DUCT: An Upper Confidence Bound Approach to Distributed Constraint Optimisation Problems *

Brammert Ottens, Christos Dimitrakakis, Boi Faltings

DUCT: An Upper Confidence Bound Approach to Distributed Constraint Optimisation Problems*

Boi Faltings

December 12, 2018

Abstract

We propose a distributed upper confidence bound approach, DUCT, for solving distributed constraint optimisation problems. We compare four variants of this approach with a baseline random sampling algorithm, as well as other complete and incomplete algorithms for DCOPs. Under general assumptions, we theoretically show that the solution found by DUCT after T steps is approximately T^{-1}-close to the optimal. Experimentally, we show that DUCT matches the optimal solution found by the well-known DPOP and O-DPOP algorithms on moderate-size problems, while always requiring less agent communication. For larger problems, where DPOP fails, we show that DUCT produces significantly better solutions than local, incomplete algorithms. Overall we believe that DUCT is a practical, scalable algorithm for complex DCOPs.

1 Introduction

Many modern optimisation problems involve multiple agents trying to solve a common optimisation goal. Examples include scheduling meetings between multiple participants with different constraints and preferences (Petcu, 2006), planning delivery times in a heterogeneous logistics network (Léauté and Faltings, 2011a) or minimising energy use in smart devices (Rust et al., 2016). Another natural problem is channel allocation in wireless networks, for which we introduce a benchmark in Section 5.1. The common goal is to minimise the total cost of participants, while ensuring that none of their constraints are violated.

While we can redefine such problems in a centralised manner, it is better to treat them as distributed. One reason is that this allows parallelisation of computations, through the development of distributed algorithms. The second reason is that the structure of the problem itself also allows joint decision making

*The canonical version of this paper appears in ACM Transactions on Intelligent Systems and TechnologyOttens et al. (2017).

by heterogeneous agents, without knowing their preferences and constraints a priori.

In this paper, we introduce the Distributed Upper Confidence Tree (DUCT) algorithm for solving such distributed constraint optimisation problems (DCOP). DUCT works by first defining a tree structure for the communication between agents, which is created using knowledge of the shared constraints between agents. The agents communicate by passing messages on this structure. Heuristic upper confidence bounds are used to guide the search for good solutions by focusing on the most promising solution subsets. This is unlike dynamic programming algorithms which must find optimal solutions for every subtree. Experimentally, we show that DUCT performs as well as dynamic programming methods for small problems, with significantly less communication overhead, and that it is able to find good solutions to problems which are too large for dynamic programming to solve. DUCT can be seen as a dynamic programming algorithm where the solution space is stochastically enlarged. Thus, while we eventually recover the exact solution, we can stop earlier with a nearly optimal solution.

Theoretically, we show that under mild assumptions, it is possible to find a nearly-optimal solution quickly. The assumption is that the number of near-optimal solutions is not insignificant. This allows DUCT to find an ϵ-optimal solution in order $1/\epsilon$ steps.

The remainder of the paper is organised as follows. Section 1.1 discusses related work and our contribution. Section 2 formally introduces the problem of distributed constraint optimisation. Section 3 describes DUCT, as well as a simpler, random algorithm. Section 4 analyses DUCT theoretically, while Section 5 compares the performance of DUCT to a number of local search and global optimisation algorithms. We conclude with a discussion in Section 6. Technical proofs are collected in Appendix A.1.

1.1 Related work and our contribution

Distributed Constraint Optimisation (DCOP) involves constrained optimisation problems with multiple variables, where not all of the variables are controlled by a single agent. Thus, the paradigm can be used to model coordination problems that are inherently distributed. The basic problem was introduced by Yokoo et al. (1998) in its satisfaction form, i.e. where the problem is simply to find a *feasible* solution. Many practical problems correspond to DCOPs, such as meeting scheduling (Maheswaran et al., 2004), inference in distributed sensor networks (Ali et al., 2005) and transportation problems (Léauté et al., 2010; Ottens and Faltings, 2008b).

DCOPs can be solved efficiently with a number of methods. Some of the first are the self-explanatory Synchronous Branch and Bound (Hirayama and Yokoo, 2003), as well as a significantly more efficient branch and bound algorithm, called Asynchronous Forward Bounding (AFB) (Gershman et al., 2006). Another asynchronous method is ADOPT (Modi et al., 2005), which uses exact bounds in combination with a complex backtracking mechanism. Approaches

based on dynamic programming principles include DPOP (Petcu and Faltings, 2005), and its variants such as O-DPOP (Petcu and Faltings, 2006) and the asynchronous A-DPOP (Ottens and Faltings, 2008a) as well as extensions employing branch and bound (Yeoh et al., 2010) to reduce complexity. Such methods systematically search the solution space, exploring subsets until they are certain that no good solution can be found there.

Another approach to solving DCOPs is to use local methods. One of the first such approaches was the distributed stochastic algorithm (DSA) (Fitzpatrick and Meertens, 2003). Later on, Maheswaran et al. (2004) reformulated DCOPs as a multi-player game and introduced the Maximum Gain Message (MGM) algorithm, a minor variant of DBA (Zhang et al., 2003), as well as two 2-coordinated algorithms, MGM-2 and SCA-2 (Stochastic Coordination Algorithm-2), where agents are allowed to co-ordinate actions with their neighbours. Typically, such methods have bounded space requirements. However, similarly to Markov Chain Monte Carlo (MCMC) methods in statistics, it is hard to achieve robust performance in a problem-independent way.

The method we present is inspired by the confidence bound methods in the multi-armed bandits literature. A natural starting point is the UCB algorithm (Auer et al., 2002) and its variants for tree-structured problems, such as UCT (Kocsis and Szepesvári, 2006) and HOO (Bubeck et al., 2011). Such algorithms are very successful in large search problems such as playing Go (Gelly and Silver, 2005; Silver et al., 2016). However, they generally assume stochasticity and smoothness.

While this is not the case in our setting, we propose a distributed UCT algorithm (DUCT), which is applicable to DCOPs. It combines *heuristic bounds* with random search to efficiently trade off exploring unknown parts of the search space with searching promising, but relatively well-explored subsets. Even though the bounds may initially not be admissible, their slow relaxation ensures that they *eventually* become admissible and so the optimal solution can be found. The question is how to relax them so that our solutions improve as quickly as possible.

In practice, the algorithm not only can outperform local-search in terms of solution quality, but is also able to provide good feasible solutions for problems where optimal methods are too complex. In addition, we show that DUCT does this with a very low computation and communication cost.

We provide a theoretical analysis of DUCT (as well as of a completely stochastic algorithm, RANDOM) based on weak assumptions on the problem structure, that characterise the difficulty of the problem by the number of solutions that are ϵ-close to the optimum in sets that contain the optimal solution. This is necessary: otherwise the optimal solution can be hidden within many bad solutions and no algorithm performs well.

To the best of our knowledge, no previous work uses sampling and confidence bounds for solving DCOPs. However, sampling has been used for solving both Constraint Satisfaction Problems (CSP) (Gogate and Dechter, 2003) and Quantified CSPs (Satomi et al., 2011). Furthermore, in (Léauté and Faltings, 2011b) sampling is used as a preprocessing method when dealing with stochastic

problems. In a different vein (Stranders et al., 2012) introduced MAB-DCOPs, where stochasticity in a utility function is modelled as a bandit problem. Their algorithm uses a UCB inspired bound which is asymptotically optimal. However, finding the assignment globally maximising an upper bound is harder than solving a deterministic DCOP. Thus, our point of departure is that we apply sampling to improve the complexity of solving DCOPs, rather than handling randomness in the problem. Thus our results refer to the (non-asymptotic) computational effort of solving a DCOP. Finally, Nguyen et al. (2013) followed up on our original results (Ottens et al., 2012) with an approach based on MCMC optimisation and promising preliminary results for graph colouring problems. Like any MCMC method, however, its non-asymptotic convergence properties are unclear.

2 Distributed Constraint Optimisation

A Distributed Constraint Optimisation Problem (DCOP) consists of a set of variables, owned by different agents, and a set of constraints, each defined over a set of variables. The objective is to find a variable assignment that gives a feasible solution that minimises cost. More precisely:

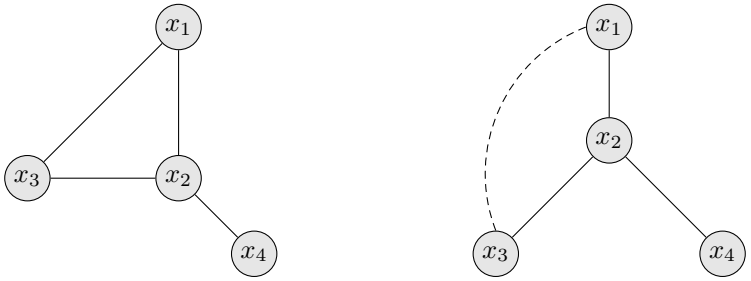
Definition 1 (A DCOP problem)**.** A discrete distributed constraint optimisation problem is a tuple $\langle \mathcal{A}, \mathcal{X}, \mathcal{M}, \mathscr{D}, \mathcal{F} \rangle$ where

- $\mathcal{A} \triangleq \{1, \ldots, K\}$ is a set of agents;
- $\mathcal{X} \triangleq \{x_1, \ldots, x_N\}$ is a set of variables. Each variable is owned by a single agent;
- $\mathscr{D} \triangleq \{D_1, \ldots, D_N\}$, is a collection of finite domains, with product space $\mathcal{D} = \prod_{i=1}^{N} D_i$, such that $x_i \in D_i$;
- $\mathcal{F} \triangleq \{f_1, \ldots, f_M\}$ is a set of *constraints* where each constraint $f_i : \mathcal{D}_i \to \mathbb{R} \cup \{\infty\}$ depends on $N(i)$ variables, with $\mathcal{D}_i \triangleq D_{i_1} \times \ldots \times D_{i_{N(i)}}$. If $\mathbf{x} = \{x_1, \ldots, x_n\}$, then $[\mathbf{x}]_{f_i}$ is used to denote the projection of $\mathbf{x}$ to the subspace on which the i-th constraint is defined, and $var(f_i)$ to denote the variables in the range of f_i.

A *feasible assignment* $\mathbf{x} \in \mathcal{D}$ is an assignment to all variables such that

$$f(\mathbf{x}) \triangleq \sum_{i=1}^{M} f_i\left([\mathbf{x}]_{f_i}\right) < \infty, \tag{1}$$

while we let $\mathcal{D}_\Phi \triangleq \{\mathbf{x} \,|\, f(\mathbf{x}) < \infty\}$ be the feasible set. The objective is to find a feasible assignment that minimises the sum of the constraints, with the minimum value being $f^*(\mathcal{D}) \triangleq \min_{\mathbf{x} \in \mathcal{D}} f(\mathbf{x})$.



(a) Constraint graph (b) pseudo-tree

Figure 1: The constraint graph for $f_1(x_1, x_2, x_3)+f_2(x_2, x_4)$, showing the dependencies between the variables. On the right side, we can see one of its possible pseudo-trees, rooted at x_1. All lines in the pseudo-tree represent dependencies, but communication is generally restricted to the solid lines comprising the tree, while *back edges* (the dashed line) do not admit communication.

Multiple variables per agent For simplicity, we assume that each agent owns only a single variable. However, our method can easily be extended to the more general case. This can be done by treating each variable as though it is owned by a different (virtual) agent. However, in that case the virtual agents would all share one computational unit.

2.1 Agent communication

Agents only have to communicate with agents that they share a constraint with. This communication structure can be nicely captured using a constraint graph (Dechter, 2006). This is a graph such that any two variables that share a constraint are linked with an edge. The equivalent constraint graph for a problem with four variables, $\mathbf{x} = (x_1, x_2, x_3, x_4)$ and two constraints, $\mathcal{F} \triangleq \{f_1, f_2\}$ is given in Figure 1(a). More formally:

Definition 2 (Constraint Graph)**.** Given a DCOP $\langle \mathcal{A}, \mathcal{X}, \mathcal{M}, \mathscr{D}, \mathcal{F} \rangle$, its constraint graph $\mathcal{G} = \langle \mathcal{X}, \mathcal{E} \rangle$ is such that $(x_i, x_j) \in \mathcal{E}$ iff there is a $f_k \in \mathcal{F}$ such that $x_i, x_j \in var(f_k)$.

Since DCOP is a distributed problem, we need to define how the agents will communicate with each other. The optimal topology for the communication network might be algorithm-dependent. A simple choice, used by some algorithms (SynchBB (Hirayama and Yokoo, 2003), AFB (Gershman et al., 2006)), is a linear topology. This ignores the constraints, disallows parallel exploration of the search space and can force unconnected agents to communicate. Instead, it is better to create communication links among agents that share constraints. One such topology is a pseudo tree, which allows computation to be parallelised in different branches. More specifically, it is a spanning tree of the constraint graph:

Definition 3 (Pseudo-tree)**.** A pseudo-tree $\mathcal{T}(\mathcal{G}, x_i) = \langle \mathcal{V}, \mathcal{E}', \mathcal{B} \rangle$ of a graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, is a set of x_i-rooted directed spanning trees (one for each connected sub graph), where each tree has to form $(\mathcal{V}, \mathcal{E}')$ of $\mathcal{G}$ with $\mathcal{B} = \mathcal{E} \setminus \mathcal{E}'$ the set of back edges, where $(v, v') \in \mathcal{B}$ means that v is an ancestor of v'.

The pseudo-tree derived from the constraint graph shown in Figure 1(a) is shown in Figure 1(b). The dashed line, called a *back edge*, denotes the part of the graph that was removed when the spanning tree was created.

Each node x_i in the pseudo-tree has a *separator* s_i, consisting of all the ancestors of x_i that must be removed from the problem to separate the subtree rooted at x_i from the problem. We define the width of a pseudo-tree as the size of the maximal separator. Algorithms like ADOPT (Modi et al., 2005) perform a top-down search on the pseudo-tree, while inference algorithms like DPOP (Petcu and Faltings, 2005) perform bottom-up inference on the same structure. In these algorithms, as well as ours, information never flows on back edges during the search for the optimal solution.

To obtain the tree, we find a spanning tree of the constraint graph by doing a distributed Depth First Search (DFS) traversal of this graph. Here, we make the assumption that each agent knows which constraint it shares with which other agents. The spanning tree is constructed by first choosing a root, and then start the DFS traversal from this root, where to choose the root we use the maximum degree heuristic. We use the algorithm as detailed in Hamadi (1998). [1]

We must now define what information to send along the communication links defined by the pseudo-tree. In our algorithms, we use the links for two purposes. Downstream communication is used to convey `CONTEXT` messages, describing relevant variable assignments to lower nodes. The same information is conveyed via `FINISHED` messages, which also indicate when a node thinks a near-optimal solution has been found. Upstream communication is used to relay `COST` messages, which describe the local and cumulative costs, or the infeasibility of the given assignment and bounds on the cost, depending on the algorithm. The main algorithmic choice is how to perform the variable assignment and is discussed in the next section.

3 Distributed UCT

To visualise the sampling strategies used by our algorithms, we represent the search space with an AND/OR graph (Dechter and Mateescu, 2004). To prevent confusion, the nodes in the AND/OR graph are called *contexts*, distinguishing them from the nodes in the pseudo-tree, which represent *agents*. The pseudo-tree defines a structure over which the agents communicate, while the AND/OR graph defines the structure of the search space itself. The current state of the search is stored in the nodes of this graph.

[1] Unfortunately, finding the minimal width pseudo tree, which would be optimal for algorithms such as DPOP, is an NP-hard problem.

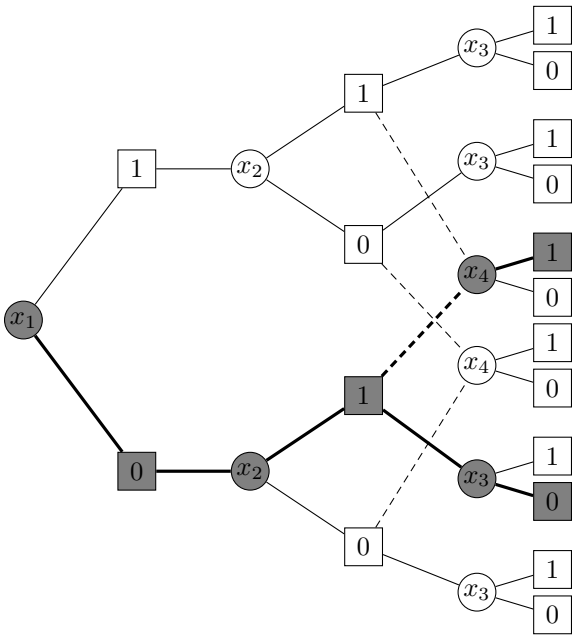


Figure 2: The AND/OR graph for the DCOP shown in Figure 1, with $\forall_i D_i = \{0, 1\}$. Different variable selections, in OR nodes, lead to different contexts as we move down the tree. Whenever variables are independent, the graph merges. For example, x_4 has only two possible contexts, since it does not depend on x_1.

The graph consists of alternating AND and OR contexts. OR contexts represent alternative choices for a particular variable. AND contexts represent a problem decomposition. Figure 2 shows the AND/OR graph for the constraint graph of Figure 1(b). The squares and circles represent AND and OR nodes respectively. Since the subproblem rooted at x_4 does not depend on the value of x_1, we merge the two subgraphs corresponding to $x_1 = 0$ and $x_1 = 1$ into a single subgraph. A *path* represents an assignment to all variables. At AND contexts, the path *branches* to all the children, while at an OR context, the path *chooses* only a single child. The bold lines in Figure 2 constitute a path, and represent the assignment $\{x_1 = 0\,, x_2 = 1\,, x_3 = 0\,, x_4 = 1\}$.

Our algorithms use the structure shown in Algorithm 1 to explore the space. At time t, each node k receives a context a containing values for relevant variables from its parent. It then selects a value d for itself, and transmits a new context $a \cup x_k = d$ to its children. Contexts are sent using a `CONTEXT` message containing all variables relevant for a child. Each child k' returns the total cost $y^t_{k'}$ incurred in its subtree using a `COST` message. The node k then sums up the costs of the children, as well as the cost $\ell^k(a, d)$ of the variables it enforces[2] and

[2]For each constraint, the agent that enforces this constraint is chosen as the agent in the constraint's scope that is the lowest in the pseudo-tree. The fact that the pseudo-tree does

returns y_k^t.

When an agent has received a `COST` message (l. 26) from all its children, we distinguish the following cases. If its parent has terminated and a local termination condition is met, it sends a termination message to its children (l. 31), otherwise it resamples and sends a new `CONTEXT` message to its children. If the total cost received from its children is infeasible ($y_k^t = \infty$) and there are still feasible solutions to its own subproblem,[3] it resamples and sends a new `CONTEXT` message to its children. Otherwise, it returns an (algorithm-dependent) `COST` message to its parent.

The main differences between our algorithms lie in the `sample()` subroutine, which chooses candidate values, and the content of the `COST` messages. The first algorithm, RANDOM, which randomly selects values, is described in Section 3.1. Section 3.2 describes DUCT, which combines stochasticity with upper confidence bounds to more intelligently guide the search. Issues common to both algorithms, namely normalisation of the cost function and dealing with hard constraints are explained in Sections 3.3 and 3.4 respectively.

3.1 Random Sampling: The Random algorithm

As a warm-up, we first describe a very simple distributed algorithm, RANDOM, which samples the solution space randomly without replacement. The sampling is done locally, by randomly choosing values for each context message received. This is implemented using the `sample()` routine (Algorithm 2) within the top-level Algorithm 1.

More precisely, the root agent randomly selects a value for its variable, and sends a `CONTEXT` message containing this variable assignment to all its children. Every time an agent k receives a `CONTEXT` message, containing a context a, it randomly chooses a value $d \in D_k$ for its variable with replacement, appends this to a, and sends this enlarged context to its children using a `CONTEXT` message. This process stops when the leaf agents are reached. At this time, the algorithm has selected a path through the AND/OR tree. For the received context a, the leaf agents calculate the minimal value the sum of the constraints they enforce can take:

$$y_k^t = \min_{d \in D_k} \ell^k(a, d)\,, \tag{2}$$

where t denotes that this is the t-th sample taken by agent k, and $\ell^k(a, d)$ is the sum of the values of the constraints enforced by agent k. y_k^t is sent up the path using the message $\texttt{COST}(k, t) \triangleq \langle y_k^t \rangle$. The k-th agent calculates its cost using:

$$y_k^t = \ell^k(a, d) + \sum_{k' \in \mathcal{C}_k} y_{k'}^t\,. \tag{3}$$

not have any back-edges across branches (i.e. back-edges are always between ancestors and descendants) guarantees the unicity of this enforcing agent.

[3]Whenever an agent receives an infeasible cost message, it marks its local problem as infeasible for the current value it chose for its variable

ALGORITHM 1: General algorithmic structure for agent k

Function `sampleAndSend`(a, k) **is**
  x_k^t = `sample`(a, k);
  for *each child* k' **do**
    `send`(k', `CONTEXT`($(a \cup \{x_k = x_k^t\})$))

initialization
  if *root* **then**
    parentFinished = true;
    `sampleAndSend`(a_k^t, k)
  else
    parentFinished = false;

when `received(CONTEXT(`a`))` from parent
  $a_k^t = a \cap s_k$;
  if *variable* k *is a leaf node* **then**
    $\ell_{\min} = \min_{d \in D_k} \ell^k(a_k^t, d)$;
    `send`(*parent*, `COST`(k, t))
  else
    `sampleAndSend`(a_k^t, k)

when `received(FINISHED(`a`))` from parent
  $a_k^t = a \cap s_k$;
  parentFinished = true;
  if *Eq.* (6) *satisfied* **then**
    for *each child* k' **do**
      `send`(k', `FINISHED`($a_k^t \cup \{x_k = \hat{d}_a\}$))
  else
    `sampleAndSend`(a_k^t, k)

when `received(COST(`k', t`))` from child k'
  if *received cost message from all children* **then**
    if $y_k^t = \infty$ **then**
      $\ell^k(a_k^t, x_k^t) = \infty$;
    if *parentFinished* **then**
      if *Eq.* (6) *satisfied* **then**
        for *each child* k' **do**
          `send`(k', `FINISHED`($a_k^t \cup \{x_k = \hat{d}_a\}$))
      else
        `sampleAndSend`(a_k^t, k)
    else
      if $y_k^t = \infty$ *and* $\exists d\, \ell^k(a_k^t, d) < \infty$ **then**
        `sampleAndSend`(a_k^t, k)
      else
        `send`(*parent*, `COST`(k, t))

ALGORITHM 2: `SAMPLE`(a, k): random sampling

```
d = random value from D_k;
while ℓ^k(a, d) = ∞ and there are untried values do
    d = random value from D_k;
return d
```

Here $\mathcal{C}_k$ denotes the children of agent k. The cost is thus the sum of the values reported by its children plus the value of its local problem.

3.1.1 Termination

As a DCOP is distributed, a local termination condition is necessary. We wish to terminate sampling only when the difference between the expected optimal solution and the currently found best is small enough. This requires that all agents are certain that they are sufficiently close to the optimal solution.

To define the termination condition, let a_k^t be the context received by agent k at time t, and x_k^t be the value chosen by agent k at time t. Agent k stores the following variables for all contexts a that it receives:

$\hat{\mu}_{a,d}^t$ — The lowest cost found for value d under context a:

$$\hat{\mu}_{a,d}^t \triangleq \min\left\{y_k^l \,\middle|\, l \leq t : a_k^l = a, x_k^l = d\right\}. \tag{4}$$

$\hat{\mu}_a^t$ — The lowest cost found under context a: $\hat{\mu}_a^t \triangleq \min_d \hat{\mu}_{a,d}^t$.

$\hat{d}_a$ — The value with the lowest cost: $\hat{d}_a \triangleq \arg\min_d \hat{\mu}_{a,d}^t$.

$\tau_{a,d}^t$ — The number of times d has been selected for x_k under context a:

$$\tau_{a,d}^t \triangleq \sum_{l=1}^{t} \mathbb{I}\left\{a_k^l = a \wedge x_k^l = d\right\}. \tag{5}$$

While strictly speaking these variables should also be indexed by k, we do not use it as a subscript in the above notation, as it is clear from context.

An agent k **terminates** when the following conditions are met:

1. Its parent has terminated (this condition trivially holds for the root agent);
2. The following inequality holds for the last context a reported by its parent:

$$\varepsilon_a \triangleq \max_{d \in D_k} \hat{\mu}_a^t - (\hat{\mu}_{a,d}^t - \sqrt{\frac{\ln \frac{2}{\delta}}{\tau_{a,d}^t}}) \leq \epsilon, \tag{6}$$

ALGORITHM 3: `SAMPLE`(a, k): DUCT sampling

```
1 if all values in D_k have been sampled at least once then
2 |   return arg min_{d ∈ S_a^t} B_{a,d}^t
3 else
4 |   d = random unsampled value from D_k;
5 |   while ℓ^k(a, d) = ∞ do
6 |   |   d = random unsampled value from D_k;
7 |   return d
```

where ϵ and δ are parameters of the algorithm.[4] If all d are infeasible for a, we effectively take the inequality to be true.

In Algorithm 1, `FINISHED` messages are used to signal termination. As seen in line 21, when an agent k terminates, it adds $x_k = \hat{d}_a$ to the context a, and sends a `FINISHED` message to its children. Since all our domains are finite, there will be at least one $\tau_{a,d}^t$ that always increases, and so the algorithm is guaranteed to satisfy this condition in a finite number of steps. After an agent has terminated, its children continue sampling until their termination condition is met. By setting their value to $\hat{d}_a$ upon termination, the agents reconstruct the best global assignment seen so far.

3.2 Confidence Bounds: The DUCT algorithm

An agent-context pair (k, a) corresponds to an OR node in the AND/OR graph. Rather than choosing the next value d randomly, an agent could focus the search on more promising choices. Our DUCT family of algorithms achieves this as follows. At time t, an agent k receiving a context a chooses d minimising $B_{a,d}^t$, a heuristic lower bound on the cost, with ties broken randomly. We explore four variants of this algorithm, that all differ in the definition of the lower bound.

The local confidence bound The first step towards building a confidence bound for the complete DCOP, is to define a measure of local uncertainty for every OR node in the graph. Intuitively, the more times we have tried each option d in an OR node, the more certain we should be about how close we are to the best possible value for that choice. In addition, the uncertainty should also depend on the number of remaining choices to be made down the tree. In our paper, we use the following UCB-style bound (Auer et al., 2002):

$$L_{a,d}^t = \sqrt{\frac{2\lambda_a \ln \tau_a^t}{\tau_{a,d}^t}}, \qquad \tau_a^t \triangleq \sum_{l=1}^{t} \mathbb{I}\left\{a_k^l = a\right\}, \tag{7}$$

[4] Intuitively, ϵ controls how close we wish to be to the optimal solution, while δ can be roughly seen as the probability that the algorithm will give up without having found an ϵ-optimal solution. A more precise analysis of the effects of these parameters will be given in Sec. 4

where τ_a^t is the number of times context a has been received and λ_a is the length of the path to the deepest leaf node.

The *path length* can be easily obtained during initialisation, and in particular during the normalisation phase (Sec. 3.3). Putting $\lambda_a = 1$ transforms Eq. (7) into the standard UCB bound. However, this treats a node close to the root the same way as deeper nodes, while the remaining search space is much bigger for the former. A similar observation has been made for other optimisation settings in (Coquelin and Munos, 2007; Kleinberg, 2005). By setting λ_a to the length of the maximal path from a to a leaf node, we ensure that the higher in the tree a node is, the more samples it must collect to reach a confidence level. Finally, when a value d is not sampled for a while, the bound slowly increases. As a result, no region of the search space will be ignored and a good balance between exploitation of currently known good branches and exploration of unknown parts of the search space is achieved.

A naive bound A simple way to use (7) for choosing values is to simply subtract it from the best value seen so far. This is also the way such a bound was used in UCT (Kocsis and Szepesvári, 2006). The bound combines the local constraints with the best value seen so far and a confidence interval:

$$B_{a,d}^t \triangleq \hat{\mu}_{a,d}^t - L_{a,d}^t. \tag{8}$$

For leaf agents, we simply set $B_{a,d}^t = \ell^k(a,d)$.

A recursive bound Due to the deterministic and finite nature of the DCOP problems, at some point all nodes of a particular subtree will have been sampled. Then further sampling will not yield any new information. The bound as described in Eq. (8) is not able to recognise this situation. For the parent of a leaf node, by definition this situation already occurs after sampling it once, i.e. after sampling it for the first time its confidence bound is reduced to 0. The following recursive bound, which is similar to the one employed in HOO (Bubeck et al., 2011), makes use of this information to allow agents to recognise when a part of the tree does not need to be sampled anymore:

$$B_{a,d}^t \triangleq \max \left\{ \hat{\mu}_{a,d}^t - L_{a,d}^t,\ \ell^k(a,d) + \sum_{k' \in \mathcal{C}_k} B_{k'}^t \right\}, \tag{9}$$

where $B_{k'}^t = \min_{d' \in D_{k'}} B_{a',d'}^t$ is the bound reported by agent k' for context $a' = a \cup \{x_k = d\}$. For leaf agents, $B_{a,d}^t = \ell^k(a,d)$.[5] Intuitively, bound (9) is an optimistic estimate of the optimal value for the sub-tree rooted at agent k for context a and choice d.

No matter which bound is used, the DUCT algorithm sends the cost message $\texttt{COST}(k,t) \triangleq \langle y_k^t, B_k^t \rangle$ for each node k.

[5]Note that the bounds need only be calculated when we are going back up the tree after a new sample has been taken.

The sampling procedure When $B^t_{a,d} = \ell^k(a,d) + \hat{\mu}^t_{a,d}$, the optimal cost for the subtree rooted at agent k, for context a and choice d, has been found. After this occurs, d should be ignored under context a. More precisely, when in context a at time t, only the following values of d will be sampled:

$$S^t_a = \{d \in D_k \mid B^t_{a,d} \neq \ell^k(a,d) + \hat{\mu}^t_{a,d}\}. \tag{10}$$

If not yet all choices in D_k have been tried for context a, then a random choice is made among the remaining values. If all values have been tried at least once, then the agent samples according to:

$$x^t_k \triangleq \arg\min_{d \in S^t_a} B^t_{a,d}, \tag{11}$$

with ties broken randomly.

The termination procedure This is identical to the one in RANDOM, with the exception of the local termination condition, which is now limited to the set of allowed choices:

$$\varepsilon_{a,d} \triangleq \max_{d \in S^t_a} \hat{\mu}^t_a - (\hat{\mu}^t_{a,d} - \sqrt{\frac{\ln \frac{2}{\delta}}{\tau^t_{a,d}}}) \leq \epsilon. \tag{12}$$

The DUCT algorithm has the high level structure described in Algorithm 1. The main difference with RANDOM is that it uses the sampling procedure `sample()` detailed in Algorithm 3.

3.3 Normalisation

In our paper, we generally assume that the range of the *global* cost is $[0,1]$. In general DCOP problems this is not the case. Hence a normalisation procedure must be carried out before we run our algorithms. For simplicity, assume that the $\min_{x \in \mathcal{D}_\Phi} f_i([\mathbf{x}]_{f_i}) = 0$ for all constraints f_i. If it is not, then it is trivial to make it so by subtracting the minimal feasible value of the sum of all constraints.

An upper bound on the global cost can now be found by summing up all the maximal values of all the individual local cost functions. More precisely, we simply determine the following upper bound on the cost of feasible solutions:

$$f^+ \triangleq \sum_{i=1}^{M} \max_{x \in \mathcal{D}_\Phi} f_i([\mathbf{x}]_{f_i}) \geq \max_{x \in \mathcal{D}_\Phi} \sum_{i=1}^{M} f_i([\mathbf{x}]_{f_i}). \tag{13}$$

To calculate the upper bound in a distributed manner, we can start from the leaf nodes of the tree and propagate the sum of the upper bounds to the root. As soon as the root of the pseudo-tree has received this upper bound, f^+, we then use a top-down procedure to disseminate the upper bound to all the agents.

3.4 Hard Constraints

In many problems, there is some non-empty infeasible set $\mathcal{D}_I = \mathcal{D} \setminus \mathcal{D}_\Phi$. This results in hard constraints, as $f(\mathbf{x}) = \infty$ for any $\mathbf{x} \in \mathcal{D}_I$. Dealing with such constraints is an important sub-problem.

Our approach is to ignore all infeasible parts of $\mathcal{D}$. An infeasible part of the search space has been found when, given a context, the sub problem rooted at an agent is infeasible. For leaf nodes, this means that their local problem is infeasible. For inner nodes, the subproblem is infeasible if for every possible value either (a) its local problem is infeasible or (b) at least one of its children reported an infeasible utility.

This procedure is detailed in Lines 26-40 of Algorithm 1. When a child reports an infeasible value, the value of the local problem corresponding to the current context a and the sampled value is set to ∞. After that, a new value is sampled and reported to the children. This means that when an agent receives an infeasible value from one of its children, it will continue searching for a feasible solution or stop when infeasibility of that context has been proven.

4 Theoretical Analysis

This section analyses the computational complexity of DUCT. We introduce the notion of simple regret (c.f. Cesa-Bianchi and Lugosi (2006)) to define performance, as well as our main assumptions, and provide a lower bound on regret. This is complemented by upper bounds on the regret for the RANDOM algorithm, followed by an upper bound for the DUCT algorithm. Finally, we discuss the space and communication complexity. Longer technical proofs are deferred to the appendix.

In our complexity analysis, we are interested in measuring the performance of the algorithm in terms of the (simple) regret. In particular, for any sequence of samples $y^t_{\mathcal{D}}$ from the domain, we define the following three notions of regret.

Definition 4. Let r_t be the *instantaneous regret*

$$r_t \triangleq y^t_{\mathcal{D}} - f^*(\mathcal{D}), \tag{14}$$

and ρ_T be the *(simple) regret* after T steps:

$$\rho_T \triangleq \min \{r_t \,|\, t = 1, \ldots, T\}. \tag{15}$$

Finally, we define the *total regret*:

$$R_T \triangleq \sum_{t=1}^{T} r_t. \tag{16}$$

We use this particular definition of simple regret because, after the algorithm has terminated, the value of the best choice found so far is known with perfect accuracy. The main problem in our analysis is how fast the regret decreases as

T increases. Since our algorithms are stochastic, apart from worst-case bounds, we shall also consider bounds on the regret both in expectation and with high probability.

It is important to note that a general problem-independent analysis for DCOP is not possible. This is because one can always construct a trivial counterexample with only one optimal solution $\mathbf{x}^*$, with all remaining choices of $\mathbf{x}$ being either infeasible or far from optimal. For this reason, we make the following assumption on the number of sub-optimal solutions.

Assumption 1. *Let λ be the counting measure on $\mathcal{D}$ and denote optimal value of $A \subset \mathcal{D}$ by $f^*(A) \triangleq \min\{f(\boldsymbol{x}) \,|\, \boldsymbol{x} \in A\}$. There exists $\beta > 0$ and $\gamma \in [0,1]$ s.t. $\forall A \subset \mathcal{D}$, $\epsilon \geq \gamma^{1/\beta}$:*

$$\lambda\left(\{\boldsymbol{x} \in A \,|\, f(\boldsymbol{x}) > f^*(A) + \epsilon\}\right) \leq \lambda(A)\,\gamma\epsilon^{-\beta}. \tag{17}$$

As this assumption guarantees that there are not too many solutions that are very far from optimal, it effectively guards against pathological cases where almost all choices are extremely bad. Here β can be seen as controlling the rate at which the number of sub-optimal solutions increases, and γ controlling the quality of the proportion of the worst possible solutions. The assumption is quite weak, as it does not guarantee how many choices will be arbitrarily close to the optimum. In particular, upper-bounding the right side by $\lambda(\mathcal{D})$ gives $\epsilon \geq \gamma^{1/\beta}$. A better intuition may be obtained by the following simple example.

Example 1. Consider n binary variables such that $\mathcal{D} = \{0,1\}^n$ and a cost function f with M constraints mapping to $[0,1]$ and no hard constraints. Assume that each variable affects at most m^* constraints. Define the Hamming metric $\nu(\mathbf{x},\mathbf{y}) = \sum_{i=1}^{n} \mathbb{I}\{x_i \neq y_i\}$. Then f is m^*-Lipschitz with respect to ν, as:

$$|f(\mathbf{x}) - f(\mathbf{y})| \leq m^*\nu(\mathbf{x},\mathbf{y}).$$

Let $\mathbf{x}$ be an optimal value, for any $\mathbf{y}$ such that $m^*\nu(\mathbf{x},\mathbf{y}) \leq \epsilon$, $f(\mathbf{x}) - f(\mathbf{y}) \leq \epsilon$. If $\mathcal{D}^*_\epsilon$ is the set of ϵ-optimal values, then $\mathcal{D}^*_\epsilon \supset \{\mathbf{y} \,|\, m^*\nu(\mathbf{x},\mathbf{y}) \leq \epsilon\}$ and so $\lambda(\mathcal{D}^*_\epsilon) \geq 2^{\epsilon/m^*}$ when λ is the counting measure.

Consequently, we can see that in this natural example, the number of values that are ϵ-optimal increases exponentially, rather than polynomially, with ϵ. This indicates that the assumption is quite weak.

Nevertheless, we can use this assumption to obtain bounds on the value $f^*(A)$ of any set A (see Lemma 1). In order to be able to prove something stronger, we shall need one additional assumption. In particular, the following assumption is trivially satisfied with $\lambda^* \geq 1$ when λ is the counting measure.

Assumption 2. *The set of optimal solutions has non-zero λ-measure, i.e.*

$$\lambda^* \triangleq \lambda(\mathcal{D}^*) > 0, \tag{18}$$

where $\mathcal{D}^* = \arg\min_{\boldsymbol{x}} f(\boldsymbol{x})$ is the set of optimal values. while $\bar{\lambda} \triangleq \lambda(\mathcal{D}) < \infty$.

Thus, an algorithm that samples all of $\mathcal{D}$ in fixed arbitrary order has regret bounded by Eq. (34) with $A = \mathcal{D}$, since an adversary could manipulate the problem so that the worst values are seen first. To begin the analysis, we prove a lower bound on the expected regret of any algorithm.

Theorem 1. *A lower bound on the expected regret is* $\mathbb{E}\,\rho_T \in \Omega(\gamma^{1/2\beta+T/2})$.

In order to prove this theorem, we consider a class of randomised problems. This reduces all algorithms to stochastic search. The complete proof can be found in the appendix. The next step is to consider upper bounds on the regret of stochastic search, which are given in the next section.

4.1 Analysis of Random

Here we show that RANDOM, which selects assignments uniformly, has a regret which partially depends on the size of $\mathcal{D}$. In addition, we show that it is a complete algorithm for a sufficiently small ϵ, δ in the termination condition.

Theorem 2. *Under Assumptions 1 and 2, the following bounds hold for the regret of the* RANDOM *algorithm. Firstly, the regret is* $\rho_T = 0$ *with probability at least* $1-(1-\lambda^*/\bar{\lambda})^T$ *and more generally, with probability* $1-\delta$*, the regret is:*

$$\rho_T \leq \left(\gamma\delta^{-1/T}(1-\lambda^*/\bar{\lambda})\right)^{1/\beta}, \qquad \forall\delta > \gamma^T, \tag{19}$$

Secondly, the expected regret obeys:

$$\mathbb{E}\,\rho_T \in O\left\{\left(1/\beta T + \gamma^{1/(\beta+1)}\right)\left(1-\lambda^*/\bar{\lambda}\right)^T\right\}. \tag{20}$$

When the size of $\mathcal{D}$ is very large, then the bound is dominated by the γ term, which can be much larger than $1/T$. Asymptotically, the expected regret is exponential in T, but with a base that is very close to 1. Consequently, the RANDOM algorithm may not be able to make very fast progress. Nevertheless, with appropriately selected parameters, it will eventually find the optimal solution.

Theorem 3. *For any* ϵ, RANDOM *is complete when it is run with* $\delta \leq \exp(-(1+\epsilon)^2\bar{\lambda})/2$.

To prove this theorem, we use the fact that it is necessary for the root node to stop before the algorithm stops.

4.2 Analysis of DUCT

The main idea behind DUCT is that although we have no knowledge of β, γ, we can slowly increase our bounds until they hold. Then, the algorithm focuses on nearly optimal branches. This allows us to prove the following bound on the simple regret.

Theorem 4. *For a DCOP with a binary root variable that partitions the solution space into two sets with $A_1 \cup A_2 = \mathcal{D}$, and assuming without loss of generality that A_2 contains the optimal solution, the worst-case regret is:*

$$\rho_T \leq \min \left\{ \frac{\kappa_2}{T} \left(\frac{16 \ln T}{\Delta} + \Delta \kappa_1 \right), \left(\frac{\gamma \lambda (A_2)}{T - 16\Delta^2 \ln T - \kappa_1} \right)^{1/\beta} \right\}, \tag{21}$$

where $\Delta \triangleq f^*(A_2) - f^*(A_1)$ and κ_1, κ_2 are constants depending on β, γ and the size of A_1, A_2.

The proof, given in Appendix A.2, uses a classical upper confidence bound construction. In particular, we first bound the number of times a sub-optimal branch is sampled by $O(\Delta^{-2} \ln T)$. This bounds the total regret accumulated while sampling this branch by $O(\Delta^{-1} \ln T)$. Together with a result on the relation of the simple regret to the total regret, we can bound the simple regret for sampling a sub-optimal branch by $O(T^{-1})$. Finally, we re-use Lemma 1 to bound the regret in optimal branches and thus bound the worst-case simple regret of DUCT.

Problems with no binary variables can be converted so that the theorem applies, by separating one of the variable into two or more variables, one of which is binary, and introducing suitable constraints so that the same number of possible values remains. With additional assumptions on the problem structure, one might be able to obtain slightly better (but more complex) bounds. Since it is not clear what general assumptions may be made, we leave this matter for future work. For example, by using the assumptions of HOO Bubeck et al. (2011) we could get a similar bound, but smoothness assumptions are not natural for DCOP problems.

As the theorem shows, the worst-case simple regret for a DCOP with a *binary root variable* is either approximately $T^{-1/\beta}$ or $T^{-1}\Delta^{-1}$. Which term is smallest depends largely on the size of A_1, A_2 and on Δ. Although the regret bound is of similar order to the one proved for RANDOM for small T, one must keep in mind that this was a probabilistic bound rather than a worst-case bound.

Finally, it would pay to comment on the regret bounds proved in (Stranders et al., 2012). Although of similar form for the total regret $O(\ln T)$, these are qualitatively different, since they are referring to the number of utility samples obtained from a stochastic utility function. This allows them to use standard bandit upper confidence bound techniques, through a martingale inequality, which gives a (total) regret bound in terms of *sample complexity*. In contrast, our paper discusses the number of rounds that the algorithm runs for, which gives a (simple) regret bound in terms of *computational complexity*. The computational complexity of DCOP in the stochastic setting remains an open question.

4.3 Memory complexity

Due to the distributed setting, we are interested in the maximal memory use over nodes, rather than the total memory use. For each agent, this depends on

the number of values that can be assumed by the variables in its separator, as well as the number of choices it can take and its number of children. Let m be the tree-width of the constraint graph, and assume that a variable can take at most v values (in the binary example, $v = 2$). Then the memory complexity is $O(v^m)$ in the worst case, after algorithm termination. However, the main question is how the memory complexity increases during algorithm execution.

First of all, since `CONTEXT` messages are sent down the main edges of the pseudo-tree, every variable appears only once. Secondly, every message contains a single assignment of variables in the separator, and there is only a single choice made at each OR node. Thus, the additional memory per step per variable is $O(m)$. Consequently, the memory complexity of both DUCT and RANDOM is $O(\min\{Tm, v^m\})$ after T tree traversals.

4.4 Communication complexity

The overall complexity of the algorithm strongly depends on the amount of parallelisation possible, and consequently the width of the pseudo-tree. First, we examine the local communication complexity. Since the algorithm is distributed, the global communication complexity equals the total flow of communication throughout the network. This can be analysed in terms of the local communication complexity in a worst case sense.

The local communication complexity Let Ψ the maximal branching factor of the pseudo-tree, v the maximum number of values a variable can take and w the width of the tree. Consider the k-th agent. It needs to communicate the values of w variables, including its own, to at most Ψ other nodes. Consequently, it needs to communicate $O(\Psi w \ln v)$ bits.

The global communication complexity Let $\Xi \leq K$ be the maximum depth of the pseudo-tree. Since the complexity will depend upon the topology of the pseudo-tree, let us analyse two specific cases. *Case 1: exponential-branching trees.* In that case there are $O(\Psi^\xi)$ agents at depth ξ of the tree. Consequently, the total flow at depth ξ is $O(\Psi^{\xi+1} w \ln v)$. Summing over all depths and applying the geometric series, we obtain after some simplifications $O(K\Psi^{\Xi+2} w \ln v)$. If the branching factor is always attained, the maximum depth must be logarithmic with respect to the number of agents, and so this becomes $O(K^2 \Psi w \ln v)$. *Case 2: bounded-branching trees.* In many practical cases trees have a sub-exponential tree width w. Then there are at most w agents at any depth, and the depth is bounded by K. Through the arithmetic series, we have a communication complexity of order $O(K^2 w \ln w^v)$

5 Experimental Evaluation

We compare our algorithms in a number of benchmark problems with a distributed structure. The more strongly connected the problem, the less dis-

algorithm	λ_a	sampling
DUCT-A	$\lambda_a = 1$	Eq. (8)
DUCT-B	$\lambda_a =$ path length	Eq. (8)
DUCT-C	$\lambda_a = 1$	Eq. (9)
DUCT-D	$\lambda_a =$ path length	Eq. (9)

Table 1: The DUCT variants.

tributed algorithms can take advantage of the structure.

In particular, we evaluate all algorithms on the *meeting scheduling* domain introduced in (Maheswaran et al., 2003), on randomly generated *graph colouring* problems and on the channel allocation problem introduced below. We evaluate 4 different versions of the DUCT algorithm, which are detailed in Table 1. We compare the four DUCT variants against a number of well known algorithms. We also include results for RANDOM, which was used as a baseline sampling algorithm in order to show that randomness is not the main reason why DUCT works. Comparisons are made both on solution quality, amount of information transmitted and runtime, where runtime is measured using the *simulated time* notion as introduced in (Sultanik et al., 2007).

We measured performance relative to the complete algorithms DPOP, O-DPOP, ADOPT, AFB and SynchBB, as well as the local and incomplete algorithms DSA, MGM and MGM2. Instead of including results with all these algorithms, we only present graphs for the best complete algorithm for clarity and conciseness. However, it is worthwhile to mention that for the *meeting scheduling* and *channel allocation* problems ADOPT, SynchBB and AFB could not solve the majority of the problems within the given time, while the performance of DPOP was consistently better than the performance of O-DPOP. In the *graph colouring* problem, DPOP dominated all other complete algorithms. Hence, to increase the readability of the graphs, we only included DPOP in the graphs. We nevertheless include all three incomplete algorithms in the results since their behaviours vary significantly.

5.1 The Channel Allocation Problem

WiFi access points are ubiquitous, but many use the same channel. This leads to interference, and thus less bandwidth and reliability. If each could use a different channel, the throughput would be much higher. Traditionally, such a channel allocation problem has been modelled as a graph colouring problem (Balasundaram and Butenko, 2006). Access points that can interfere one another are considered neighbours in the graph, with different channels being different colours. This oversimplifies the problem: for example, interference also depends on the transmission power and distance. Since most allocation problems are over-constrained, any solution must allow a certain amount of interference. In such cases, modelling the strength of the interference is important to obtain a good solution.

Modelling The capacity of a channel is given by the Shannon-Hartley theorem, that provides a relationship between the power of a signal, and the signal noise. The relationship has the following form $C = W \log_2(1 + \frac{S}{N})$, where C is the capacity of a channel, W is the bandwidth of the channel, S the signal strength and N the total noise or interference power. A channel causes interference to another channel when its signal has more power than background noise. The current WiFi standard allows for 13 channels. Two channels are overlapping, i.e. interfering, when the distance between them is not more than three channels.

Let $x_i \in D_i = \{1, \ldots, n\}$ denote the channel allocated to access point i, with n the number of available channels, Δ the maximal channel distance for which interference can occur, P_i the signal strength of i at the source, and $d_{i,j}$ the distance between access points i and j. Let $\mathrm{Ap}_i = \left\{ j \,\middle|\, P_j > N_b d_{i,j}^2 \right\}$ be the set of variable indices representing access points that can cause interference to the signal of i, with N_b the background noise. The function $I(x_i, x_j)$ returns 1 when x_j chooses an overlapping channel, and 0 otherwise, i.e. $I(x_i, x_j) = \begin{cases} 1 & \text{if } |x_i - x_j| \leq 3, \\ 0 & \text{otherwise.} \end{cases}$ Then, given a variable assignment $\mathbf{x} \in \prod_{j \in \mathrm{Ap}_i} D_j$, the capacity of access point i is modelled as

$$f_i(x_i, \mathbf{x}) = W \log_2 \left(1 + \frac{P_i}{\sum_{x_j \in \mathrm{Ap}_i} I(x_i, x_j) \frac{P_j}{d_{i,j}^2}} \right). \tag{22}$$

The goal is now to find an assignment $\mathbf{x} \in \mathcal{D}$ such that

$$\mathbf{x} = \arg\max_{\mathbf{y}} \sum_i f_i([\mathbf{y}]_{f_i}). \tag{23}$$

5.2 Experimental Setup

All evaluated algorithms are implemented in the FRODO platform (Léauté et al., 2009) and evaluated in three problem classes, with various parameters. For each class of problems and parameter settings, 1090 problem instances were generated using the generators provided by FRODO. The first 100 were used to select algorithm parameters, and then we plotted the final results on the remaining 990 instances (99 for each parameter value) . Each algorithm was given a maximum of 15 minutes of simulated time per problem. The experiments are run on a 64 core Linux machine, with 2Gb per run.

Experimental protocol DSA, and the DUCT variants have a number of hyperparameters, which affect how the space is searched and when the algorithm terminates. In order to perform an *unbiased* comparison between the algorithms, we performed a grid search for the best-performing hyperparameters in terms of utility on a small set of problem instances, and then tested on

a larger problem. The procedure was uniformly applied to all algorithms, to ensure a fair comparison.

More specifically, the following steps were taken; (a) For each problem class C, we generated a set $\Pi_{\text{Train}}(C)$ of 100 instances with different problem parameters. (b) For each algorithm $\mathscr{A}$, we performed a grid search, varying the hyperparameters $\theta \in \Theta$, where Θ is a discrete set of hyperparameter choices for the algorithm $\mathscr{A}$. We then measured its average utility over those instances $\bar{u}(\mathscr{A}, \theta, \Pi_{\text{Train}}(C)) = 10^{-2} \sum_{\pi \in \Pi_{\text{Train}}(C)} u(\mathscr{A}, \theta, \pi)$. For each problem class and algorithm combination, we obtained an estimated best parameter $\theta^*(\mathscr{A}, C) \in \arg\max_\theta \bar{u}(\mathscr{A}, \theta, \Pi_{\text{Train}}(C))$. Finally (c) we measured the performance of each algorithm on a much larger set of instances $\Pi_{\text{Test}}(C)$, of size 10^4, i.e. we measured the utility $u(\mathscr{A}, \theta^*(\mathscr{A}, C))$, as well as other statistics, for the selected hyperparameter.

Algorithm hyperparameters The hyperparameters we tuned according to the protocol were (a) the stopping conditions for DUCT and (b) the stochasticity of DSA. Our selections for these parameters for each problem can be seen in Table 2. Although sometimes the optimal hyperparameters selected are very

	Meeting Scheduling	Channel Allocation	Graph Colouring
DSA	p = 0.3, strategy=A	p = 0.9, strategy=A	p = 0.3, strategy=A
MGM2	q = 0.5	q=0.5	q = 0.5
RANDOM	$\delta = 0.8, \epsilon = 0.7$	$\delta = 0.5, \epsilon = 0.2$	$\delta = 0.2, \epsilon = 0.9$
DUCT-A	$\delta = 0.6, \epsilon = 0.2$	$\delta = 0.2, \epsilon = 0.9$	$\delta = 0.6, \epsilon = 0.9$
DUCT-B	$\delta = 0.1, \epsilon = 0.6$	$\delta = 0.9, \epsilon = 0.5$	$\delta = 0.4, \epsilon = 0.7$
DUCT-C	$\delta = 0.1, \epsilon = 0.1$	$\delta = 0.2, \epsilon = 0.1$	$\delta = 0.2, \epsilon = 0.1$
DUCT-D	$\delta = 0.6, \epsilon = 0.1$	$\delta = 0.1, \epsilon = 0.1$	$\delta = 0.1, \epsilon = 0.1$

Table 2: The selected hyper-parameters.

different, the DUCT algorithms were not very sensitive to their choice.

Problem parameters Each problem class has different parameters. For *meeting scheduling*, we used a pool of 30 agents, with 3 agents per meeting. The parameter was the number of meetings per instance, which we varied from 11 to 20. For each agent and for each of the possible 8 timeslots, the cost to that agent of having any meeting in that timeslot was randomly drawn from $[0, 10]$. For the *graph colouring* problems, graphs with a density of 0.4 were randomly generated, with 3 available colours. The parameter was the number of nodes per instances, which we varied from 20 to 30. Problems were modelled as Min-CSPs. For the *channel allocation* problems, the access points were randomly placed on a map of size 100x100, with a minimal distance of 4 units. The power of an access point ranges between 490 and 510, $W = 20$, the number of channels is set to 6 and $N_b = 1$. The parameter was the number of nodes in the network, which we varied from 7 to 15.

Reported metrics For each case, we report the median over the 99 instances for each parameter setting, together with 95% confidence intervals. The reason that the DUCT algorithms appear to find better solutions than the optimal algorithms, is because the optimal algorithms time out for the bigger instances, and consequently, their median solution quality can be worse than the median for some DUCT variants.

Termination condition for DSA, MGM and MGM-2 The algorithms DSA, MGM and MGM-2 don't have a natural termination condition. In our experiments we let the models run until the algorithm has converged, or a 15 minute timeout was reached. Convergence in this setting means that none of the agents will change its value if the algorithm continues running. In our implementation we make use of the fact that we run the entire algorithm in a single thread to efficiently detect this.

5.3 Experimental Results

The results for the meeting scheduling domain are shown in Figure 3 on page 32, those of the graph colouring domain in Figure 4 on page 33, while the channel allocation results are presented in Figure 5 on page 34. In all cases, sub-figures (a) and (b) show the cost of DUCT variants and comparison of DUCT-D with other approaches, respectively, while subfigures (c) and (d) show the time and information complexity of the various algorithms. Figure 6 shows the number of instances solved for the meeting scheduling and channel allocation problems.

Figure 3(a) shows the performance of the different DUCT variants on the meeting scheduling problem. It is clear that DUCT-D finds the best solution, although Figure 3(c) shows that in terms of runtime DUCT-D is outperformed by both DUCT-A and DUCT-B. In general, the DUCT versions with $\lambda_a = 1$ terminate faster than the DUCT versions with $\lambda_a =$ path length, for the simple reason that the latter versions need more samples to make the confidence bounds small enough. That increasing the amount of time spent sampling does not lead to better solutions can be seen from the fact that although RANDOM takes the most time, it doesn't find the best solution. It is the steering of the search using the UCB bounds that leads to a better solution.

When compared to the other algorithms on the meeting scheduling problem (Figure 3(b)), one can see that DUCT-D finds the best known solution most of the time. In fact, 93% of the solutions found fall within the 5% error range set as the target for the DUCT algorithms. This means that the assumptions underlying the bound used characterise the search space well. Where the DUCT variants are able to find a solution to most scheduling problems, DPOP is only able to solve problems with up to 14 meetings, after which it times out on an increasing fraction of problems. DPOP can have a higher median cost than DUCT, due to the fact that the costs for runs where DPOP timed out are set to infinity. The local-search algorithms, given as much time as DUCT-D, failed to find a feasible solution in more than 50% of the instances while DUCT-D always found a feasible schedule (See Figure 6(a)). To still give an idea of the

performance of local search algorithms, Figure 3(b) contains only the solved solutions.

Figure 4 shows the solution quality and runtime for the graph colouring problem. In terms of the solution quality found, DUCT D, C and Random find equally good solutions (Figure 4(a)). The performance of the local-search algorithms is rather poor for the smaller problems, but for the bigger problems they are on par with DUCT-D (Figure 4(b)), with MGM2 finding better solutions than the DUCT algorithms. When looking at runtime (Figure 4(c)), it is clear that the DUCT variants scale much better than DPOP, which fails to solve problems bigger than 24 nodes in the allotted time. DUCT-C and DUCT-D consistently run faster than DUCT-A, DUCT-B and RANDOM, with DUCT-C being indistinguishable from DUCT-D. In the graph colouring setting, the influence of the value of λ_a is thus of less importance, while considering the child bounds brings a significant speed up.

For the channel allocation problem (Figure 5(a), 5(b)), there is little difference between the algorithms in terms of final cost. When comparing the runtime of the DUCT variants to that of DPOP, it is clear that all DUCT variants scale much better. In fact, as can be seen from Figure 6(b), while the number of problems that DPOP solves drops rapidly, the DUCT variants are able to solve all instances.

Compared to the other DUCT variants, DUCT-D has the most consistent performance, in terms of runtime, average solution quality, number of found solutions and amount of information exchanged. It thus appears that taking into account node depth as well as child bounds is necessary for obtaining good solutions quickly, since either measure results in only a small improvement by itself. In addition, its performance improvement over RANDOM in two of the three problems shows that stochasticity is not sufficient for good results.

Overall, DUCT-D always matches or comes close to the best alternative algorithm in terms of median cost. In addition, it seems to be more robust, with the highest proportion of solved instances found. Most importantly for truly distributed applications, it always transmits less information compared to any other complete algorithm. It doesn't always send less information than the local search algorithms, but since it is not always feasible to efficiently detect termination of all variables the amount of information sent by local search algorithms should be seen as a lower bound. After our unbiased tuning procedure, all other complete methods use much more information than our algorithm. Finally, this amount of information exchanged scales extremely well as the problem size increases. We believe this to be the most interesting feature of this family of algorithms, which in our view is overall extremely robust and scalable.

6 Conclusions

We introduced an upper confidence bound based approach for solving Distributed Constraint Optimisation Problems, inspired by Monte-Carlo tree search and the application of UCB to tree-structured problems. The result, a Dis-

tributed UCT algorithm (DUCT), takes advantage of the distributed and deterministic nature of the problem and the hard constraints to efficiently search the solution space.

We show that even though DCOP is not a smooth domain, we can still obtain meaningful bounds on the regret. However, the gap between the upper and lower bounds indicates that even better results may be possible. It is our view that a more intricate analysis requires additional assumptions, which we shall investigate in further work.

The experiments clearly show that DUCT not only can obtain feasible and low-cost solutions within a reasonable amount of time, but also that for most problem instances it performs at least as well as local search. We also show that DUCT can handle much bigger problems than optimal algorithms, while it performs just as well for smaller domains. The fact that it consistently transmits a much smaller amount of information than other algorithms, makes it a very suitable algorithm for practical use in distributed environments, where communication, rather than local computation, is the principal bottleneck. In all, DUCT is thus very well suited for solving DCOPs.

Acknowledgement. The authors would like to thank the anonymous reviewers for their helpful suggestions. The work was supported by the a Marie Curie Intra-European Fellowship for the project *Efficient Sequential Decision Making Under Uncertainty* (ESDEMUU) under Grant No. 237816 and the Swiss National Science Foundation for the project *Distributed and Multi-agent Constraint Optimization in Dynamic Environments* under Grant No. 200020_129515

References

S. Ali, S. Koenig, and M. Tambe. 2005. Preprocessing Techniques for Accelerating the DCOP Algorithm ADOPT. In *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'05)*. Utrecht, Netherlands, 1041–1048.

Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. 2002. Finite Time Analysis of the Multiarmed Bandit Problem. *Machine Learning* 47, 2/3 (2002), 235–256.

Balabhaskar Balasundaram and Sergiy Butenko. 2006. Graph Domination, Coloring and Cliques in Telecommunications. In *Handbook of Optimization in Telecommunications*. 865–890.

Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and Csaba Szepesvári. 2011. X-Armed Bandits. *Journal of Machine Learning Research* 12 (2011), 1655–1695.

Nicolò Cesa-Bianchi and Gábor Lugosi. 2006. *Prediction, Learning and Games*. Cambridge University Press, Cambridge, UK.

Pierre-Arnaud Coquelin and Rémi Munos. 2007. Bandit Algorithms for Tree Search. In *Proceedings of the 23rd Conference Conference on Uncertainty in Artificial Intelligence (2007)*, Ron Par and Linda van der Gaag (Eds.). Vancouver, Canada.

Rina Dechter. 2006. *Tractable Structures for Constraint Satisfaction Problems.* Elsevier Science Inc., New York, NY, USA, Chapter 7, 209–244.

Rina Dechter and Robert Mateescu. 2004. Mixtures of Deterministic-Probabilistic Networks and their AND/OR Search Space. In *UAI04*. 120–129.

Stephen Fitzpatrick and Lambert Meertens. 2003. Distributed coordination through anarchic optimization. In *Distributed Sensor Networks*. Springer, 257–295.

Sylvain Gelly and David Silver. 2005. Achieving Master Level Play in 9 x 9 Computer Go. In *Proceedings of the 23th National Conference on Artificial Intelligence (AAAI'08)*. Chicago, USA, 1537–1540.

Amir Gershman, Amnon Meisels, and Roie Zivan. 2006. Asynchronous Forward-Bounding for Distributed Constraints Optimization. In *Proceedings of the 17th Eureopean Conference on Artificial Intelligence (ECAI'06),*. 103–107.

Vibhav Gogate and Rina Dechter. 2003. A New Algorithm for Sampling CSP Solutions Uniformly at Random. In *Proceedings of the 12th Internation Conference on Principles and Practice of Constraint Programming - CP 2006*. Nantes, France, 711–715.

Youssef Hamadi. 1998. Backtracking in Distributed Constraint Networks. In *International Journal on Artificial Intelligence Tools*. 219–223.

Katsutoshi Hirayama and Makoto Yokoo. 2003. Distributed partial constraint satisfaction problem. In *Proceedings of the 9th Internation Conference on Principles and Practice of Constraint Programming - CP 2003*. Kinsale, Ireland, 222–236.

Robert Kleinberg. 2005. Nearly Tight Bounds for the Continuum-armed Bandit Problem. In *Advances in Neural Information Processing Systems 17*. MIT Press, 697–704.

Levente Kocsis and Csaba Szepesvári. 2006. Bandit based Monte-Carlo Planning. In *Proceedings of the 17th European Conference on Machine Learning, (ECML06)*.

Thomas Léauté and Boi Faltings. 2011a. Coordinating Logistics Operations with Privacy Guarantees, See Walsh (2011), 2482–2487.

Thomas Léauté and Boi Faltings. 2011b. Distributed Constraint Optimization under Stochastic Uncertainty. In *Proc. of the 25th National Conference on Artificial Intelligence (AAAI'11)*. San Francisco, USA, 68–73.

Thomas Léauté, Brammert Ottens, and Boi Faltings. 2010. Ensuring Privacy through Distributed Computation in Multiple-Depot Vehicle Routing Problems. In *Proceedings of the ECAI'10 Workshop on Artificial Intelligence and Logistics (AILog'10)*.

Thomas Léauté, Brammert Ottens, and Radoslaw Szymanek. 2009. FRODO 2.0: An Open-Source Framework for Distributed Constraint Optimization. In *Proceedings of the IJCAI'09 Distributed Constraint Reasoning Workshop (DCR'09)*. 160–164.

Rajiv T. Maheswaran, Jonathan P. Pearce, and Milind Tambe. 2004. Distributed Algorithms for DCOP: A Graphical-Game-Based Approach. In *Proceedings of ISCA PDCS'04*. 432–439.

R. T. Maheswaran, M. Tambe, E. Bowring, J. P. Pearce, and P. Varakantham. 2003. Taking DCOP to the Real World: Efficient Complete Solutions for Distributed Multi-Event Scheduling. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'03)*. Melbourne, Australia, 310–317.

Pragnesh J. Modi, Wei-Min Shen, Milind Tambe, and Makoto Yokoo. 2005. ADOPT: Asynchronous Distributed Constraint Optimization with Quality Guarantees. *Artificial Intelligence* 161 (2005), 149–180.

Duc Thien Nguyen, William Yeoh, and Hoong Chuin Lau. 2013. Distributed Gibbs: a memory-bounded sampling-based DCOP algorithm. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*. International Foundation for Autonomous Agents and Multiagent Systems, 167–174.

Brammert Ottens, Christos Dimitrakakis, and Boi Faltings. 2012. DUCT: An Upper Confidence Bound Approach to Distributed Constraint Optimization Problems. In *Proceedings of the 26th National Conference on Artificial Intelligence (AAAI'12)*. Toronto, Canada.

Brammert Ottens, Christos Dimitrakakis, and Boi Faltings. 2017. DUCT: An Upper Confidence Bound Approach to Distributed Constraint Optimization Problems. *ACM Trans. Intell. Syst. Technol.* 8, 5, Article 69 (July 2017), 27 pages. DOI:http://dx.doi.org/10.1145/3066156

Brammert Ottens and Boi Faltings. 2008a. Asynchronous Open DPOP. In *Proceedings of the Tenth International Workshop on Distributed Constraint Reasoning - AAMAS08*. Estorial Portugal. http://liawww.epfl.ch/Publications/Archive/Ottens2008.pdf

Brammert Ottens and Boi Faltings. 2008b. Coordinating Agent Plans Through Distributed Constraint Optimization. In *Proceedings of the ICAPS'08 Multiagent Planning Workshop (MASPLAN'08)*.

Adrian Petcu. 2006. *FRODO: A FRamework for Open/Distributed constraint Optimization.* Technical Report 2006/001. Swiss Federal Institute of Technology (EPFL), Lausanne (Switzerland). 10 pages.

Adrian Petcu and Boi Faltings. 2005. DPOP: A Scalable Method for Multiagent Constraint Optimization. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI'05)*. 266–271.

Adrian Petcu and Boi Faltings. 2006. O-DPOP: An algorithm for Open/Distributed Constraint Optimization. In *Proceedings of the 21th National Conference on Artificial Intelligence (AAAI'06)*. Bosten, U.S.A., 703–708.

Pierre Rust, Gauthier Picard, and Fano Ramparany. 2016. Using Message-passing DCOP Algorithms to Solve Energy-efficient Smart Environment Configuration Problems. In *International Joint Conference on Artificial Intelligence*.

Bab Satomi, Yongjoon Joe, Atsushi Iwasaki, and Makoto Yokoo. 2011. Real-Time Solving of Quantified CSPs Based on Monte-Carlo Game Tree Search, See Walsh (2011), 655–661.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, and others. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (2016), 484–489.

R. Stranders, L. Tran-Tranh, F. Fave, A. Rogers, and N. Jennings. 2012. DCOPs and Bandits: Exploration and Exploitation in Decentralised Coordination. In *Proceedings of the 11th International Conference on Autonomous Agents and Multi Agent Systems*. Valencia, Spain.

Evan A. Sultanik, Robert N. Lass, and William C. Regli. 2007. DCOPolis: A Framework for Simulating and Deploying Distributed Constraint Optimization Algorithms. In *Proceedings of the 9th Intl Workshop on Distributed Constraint Reasoning (CP-DCR'07)*.

Toby Walsh (Ed.). 2011. *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI'11)*. AAAI Press, Barcelona, Spain.

William Yeoh, Ariel Felner, and Sven Koenig. 2010. BnB-ADOPT: An Asynchronous Branch-and-Bound DCOP Algorithm. *Journal of Artificial Intelligence Research (JAIR)* 38 (2010), 85–133.

M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara. 1998. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *Knowledge and Data Engineering, IEEE Transactions on* 10, 5 (sep/oct 1998), 673 –685.

Weixiong Zhang, Guandong Wang, Zhao Xing, and Lars Wittenburg. 2003. A comparative study of distributed constraint algorithms. In *Distributed Sensor Networks*. Springer, 319–338.

A Appendix

A.1 Collected proofs

Theorem 1. The main idea is to remove any inherent structure in the problem via randomisation. This would make any algorithm equivalent to RANDOM. To prove the lower bound, we can restrict our attention to the following example. Consider a function g in only one variable, for which Assumption 1 holds with equality. Assume a randomly chosen bijection $h : \mathcal{D} \to \mathcal{D}$. (Note that this must be over the complete domain, rather than each individual domain). Because h is a bijection, the function $f(\mathbf{x}) = g(h(\mathbf{x}))$ also satisfies Assumption 1. Since the bijection is random, any algorithm is equivalent to RANDOM. Since we take the worst case, γ is the probability that we choose an infeasible solution and $\gamma\epsilon^{-\beta}$ is the probability that the solution is not ϵ-optimal. Consequently,

$$\mathbb{E}\,\rho_T = \mathbb{E}(\rho_T \mid \rho_T \geq \epsilon)\,\mathbb{P}(\rho_T \geq \epsilon) + \mathbb{E}(\rho_T \mid \rho_T < \epsilon)\,\mathbb{P}(\rho_T < \epsilon) \tag{24}$$

$$\geq \epsilon \cdot (\gamma\epsilon^{-\beta})^T + 0 = \gamma^T \epsilon^{1-\beta T}. \tag{25}$$

Selecting $\epsilon = \gamma^{1/2\beta}$ completes the proof. □

Theorem 2. Since we sample uniformly with respect to the λ measure, the probability that we sample an optimal value at any given round is at least $\lambda^*/\bar{\lambda}$ by Assumption 2. If we do, then our regret is zero. Consequently,

$$\mathbb{P}(\rho_T > 0) \leq (1 - \lambda^*/\bar{\lambda})^T. \tag{26}$$

Now consider the case where we have not sampled an optimal value after T rounds. Then via Ass. 1, the probability that the regret exceeds ϵ is bounded by the probability that all our T last samples are ϵ-far from the optimal:

$$\mathbb{P}\left(\rho_T > \epsilon \mid \rho_T > 0\right) = \mathbb{P}\left(\bigwedge_{t=1}^{T} f\left(\mathbf{x}_t\right) > f^*(\mathcal{D}) + \epsilon \;\middle|\; \bigcup_{t=1}^{T} \{\mathbf{x}_t\} \cap \mathcal{D}^* = \emptyset\right)$$

$$\leq \left(\gamma\epsilon^{-\beta}\right)^T. \tag{27}$$

This follows from the fact that the probability of sampling a value worse than ϵ-optimal at any step is bounded by $\gamma\epsilon^{-\beta}$ by Assumption 1. The second case is that we have sampled an optimal value sometime in the previous T rounds. Then our regret is zero. Using the fact that, for any $\epsilon > 0$, the event $\rho_T > 0$ is contained within $\rho_T > \epsilon$, and equations (26) and (27) the overall probability that the regret is greater than ϵ is:

$$\mathbb{P}\left(\rho_T > \epsilon\right) = \mathbb{P}\left(\rho_T > \epsilon \wedge \rho_T > 0\right) = \mathbb{P}\left(\rho_T > \epsilon \mid \rho_T > 0\right)\mathbb{P}(\rho_T > 0) \tag{28}$$

$$\leq (\gamma\epsilon^{-\beta})^T (1 - \lambda^*/\bar{\lambda})^T. \tag{29}$$

Setting $\epsilon = \left(\gamma\delta^{-1/T}(1 - \lambda^*/\bar{\lambda})\right)^{1/\beta}$ we prove the first inequality.

The second part of the theorem can be proven as follows. From (27) and the fact that the regret and probabilities are bounded in $[0, 1]$ we have that

$\forall\epsilon$, $\mathbb{E}\,\rho_T \leq \left(\gamma\epsilon^{-\beta}\right)^T + \epsilon$. Setting the derivative to zero, we find that $\epsilon_0 = \left(\beta T\gamma^T\right)^{1/(\beta T+1)}$. Replacing to find the tightest bound, we obtain the following inequality.

$$\mathbb{E}\,\rho(T) \leq \gamma^T(\beta T\gamma^T)^{-\frac{\beta T}{\beta T+1}} + (\beta T\gamma^T)^{\frac{1}{\beta T+1}} \leq \frac{1}{\beta T} + (\beta T\gamma^T)^{\frac{1}{\beta T+1}}. \tag{30}$$

Now note that $\max_x x^{1/(1+x)} < \max_x x^{1/x}$ and that:

$$\frac{d}{dx}x^{1/x} = \frac{d}{dx}e^{\frac{1}{x}\ln x} = e^{\frac{1}{x}\ln x}x^{-2}(1-\ln x),$$

which has a root $x = e$. Consequently, $(\beta T)^{1/(\beta T+1)} < e^{1/e}$. Finally, $\gamma^{T/(\beta T+1)} \leq \gamma^{1/(\beta+1)}$. □

Theorem 3. Note that the algorithm does not stop until the root node has stopped. For any $\epsilon > 0$, setting $\delta = 2\epsilon e^{\bar{\lambda}}$ guarantees that the algorithm can only stop when the complete space has been searched. To see this, note that all values are in $[0, 1]$. Say Δ is the difference between the first and second choice so far. Then the condition becomes:

$$0 \leq \Delta - \sqrt{\ln(2/\delta)/\tau} + \epsilon \tag{31}$$

$$\ln(2/\delta) \leq (1+\epsilon)^2\tau \tag{32}$$

$$\delta \leq \exp[-(1+\epsilon)^2\tau]/2. \tag{33}$$

Since $\tau < \bar{\lambda}$, the result follows. □

A.2 Theorem 4

In order to bound the regret of DUCT, we first need some auxiliary results. We begin by proving a lemma on the best value $f^*(A)$ of any set A. Assume that we have taken n samples from A, with values $(y_k^1, \ldots, y_k^n)$. In the worst-case, these are the n worst values, allowing us to bound the error. Formally:

Lemma 1. *Under Ass. 1, after taking $n < \lambda(A)$ non-identical samples $X_n = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\}$ from A, with values $Y_T = \left\{y_A^1, \ldots, y_A^n\right\}$, it holds that:*

$$\min Y_n - f^*(A) \leq \left(\frac{\gamma\lambda(A)}{n}\right)^{1/\beta}. \tag{34}$$

Proof. In the worst-case, these are the n worst values, so the result follows by re-arranging equation (17) in Ass. 1. □

For a given problem, the values of γ and β are unknown. Taking $\beta = 2$, and letting γ slowly increase over time will result in these bounds eventually holding. This is the intuition behind the confidence bound we have used.

The second lemma we need deals with the regret at a binary node in the AND/OR tree.

Lemma 2. *Consider two disjoint sets $A_1, A_2 \subset \mathcal{D}$ with $f^*(A_1) = f^*(A_2) + \Delta$, $A \triangleq A_1 \cup A_2$. If A is sampled T times, then* DUCT *only samples the sub-optimal set A_1 at most $16\Delta^{-2} \ln T + \kappa_1$ times, where κ_1 only depends on γ, β.*

Lemma 2. We split the analysis in two parts. The first Let $\hat{\mu}_i^t$ be the best-measured value in A_i at time t. Since our confidence intervals $L_{i,t}$ are heuristic, they do not necessarily contain the optimal value in the interval A_i. However Lemma 1 ensures that they will do so, when $L_{i,t} \geq (\gamma \lambda(A)/T)^{1/\beta}$. Using the definition of $L_{i,t}$ from equation (7), we see that it is sufficient to sample $e^{\gamma/2}$ times interval A, and sample A_i

$$\kappa_1 = (\gamma/2)^{\beta/(2-\beta)}$$

times. In that case, our confidence intervals contain the optimal value in A_i, i.e.

$$f^*(A_i) \in [\hat{\mu}_i^t - L_{i,t}, \hat{\mu}_i^t],$$

where the upper bound holds trivially.

If we select A_1 rather than A_2, we have: $f^*(A_2) \geq \hat{\mu}_2^t - L_{2,t} \geq \hat{\mu}_1^t - L_{1,t}$. But $\hat{\mu}_1^t \geq f^*(A_1) - L_{1,t}$. Consequently, in order to select A_1 it is necessary that $f^*(A_2) \geq f^*(A_1) - 2L_{1,t}$. From the definition, we obtain $T_i \leq \left(\frac{4}{\Delta}\right)^2 \ln T$, for a total bound of

$$\left(\frac{4}{\Delta}\right)^2 \ln T + (\gamma/2)^{\beta/(2-\beta)} + e^{\gamma/2}.$$

□

Whenever we sample the suboptimal interval, we suffer regret at least Δ and at most $\kappa_2 \Delta \leq 1$, for some $\kappa_2 > 0$. By multiplying with $\kappa_2 \Delta$, where $\kappa_2 \leq \lambda(A_1)\gamma$, we obtain the total regret during exploration of the suboptimal interval. This is summarised in the following corollary.

Corollary 1. *The worst-case total regret due to sampling in the sub-optimal interval is bounded by:*

$$R_T \leq \kappa_2 \left(\frac{16 \ln T}{\Delta} + \Delta \kappa_1 \right). \tag{35}$$

We now introduce a simple lemma to relate the simple regret to the total regret:

Lemma 3. *If there exists some function g such that $R_T \leq g(T)$ and $r_t \in [0, 1]$ for all t then:*

$$\rho_T \leq g(T)/T. \tag{36}$$

Proof. First recall that the regret due to a particular sequence r_t is $\rho_T = \min_{t=1,\dots T} r_t$. Taking the maximum over all such sequences, we obtain

$$\max_{\{r_t\}} \min_t \{r_t \,|\, R_T \leq g(T), r_t \in [0,1]\},$$

which is has unique solution $r_t = g(T)/T$ for all t. □

This directly implies the following:

Lemma 4. *The simple regret for sampling the sub-optimal interval is:*

$$\rho_T^1 \leq \frac{\kappa_2}{T}\left(\frac{\ln T}{\Delta} + \Delta\kappa_1\right). \tag{37}$$

We are now ready to prove the main theorem.

Theorem 4. Let ρ_T^1, ρ_T^2 be the regret due to sampling the sub-optimal and optimal interval respectively. Then the regret is:

$$\rho_T = \min\left\{\rho_T^1, \rho_T^2\right\} \tag{38}$$

$$= \min\left\{\frac{\kappa_2}{T}\left(\frac{16\ln T}{\Delta} + \Delta\kappa_1\right), \left(\frac{\gamma\lambda(A_2)}{T_2}\right)^{1/\beta}\right\}, \tag{39}$$

from Lemma 4 and Lemma 1. We complete the proof by noting that $T_2 = T - T_1 \geq T - 16\Delta^2 \ln T - \kappa_1$. □

A.3 Additional experimental results.

In this section we add some additional results for the number of solved instances in the two problems where not all solutions are feasible in Figure 6. In the meeting scheduling problem (Fig. 6(a)) we see that feasible solutions are only found reliably in the available time with either DUCT or the random algorithm. In the channel allocation problem (Fig. 6(b)) we can see that most methods manage to terminate with a feasible solution within the time given. The main reason that DPOP fails appears to be that it doesn't always have sufficient time to terminate.

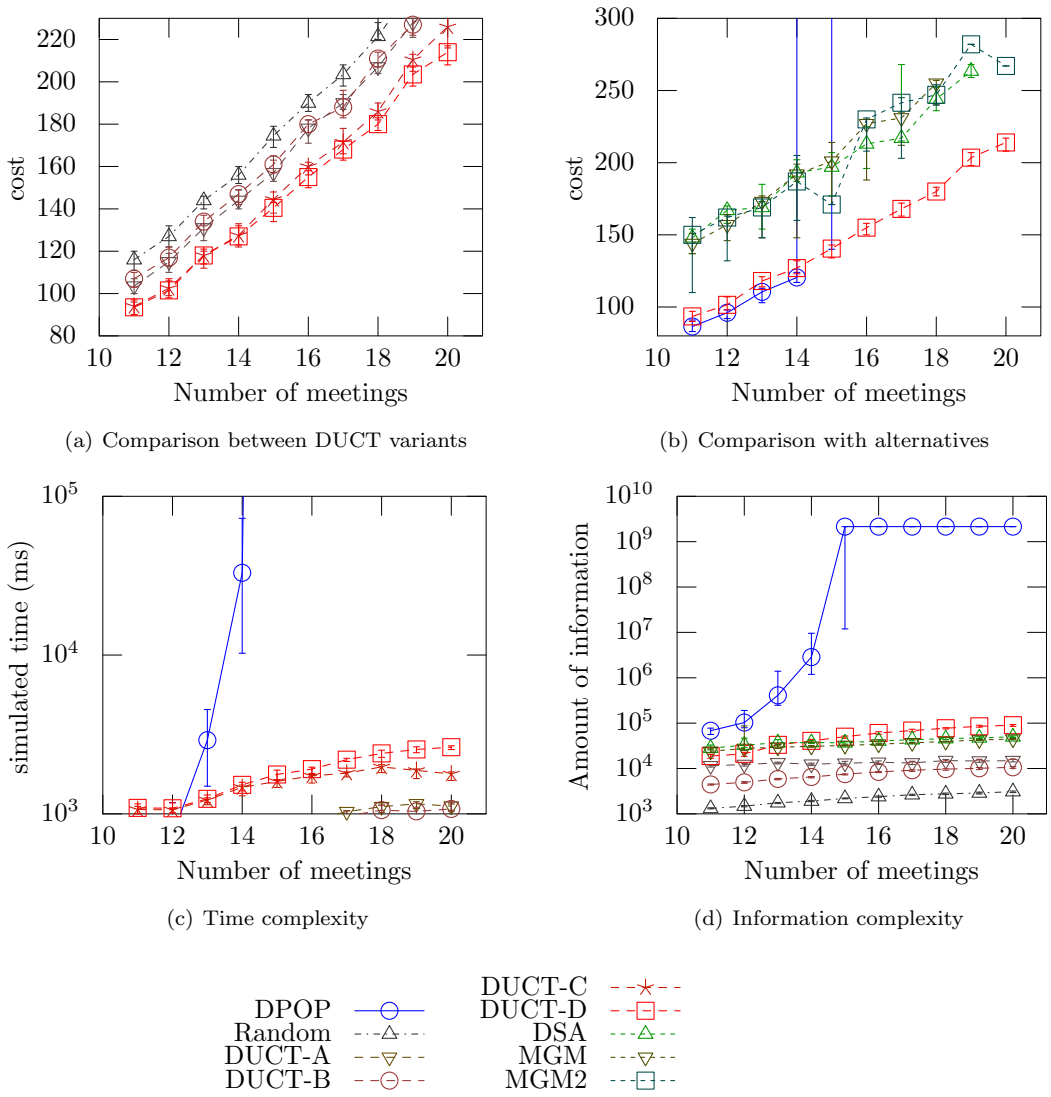


Figure 3: **Meeting scheduling** results. Figures 3(a) shows the cost achieved (i.e. lower scores are better) by DUCT variants and RANDOM. Figure 3(b) compares the cost of DUCT-D with alternative approaches. Simulated time to solution is shown in Figure 3(c) while the total information exchanged for each solution is shown in Figure 3(d)

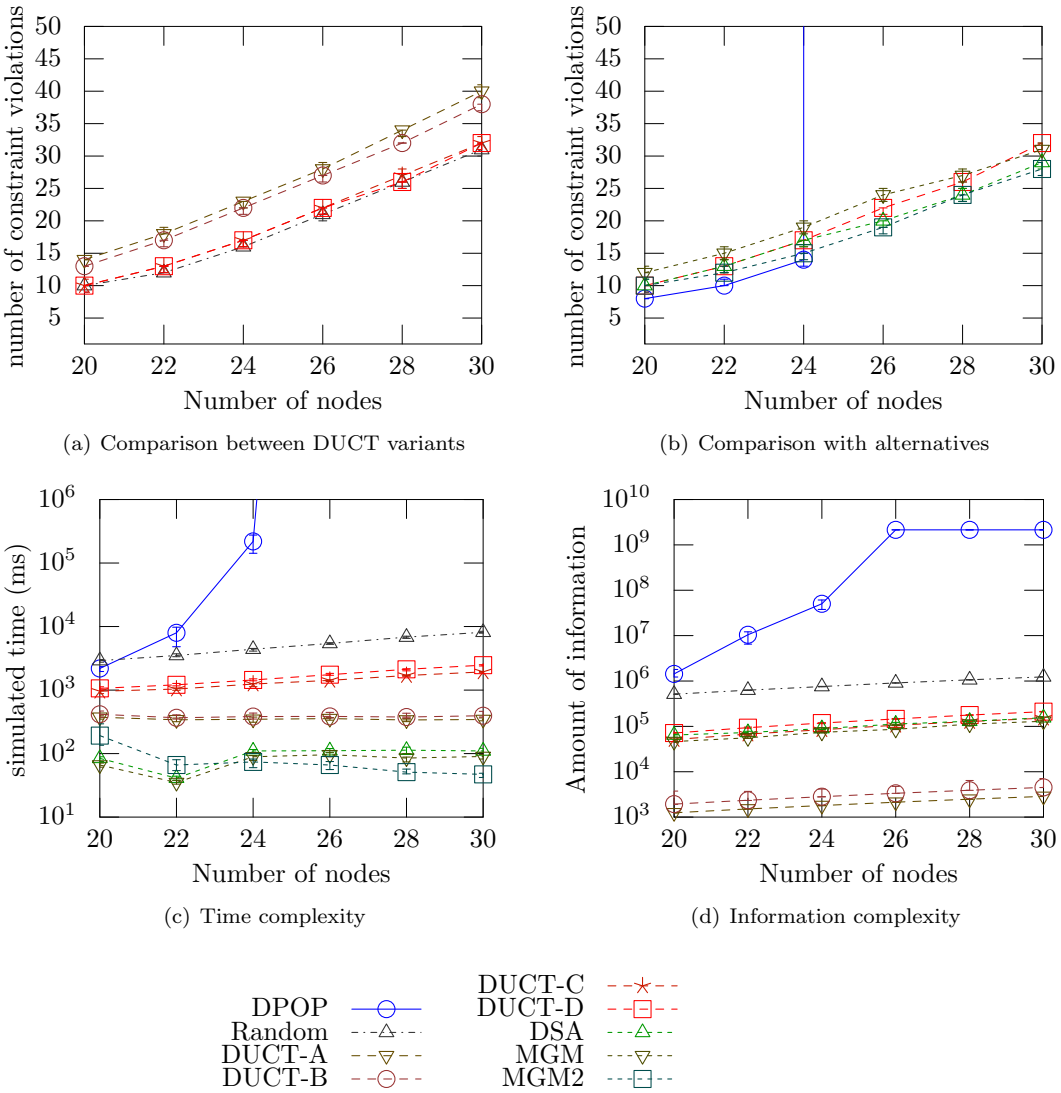
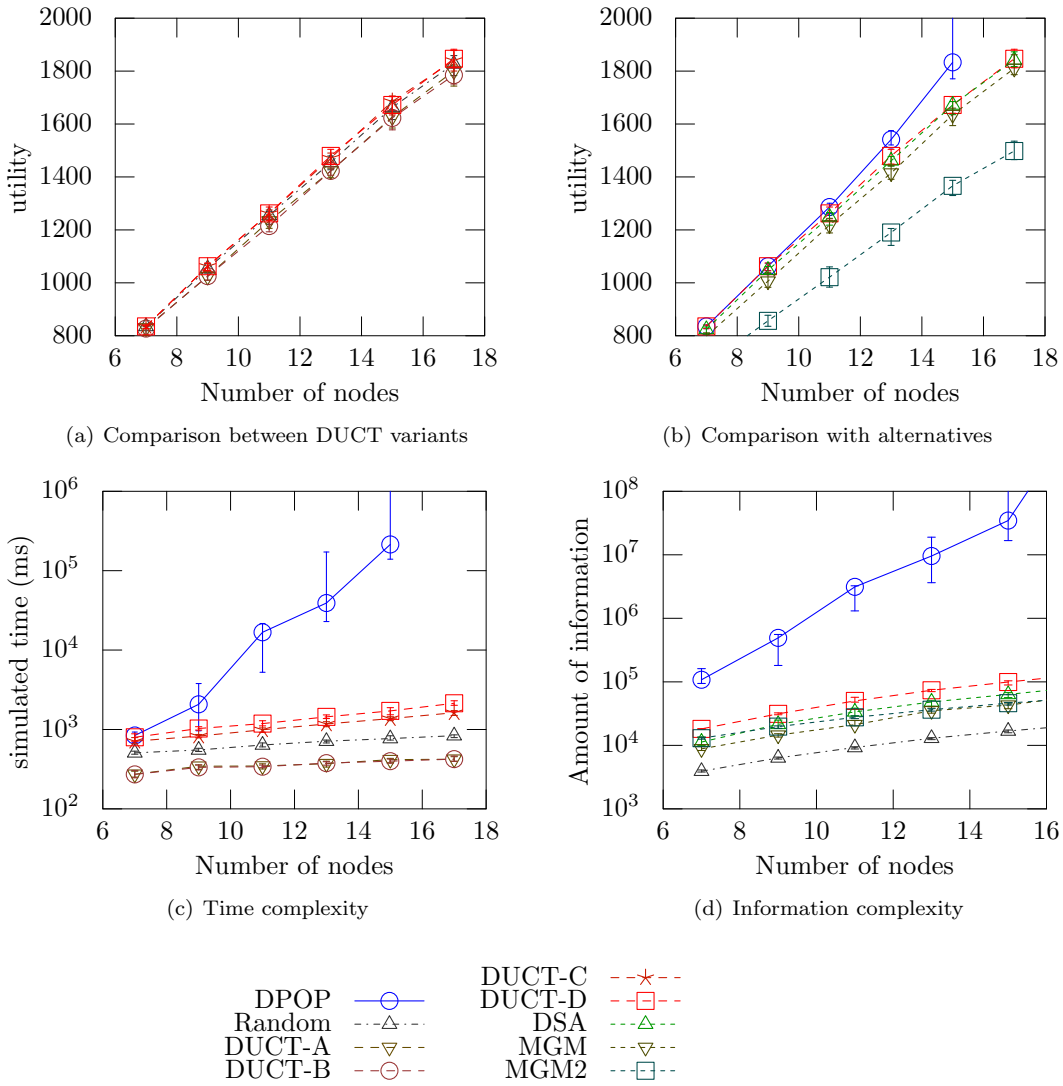


Figure 4: **Graph colouring** results. Figures 4(a) shows the number of violations (i.e. lower scores are better) by DUCT variants and RANDOM. Figure 4(b) compares the solution quality of DUCT-D with alternative approaches. Simulated time to solution is shown in Figure 4(c) while the total information exchanged for each solution is shown in Figure 4(d)



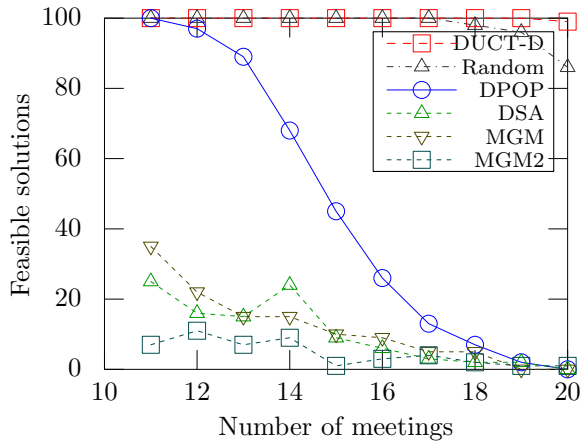
(a) Comparison between DUCT variants

(b) Comparison with alternatives

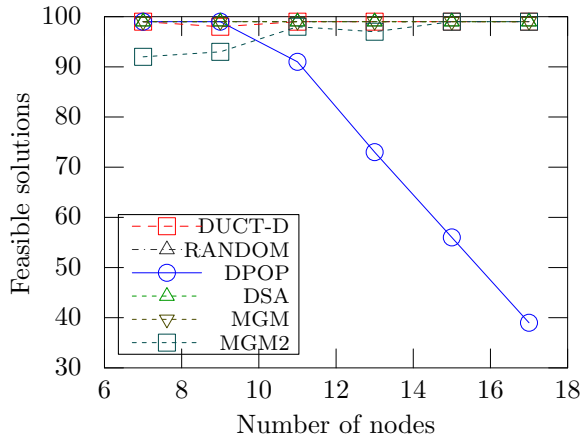
(c) Time complexity

(d) Information complexity

Figure 5: **Channel allocation** results. Figure 5(a) shows the utility (i.e. the higher scores are better) achieved by DUCT variants and RANDOM. Figure 5(b) compares DUCT-D to local methods. Simulated time to solution is shown in Figure 5(c) while the total information exchanged for each solution is shown in Figure 5(d)



(a) Meeting scheduling



(b) Channel allocation

Figure 6: Number of solved instances for the meeting scheduling and channel allocation problems.