of the subtree. In our algorithm, we set the threshold value of a node expansion to one, i.e., a leaf node is expanded on the first visit. When a leaf node is expanded, its children are generated and they become new leaf nodes. Then, the algorithm selects one new leaf node randomly, and starts a playout from the selected node.

In this algorithm, for each playout, we assign values randomly to all variables based on the sequence of quantifiers. The existential player wins if all constraints are satisfied, otherwise the universal player, i.e., the adversary, wins. If the existential player wins, the evaluation values of the nodes are updated by 1, otherwise by 0. The update procedure from the universal player's point of view is symmetric. More specifically, the evaluation values of the nodes are updated by -1 if the existential player wins.

We introduce a *Pure Value Rule* [Gent *et al.*, 2005] for basic pruning that is defined as follows:

- It prunes the search space using a concept called *pure value*, which is defined as follows:
  - Value $a \in D_i$ of QCSP QC is *pure* iff $\forall Q_j x_j \in Q$, where $x_j \neq x_i$ and $\forall b \in D_j$, the assignments (x_i, a) and (x_j, b) are compatible.

  An existential variable with a *pure value* can be set to that value, while a *pure value* is removed from the domain of a universal variable.

In our algorithm, a *Pure Value Rule* is applied in a process selectByUCB() in Algorithm 1 when a node is expanded. The algorithm creates child nodes for values except for values removed by the *Pure Value Rule*, which reduces the number of child nodes since it is applied before a node is expanded. However, the *Pure Value Rule* is not powerful enough to reduce the search space. Since the player and the adversary are extremely asymmetric in a QCSP, the probability that a set of random value assignments satisfies all constraints is very small. Thus, finding a game sequence where the player wins is very rare, and evaluation values are updated only by 0 in most cases. As a result, the decision of UCT is about the same as a random guess. To overcome this difficulty, we introduce more powerful constraint propagation techniques so that the algorithm can concentrate on the part of the game tree where the player has some chance to win.

3.2 Improvement of Basic Algorithm with Constraint Propagation

We modified a basic real-time algorithm for solving a QCSP by introducing constraint propagation techniques, i.e., we remove values from the domains, that cannot be a part of the final solution. This corresponds to pruning nodes where the universal player wins in the future. By pruning such nodes, the algorithm can concentrate on the part of the game tree where the player has some chance to win. We present two different approaches on how to update evaluation values of nodes based on the result of constraint propagation: one is called *shallow* update, the other is called *deep* update.

Algorithm 2 Process of node expansion

```
if node is a leaf node then
  pureValueRule(node);
  for each value v in domain of variable that the node has do
    child := new node(v);
  end for
  for each child do
    constraintPropagate(child);
  end for
end if
```

Incorporating More Powerful Constraint Propagation Techniques

We introduce a constraint propagation technique based on a concept called *arc-consistency*. A CSP is called *arc-consistent*, if for any pair of variables x_1 and x_2, for any value a in D_1, there exists at least one value b in D_2 that satisfies the constraint between x_1 and x_2. An arc-consistency algorithm removes the values if they do not satisfy the above condition. Our algorithm achieves Strongly Quantified Generalized Arc Consistency (SQGAC) [Nightingale, 2007] by constraint propagation. For QCSP, constraint C is SQGAC iff for each variable x included in C, for each value $a \in D_x$, for all universal variables $x_i, x_j, \cdots$,, which appear after x in sequence Q, and for each partial assignment $\{(x, a), (x_i, b) \mid b \in D_i, (x_j, c) \mid c \in D_j, \cdots\}$, there exists a strategy to assign other variables in C so that constraint c is satisfied. When all constraints are binary, SQGAC is reduced to Quantified Arc Consistency (QAC) [Bordeaux and Monfroy, 2002].

In our algorithm, when a node is expanded, or a value of a variable is determined within a random playout, constraint propagation is executed to achieve SQGAC. We present two alternative methods (deep/shallow update) that are different in how far the update of evaluation values continues.

Details of Algorithm

The modified algorithms apply constraint propagation for all child nodes created as represented in Algorithm 2. The algorithm performs SQGAC. If the domain of an existentially quantified variable becomes empty or any value is removed from the domain of a universally quantified variable by achieving SQGAC, the assignment eventually leads to a constraint violation. In the shallow update method, when the algorithm finds such a constraint violation, it updates the evaluation value of the node that leads to a constraint violation to $-\infty$.

In our deep update method, as well as the above procedure for the shallow update, we incorporate the following additional procedure to update the evaluation values of ancestor nodes.

- Assume for node i, which represents an existentially quantified variable, the evaluation values of all i's child nodes become $-\infty$ (as a result of several shallow/deep updates). Then node i will eventually lead to a constraint violation. Thus, the algorithm updates i's evaluation value to $-\infty$.
- Assume for node i, which represents a universally quantified variable, the evaluation value of one of i's child nodes is $-\infty$ (as a result of another shallow/deep update). Then node i will eventually lead to a constraint violation (assuming the adversary takes the child node whose evaluation value is $-\infty$). Thus, the algorithm updates i's evaluation value to $-\infty$.

Note that value $-\infty$ is used as a sign that the node causes a constraint violation. When updating the evaluation value of its parent node, it is treated as 0.

While constraint propagation is effective for pruning the search space and for concentrating playouts on the part of the game tree where the player has some chance to win, we need

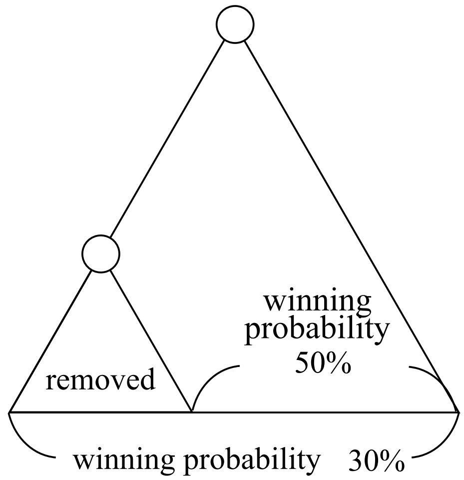

Figure 2: Change of winning probability with constraint propagation

to redefine the way of calculating the winning probability of a node based on the result of the playouts. More specifically, when a value is removed by constraint propagation, the estimated winning probability obtained by playouts is skewed, as illustrated in Figure 2.

Here, assume 100 terminal nodes exist from the leaf node in total. Each terminal node represents a state where the values of all variables are assigned. Within 100 terminal nodes, since 40 are descendants of the removed node, they are not considered in the playouts. Assume the estimated winning probability by random playouts for this leaf node is 50%. This means that within the 60 unremoved nodes, the player can win around 30 terminal nodes. On the other hand, the correct winning probability should be $\frac{30}{40+60} = 30\%$.

To overcome this problem, we need to incorporate the information of the removed nodes, (which result in constraint violations, i.e., player's loss) into the evaluation values and UCB values. We redefine a UCB value as follows:

$$\bar{X}_j \times \left(1 - \frac{l}{L}\right) + \sqrt{\frac{2\log t}{t_j}} \quad (4)$$

l is the number of terminal nodes pruned, and L is the total number of terminal nodes. Thus, l/L is the rate of the pruned terminal nodes (thus unexplored). Therefore, $\bar{X}_j \times (1 - l/L)$ is the adjusted winning probability including the pruned terminal nodes. This probability should be close to the real winning probability. When the universal player wins in all playouts from a node, the node's UCB value is determined only by the second term since the first term is 0.

4 Experiments

In this section, we experimentally compare the winning probability of our UCT-based algorithm and the state-of-the-art alpha-beta search algorithm, when they play against a deliberative and random adversary. We can consider a random adversary represents a choice of nature or an irrational agent. Ideally, a real-time algorithm should perform well against both rational/irrational adversaries.

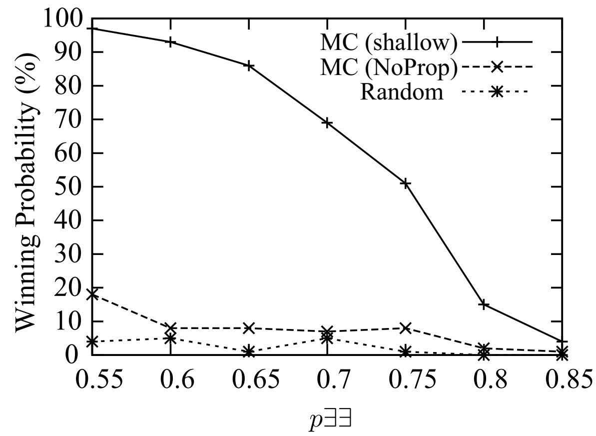

Figure 3: Effect of constraint propagation: QCSP against random adversary ($n = 20, d = 8$)

We created problem instances with a strictly alternating sequence of $\exists$ and $\forall$ quantifiers as [Stynes and Brown, 2009]. A random binary QCSP instance is generated based on five parameters; $\langle$ n, d, p, $p_{\exists\exists}$, $p_{\forall\exists}$ $\rangle$, where n is the number of variables, d represents the domain size, which is the same for all variables, and p represents the number of binary constraints as a fraction of all possible pairs of variables. $p_{\exists\exists}$ represents the number of constraints in the form of $\exists x_i \exists x_j, c_{ij}$ as a fraction of all possible tuples. $p_{\forall\exists}$ is a similar quantity for $\forall x_i \exists x_j, c_{ij}$ constraints, described below. The other two types of constraints are not generated since they can be removed by preprocessing.

When many constraints exist in the form of $\forall x_i \exists x_j, c_{ij}$, most problem instances are insolvable. To generate enough solvable instances, constraints in the form of $\forall x_i \exists x_j, c_{ij}$ are restricted in the following way, as described in [Gent *et al.*, 2005]. We generate a random total bijection from one domain to the other. All tuples that are not in this bijection are excluded in the constraint. From this total bijection, choose $p_{\forall\exists}$ fraction of tuples as constraints.

In Figures 3–5, we chose the following parameters: $n = 20$, $d = 8$, $p = 0.20$, $p_{\forall\exists} = 0.5$. Then we varied $p_{\exists\exists}$ from 0.55 to 0.85. For each value of $p_{\exists\exists}$, we generated 100 instances. The time limit in which each player determines a value is 1000ms, i.e., if a player uses a UCT-based algorithm, it tries to perform as many playouts as possible until the time limit. Also, if a player uses a standard game tree search, it tries to lookahead the search tree as deep as possible until the time limit. All experiments were run on an Intel Xeon 2.53GHz processor with 24GB RAM. For this parameter setting, we can check whether a problem instance has a winning strategy or not by using an off-line algorithm. When $p_{\exists\exists} = 0.55$, almost all problem instances have winning strategies. When $p_{\exists\exists} = 0.60$, approximately 95% of problem instances have winning strategies. Also, when $p_{\exists\exists} = 0.65, 0.70$, and 0.75, the ratios of problem instances with winning strategies are about 80%, 60%, and 20%, respectively.

Figures 3 and 4 show the ratio of problem instances that the existential player wins. Figure 3 illustrates the effect of

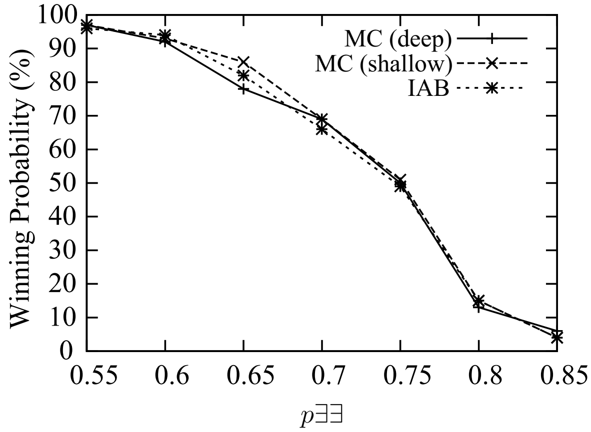

Figure 4: QCSP against random adversary ($n = 20, d = 8$)

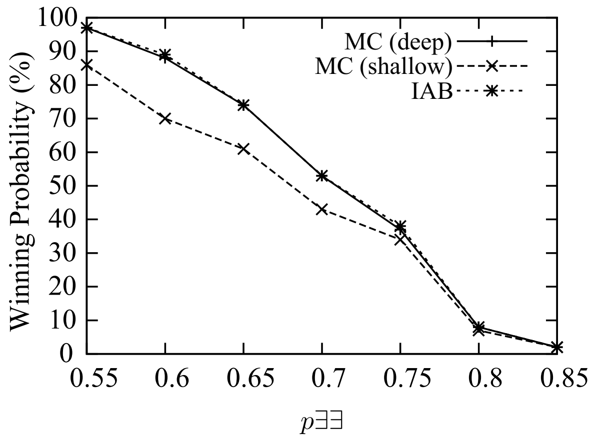

Figure 5: QCSP against alpha-beta ($n = 20, d = 8$)

constraint propagation. Our UCT-based algorithm without constraint propagation (MC (NoProp)) performed very badly; it is slightly better than a random player (Random). On the other hand, the performance improves significantly by incorporating constraint propagation with a shallow update method (MC (shallow)). Constraint propagation is clearly very effective in our real-time algorithm for solving QCSPs.

Figure 4 compares our UCT-based algorithms and the state-of-the-art lookahead algorithm. MC (deep) is the result of our UCT-based algorithm with a deep update method, and IAB is the result of the lookahead algorithm with an Intelligent Alpha Beta (IAB) strategy [Stynes and Brown, 2009]. The evaluation results reported in [Stynes and Brown, 2009] indicate that IAB with a static evaluation function called Dynamic Geelen's Promise (DGP), which uses the product of the sizes of future existential domains, performs best. Thus, we also used DGP for the static evaluation function of IAB. In Figure 4, the results of each algorithm are almost equivalent. Actually, these differences are not significant[1].

Figure 5 shows the experimental results where the universal player applies the lookahead algorithm with Alpha Beta. As shown in Figure 5, the performance of MC (shallow) is worse than IAB and MC (deep), and these differences are significant. On the other hand, MC (deep) and IAB are almost equivalent, and this difference is not significant.

In Figures 6 and 7, we show the result for larger problem instances of the following parameters: $n = 50$, $d = 16$, $p = 0.20$, $p_{\forall\exists} = 0.5$. We varied $p_{\exists\exists}$ from 0.25 to 0.50 when the universal player applies a random adversary. Then we varied $p_{\exists\exists}$ from 0.25 to 0.45 when the universal player applies a lookahead algorithm with Alpha Beta. The time limit in which each player determines a value is 3000ms. For each value of $p_{\exists\exists}$, we generated 100 instances.

Figure 6 shows the experimental result against a random adversary. In this experiment, MC (shallow) performs much better than IAB and MC (deep), and these differences are significant. IAB performs better than MC (deep) (in particular, when $p_{\exists\exists}$ is 0.4 and 0.45), but the difference is not significant for overall settings of $p_{\exists\exists}$.

Figure 7 shows the experimental result against Alpha Beta. In this experiment, our UCT-based algorithms performs much better than IAB, and these differences are significant. MC (shallow) and MC (deep) are almost equivalent, and this difference is not significant. When the search tree becomes too large, a lookahead based algorithm cannot obtain enough information to make a reasonable decision. On the other hand, our UCT-based algorithm still manages to obtain some useful information within a limited amount of time.

The results for MC (deep) and (shallow) are somewhat puzzling. Initially, we expected that MC (deep) will consistently outperform MC (shallow). Our expectation was as follows. Assume MC (deep) removes a move/value of the existential player. Then, MC (shallow) also will not select the move after enough random playouts, since in the UCT, a node that has more chance to win is selected more frequently. Thus, we assumed that the removal never hurts. However, in larger problem instances, MC (shallow) outperforms MC (deep) when the adversary is random, and they are almost equivalent when the adversary is deliberative.

We do not have a definite answer for explaining this fact yet. Our current conjecture is as follows. In a larger problem instance (with a longer time limit), the subtree stored by our Monte-Carlo algorithm becomes large. Then, performing a deep update requires certain overhead. As a result, MC (shallow) can run more random playouts than MC (deep). MC (deep) tries to avoid a value assignment that can lead to its loss by applying an additional procedure to update the evaluation values, assuming the adversary is rational. This additional effort does not pay very well, especially when the adversary is a random player. More detailed analysis is needed to clarify the merit/demerit of deep/shallow update procedures.

5 Conclusions

In this paper, we presented a real-time algorithm for solving a QCSP. We applied a Monte-Carlo game tree search technique called UCT, which has been very successful in games like Go, and does not require a static evaluation function. We found that a simple application of UCT does not work

[1]In this section, we apply paired t-tests and the significant level is 5%.

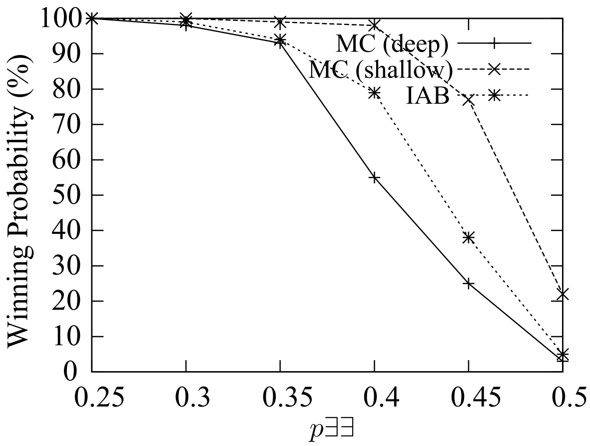

Figure 6: Large-scale QCSP against random adversary ($n = 50, d = 16$)

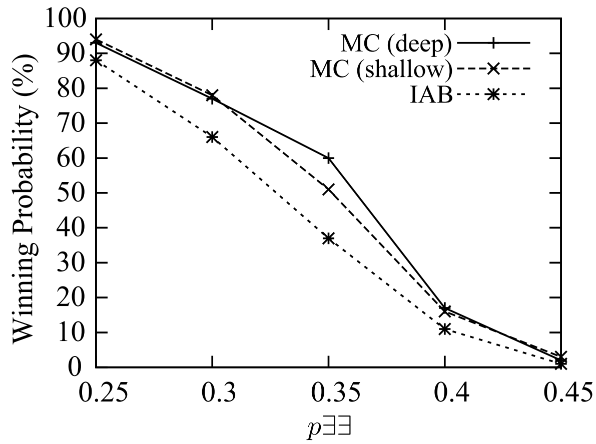

Figure 7: Large-scale QCSP against alpha-beta ($n = 50, d = 16$)

for a QCSP because the player and the adversary are extremely asymmetric and finding a game sequence where the player wins is very rare. As a result, the UCT's decision is about the same as a random guess. Thus, we introduced constraint propagation techniques so that the algorithm can concentrate on the part of the game tree where the player has some chance to win, and obtain a better estimate of the winning probability. Experimental results showed that our UCT-based algorithm with constraint propagation greatly outperforms the algorithm with no constraint propagation. Furthermore, experimental results show that our algorithm is better than the state-of-the-art alpha-beta search algorithm for large-scale problems. Our future works include developing more efficient algorithms for solving a QCSP by improving the formula of the UCB value calculation and introducing an endgame database. Also, we hope to perform experiments with non-random QCSP instances, such as bin-packing games presented in [Stynes and Brown, 2009].

Furthermore, our ultimate research goal is to develop an real-time algorithm for a quantified distributed CSP (QDCSP) [Baba *et al.*, 2010; Yokoo *et al.*, 1998]. A QDCSP is a QCSP in which variables are distributed among agents. In a QDCSP, as in a QCSP, obtaining a complete plan off-line is intractable. Thus, a team of cooperative agents need to make their decisions in real-time. We hope to extend the algorithm developed in this paper to QDCSP.

Acknowledgments

This research was partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (A), 20240015, 2008. The authors would like to thank anonymous reviewers for their helpful comments.

References

[Auer *et al.*, 2002] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

[Baba *et al.*, 2010] S. Baba, A. Iwasaki, M. Yokoo, M. C. Silaghi, K. Hirayama, and T. Matsui. Cooperative problem solving against adversary: quantified distributed constraint satisfaction problem. In *AAMAS*, pages 781–788, 2010.

[Bacchus and Stergiou, 2007] F. Bacchus and K. Stergiou. Solution directed backjumping for QCSP. In *CP*, pages 148–163, 2007.

[Bordeaux and Monfroy, 2002] L. Bordeaux and E. Monfroy. Beyond NP: Arc-consistency for quantified constraints. In *CP*, pages 371–386, 2002.

[Chen, 2004] H. M. Chen. *The computational complexity of quantified constraint satisfaction*. PhD thesis, Cornell University, 2004.

[Coulom, 2006] R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *CG*, pages 72–83, 2006.

[Finnsson and Björnsson, 2008] H. Finnsson and Y. Björnsson. Simulation-based approach to general game playing. In *AAAI*, pages 259–264, 2008.

[Gent *et al.*, 2005] I. P. Gent, P. Nightingale, and K. Stergiou. QCSP-Solve: A solver for quantified constraint satisfaction problems. In *IJCAI*, pages 138–143, 2005.

[Kocsis and Szepesvári, 2006] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *ECML*, pages 282–293, 2006.

[Mackworth, 1992] A. K. Mackworth. Constraint satisfaction. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 285–293. John Wiley & Sons, 1992.

[Nightingale, 2007] P. Nightingale. *Consistency and the Quantified Constraint Satisfaction Problem*. PhD thesis, University of St Andrews, 2007.

[Stynes and Brown, 2009] D. Stynes and K. N. Brown. Realtime online solving of quantified CSPs. In *CP*, pages 771–786, 2009.

[Yokoo *et al.*, 1998] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, 1998.