# Dynamic Configuration of Agent Organizations

**Evan A. Sultanik** **Robert N. Lass** **William C. Regli**
Department of Computer Science
Drexel University
3141 Chestnut St.
Philadelphia, PA 19104
{eas28, urlass, regli}@cs.drexel.edu

## Abstract

It is useful to impose organizational structure over multiagent coalitions. Hierarchies, for instance, allow for compartmentalization of tasks: if organized correctly, tasks in disjoint subtrees of the hierarchy may be performed in parallel. Given a notion of the way in which a group of agents need to interact, the Dynamic Distributed Multiagent Hierarchy Generation (DynDisMHG) problem is to determine the best hierarchy that might expedite the process of coordination. This paper introduces a distributed algorithm, called Mobed, for both constructing and maintaining organizational agent hierarchies, enabling exploitation of parallelism in distributed problem solving. The algorithm is proved correct and it is shown that individual additions of agents to the hierarchy will run in an amortized linear number of rounds. The hierarchies resulting after perturbations to the agent coalition have constant-bounded edit distance, making Mobed very well suited to highly dynamic problems.

## 1 Introduction

It is useful to impose organizational structure over multiagent coalitions. Hierarchies, for instance, allow for compartmentalization of tasks: if organized correctly, tasks in disjoint subtrees of the hierarchy may be performed in parallel. Intuitively, the shallower the hierarchy the more subordinates per manager, leading to more potential for parallelism. The difficulty lies in determining a minimum depth hierarchy that is isomorphic to the problem being solved; in a business, for example, there is very little sense in assigning an accountant from the billing department as the superior of a marketing associate. Given a notion of the way in which the agents need to interact, the initial problem, then, is to determine the best hierarchy that might expedite the process of coordination. This is called the Dynamic Distributed Multiagent Hierarchy Generation (DynDisMHG) problem.

Solutions to the DynDisMHG problem currently have direct application in the field of multiagent systems, including distributed problem solving [Shehory and Kraus, 1998], cooperative multiagent systems [Sycara *et al.*, 1996], distributed constraint reasoning (DCR), command and control, mobile *ad hoc* networks (MANETs), sensor nets, and manufacturing. For example, the computation time required by most complete DCR algorithms is determined by the topology of a hierarchical ordering of the agents [Silaghi and Yokoo, 2008]. The difficulty is that (1) most algorithms assume that an oracle exists to provide an efficient hierarchy; and (2) the few existing solutions to the Multiagent Hierarchy Generation problem are either centralized or do not deal well (or at all) with dynamic addition and removal of agents from the hierarchy.

One potential application of a solution to DynDisMHG is in MANETs. Protocols such as Fireworks [Law *et al.*, 2007] overlay a communications structure onto a wireless network, which is highly dynamic as the nodes are constantly moving. There is the potential for cross-layer design: if the mobile nodes are executing a multiagent system, a communications structure could be created to exploit knowledge of both network and application layer properties.

There has been interest in DCR algorithms that are able to solve constantly changing problems [Petcu and Faltings, 2007], including those in which agents can dynamically enter and leave the hierarchy [Lass *et al.*, 2008]. All existing provably superstabilizing (*i.e.*, "complete") dynamic DCR algorithms, however, make a similar assumption to their static DCR counterparts: that a separate algorithm exists to generate and maintain the dynamically changing agent hierarchy.

Similar to dynamic DCR, there has been much interest in hierarchies of *holonic* multiagent systems (or *holarchies*), with wide ranging applications in distributed problem solving and manufacturing [Fischer *et al.*, 2004]. Some have even claimed that a prerequisite for innovation in multiagent systems is the capability for subsets of agents to dynamically create *ad hoc* hierarchies, called "*adhocracies*" [van Aart *et al.*, 2004]. Empirical evaluations have concluded that agents in a dynamic hierarchy are able to perform distributed problem solving better than agents in a static hierarchy [Yokoo, 1995]. It is anticipated that solutions to the problem of distributed multiagent hierarchy/holarchy/adhocracy generation will motivate many other applications.

Given a graph of expected interaction between the agents, this paper introduces an algorithm, called Multiagent Organization with Bounded Edit Distance (Mobed), for determining and maintaining an organizational hierarchy that is compatible with the problem being solved. It is shown that Mobed is correct and that it outperforms alternative approaches by as

much as 300% in terms of edit distance between perturbations with little impact to computation and privacy.

## 2 Preliminaries

Let $G = \langle A, E\rangle$ be a graph consisting of an ordered set of agents, $A$, and a set of edges $E$. Each edge $\langle a_i, a_j\rangle \in E$ implies that agent $a_i$ will need to interact with $a_j$. We shall hereafter call this an *interaction graph*. Let $N_i$ denote the *neighborhood* of agent $a_i$: the set of all agents that share an edge with $a_i$. A *multiagent hierarchy* for a given graph $G = \langle A, E\rangle$ is an unordered, labeled, rooted tree denoted by the tuple $T = \langle A, \pi : A \to A\rangle$, where $\pi$ is a function mapping agents to their parent in the tree. The inverse of the parent function, $\pi^{-1}(a_i)$, shall be used to denote the set of children of agent $a_i$. The notation "$R^+$" shall be used to represent the transitive closure of a binary relation $R$. Agent $a_j$ is said to be the *ancestor* of an agent $a_i$ in a hierarchy if $a_j$ has the property $(a_j = \pi(a_i))^+$. Let $C_i$ be the set of ancestors of agent $a_i$. Likewise, agent $a_j$ is a *descendant* of $a_i$ if $(a_j \in \pi^{-1}(a_i))^+$. Let $D_i$ be the set of descendants of agent $a_i$. Let $v : \mathcal{P}(A \times A) \to \mathbb{B}$ be a validity testing function defined as:

$$v(I) = (\forall\langle a_i, a_j\rangle \in I : a_i \in (C_j \cup D_j))\,. \qquad (1)$$

Given an interaction graph $G = \langle A, E\rangle$, a multiagent hierarchy $T$ is said to be *valid* for a given problem if $v(E) = \text{TRUE}$. A valid hierarchy inherently has the property that each pair of neighboring agents in the interaction graph are either ancestors or descendants of each other in the hierarchy. This ensures that no interaction will necessarily occur between agents in disjoint subtrees. Therefore, interactions in disjoint subtrees may occur in parallel.

## 3 Existing Work

It is relatively trivial to prove that a simple depth-first traversal of the interaction graph will produce a valid hierarchy. Distributed algorithms for performing such a DFS traversal are known [Hamadi *et al.*, 1998; Collin and Dolev, 1994]. A general problem with DFS-based approaches, though, is that they will often produce sub-optimal hierarchies (*i.e.*, trees that are unnecessarily deep). For example, the hierarchy in Figure 1(b) might have been generated using a DFS traversal, however, the best-case hierarchy in Figure 1(c) could not have been generated using DFS. One metric for the amount of possible parallelism in a tree is induced-width; in general, the problem of finding a minimum-induced-width spanning tree of a graph is $\mathcal{NP}$-HARD [Arnborg, 1985].

Agent hierarchies, often called "variable orderings," are employed in many DCR algorithms, usually as a means to parallelize computation for portions of the constraint graph. Most provably optimal DCR algorithms require a special hierarchy in the form of *pseudotree*. A decentralized algorithm for creating valid pseudotrees has been proposed [Chechetka and Sycara, 2005], however, its efficiency relies upon *a priori* knowledge about the maximum block size of the interaction graph, and it is also unclear how it might be extended to dynamic hierarchies. Some DCR algorithms construct a DFS-based pseudotree as the problem is being solved [Hamadi *et al.*, 1998; Silaghi and Yokoo, 2008], however, it is likewise unclear how these algorithms might be extended to handle the intricacies of concurrency imposed by dynamic hierarchies. A number of algorithms based on asynchronous backtracking have been developed that dynamically reorder the agents within the hierarchy as the problem is being solved [Yokoo, 1995; Zivan and Meisels, 2006], but this approach has only been explored in terms of static DCR problems and it is unclear how it might be extended to problems in which agents can dynamically be added and removed.

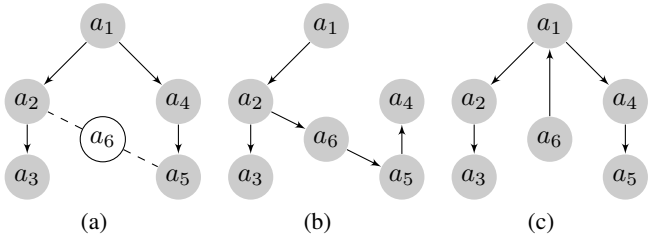



Figure 1: $a_6$, whose interaction graph neighbors are $a_2$ and $a_5$, requests to join the existing hierarchy in (a). If DFS is simply re-run the hierarchy in (b) results. Note that the parents of both $a_4$ and $a_5$ change. The optimal hierarchy in terms of minimal depth and edits—which could not have been produced by a DFS traversal—is in (c).

DFS-based algorithms are relatively inexpensive (most require only a linear number of rounds), so an argument might be made that DFS could simply be re-run every time the tree changes, possibly through the use of a self-stabilizing algorithm. In certain instances, however, such an approach might cause a large disparity between the original hierarchy and the hierarchy resulting after the perturbation, as pictured in Figure 1. Continuing the corporation example, the marketing department should not necessarily have to change its managerial structure every time a new accountant is hired in the billing department. An approach with a minimal number of edits to the existing hierarchy is therefore desirable. One way to ensure a constant number of edits is to simply add new agents as a parent of the root of the hierarchy. The problem with this method, however, is that if many new agents are added then the hierarchy will tend toward a chain, which is the worst case topology in terms of parallelism. What is ultimately desirable, then, is an approach that both minimizes edit distance between successive hierarchies and also minimizes tree depth.

## 4 The Algorithm

We assume that the communications network provides guaranteed delivery of messages, however, there may be arbitrary latency (*i.e.*, an asynchronous network [Lynch, 1997]). We furthermore assume that all agents are honest and correct and thus need not consider the problem of Byzantine failure. The agents are non-adversarial insofar as their primary goal is to optimize the utility of the entire coalition, however, the individual agents may have data that they wish to keep private from others. Agents' perceptions of the interaction graph are consistent, possibly through the use of a distributed consen-

sus algorithm [Lynch, 1997]. Each agent has a unique identifier with a globally agreed ordering. Each agent, however, may not know of all of the other agents in the network; it is only assumed that $a_i$ knows the existence of all $a_j \in N_i$. Each agent that has already been placed in the hierarchy only knows its parents, children, and interaction graph neighbors. Agents also know the relative location of interaction graph neighbors (*i.e.*, ancestor or descendant).

As Mobed applies to DCR, it should be noted that our notion of an interaction graph can be equated to DCR's notion of a constraint graph. In this sense, though, our formalization is then in the context of constraint graphs of *agents* as opposed to constraint graphs of *variables* (the latter of which is the norm for DCR). This presentation was chosen for sake of both brevity and accessibility. Nothing precludes this algorithm from being applied to constraint graphs with multiple variables per agent; in such a case the work herein may be read such that "agents" are instead "variables."

With these assumptions in mind, there are a number of challenges in devising a DynDisMHG algorithm (complicated by the fact that there is no central server and agents act asynchronously in an asynchronous network): To what extent can privacy be maintained? What if a new agent has neighbors in disjoint hierarchies? How are multiple, concurrent requests for addition and removal handled? How is the hierarchy initialized? To what extent can the perturbation of an existing hierarchy be minimized subsequent to agent addition or removal? The remainder section introduces Mobed: an algorithm that addresses all of these challenges.

## 4.1 Theory

Given an existing hierarchy and a new agent, the first problem is to determine where in the hierarchy that agent should be added such that the hierarchy remains valid. We shall now propose and prove a series of lemmas that define such an insertion point. We first define Equation 2 that tests whether or not a given agent already in the hierarchy is a valid insertion point. In Lemmas 1 and 2 we prove, respectively, that such an insertion point must exist and that it must be unique. In Lemmas 3 and 4 we prove that the new agent can be inserted either as the parent or child of the insertion point.

Let $h : A \times \mathcal{P}(A) \to \mathbb{B}$ be a function defined as

$$h(a_i, I) = v(\{a_i\} \times I \setminus \{a_i\}) \wedge \; (D_i \cap I = \emptyset \vee |\{a_j \in \pi^{-1}(a_i) : (D_j \cup \{a_j\}) \cap I \neq \emptyset\}| > 1) \wedge (\forall a_j \in C_i : \neg h(a_j, I)). \tag{2}$$

We shall hereafter refer to an agent $a_i$ that satisfies $h(a_i, I)$ as an *insertion point* for the set $I$. $h(a_i, I)$ will be TRUE if all of the following are true: $a_i$ is an ancestor or descendant of all agents in $I \setminus \{a_i\}$; $a_i$ either has no descendants in $I$ or $a_i$ has more than one child whose subtree has agents in $I$; and none of $a_i$'s ancestors are insertion points for $I$.

**Lemma 1.** *Given a valid hierarchy $T = \langle A, \pi \rangle$ and an agent $a_i \notin A$, then*

$$N_i \subseteq A \implies (\exists a_\ell \in A : h(a_\ell, N_i) = \text{TRUE}). \tag{3}$$

*Proof.* Let us assume, on the contrary, that $N_i \subseteq A$ but there does not exist an agent $a_\ell$ such that $h(a_\ell, N_i)$. This means that either $\forall a_j \in A : N_i \setminus \{a_j\} \nsubseteq D_j \cup C_j$; every agent has exactly one child whose subtree contains an agent in $N_i$; or every agent has at least one ancestor that is an insertion point for $N_i$. The first case contradicts $N_i \subseteq A$. The second case implies that the hierarchy $T$ is cyclic (and therefore invalid) which is a contradiction. The third case either means that an $a_\ell$ must exist or it means that the hierarchy $T$ is cyclic, both of which are contradictions. □

**Lemma 2.** *The insertion point must be unique.*

*Proof.* Let us assume, on the contrary, that there are at least two agents in $A$ that satisfy the existential quantification of $a_\ell$ in Equation 3; let us call two such agents $a_1$ and $a_2$. $h(a_1, N_i)$ and $h(a_2, N_i)$ both must be TRUE, which implies that neither $a_1$ nor $a_2$ is an ancestor of the other, further implying that $a_1$ and $a_2$ must be in disjoint subtrees. Since the hierarchy is valid it must not be cyclic, and there must be some agent $a_3$ that is the deepest common ancestor of $a_1$ and $a_2$. Since $a_1$ and $a_2$ are both insertion points for $N_i$, all of the agents in $N_i \setminus \{a_3\}$ must be in $C_3$, which by Equation 2 means that $h(a_3, N_i)$ must be TRUE, contradicting the fact that both $a_1$ and $a_2$ are insertion points for $N_i$. □

**Lemma 3.** *Given a valid hierarchy $T = \langle A, \pi \rangle$, the addition of a new agent $a_i \notin A$ inserted between $a_\ell \in A$ and $\pi(a_\ell)$ will produce a valid new hierarchy $T' = \langle A \cup \{a_i\}, \pi \cup \{\langle a_\ell, a_i \rangle, \langle \pi(a_i), a_\ell \rangle\} \setminus \{\langle \pi(a_i), a_i \rangle\} \rangle$ if $h(a_\ell, N_i)$.*

*Proof.* Since $a_\ell$ will be the only child of $a_i$, $a_i$ will share all of $a_\ell$'s previous ancestors and descendants. Since $a_\ell$ was already a valid insertion point, $a_i$ must also remain valid. □

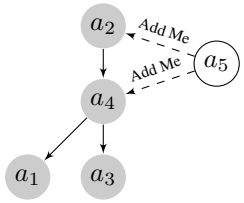
**Lemma 4.** *Given a valid hierarchy $T = \langle A, \pi \rangle$, the addition of new agent $a_i \notin A$ as a child of $a_\ell \in A$ will produce a valid new hierarchy $T' = \langle A \cup \{a_i\}, \pi \cup \{\langle a_i, a_\ell \rangle\} \rangle$ if $N_i \cap D_\ell = \emptyset$ and $a_\ell$ is the insertion point for $N_i$.*

*Proof.* Assume, on the contrary, that the addition of $a_i$ as a child of $a_\ell$ will produce an invalid hierarchy. By Equation 1, this means that there is at least one pair of neighboring agents that are neither ancestors nor descendants of each other. Since the original hierarchy $T$ was valid, we know that such a pair of agents must be in $\{a_i\} \times N_i$. Since $v(\{a_\ell\} \times N_i)$ is true, we know that all of $a_i$'s interaction graph neighbors are either ancestors or descendants of $a_\ell$. Since $a_i$ is added as a child of $a_\ell$ and therefore shares all of $a_\ell$'s ancestors, it must be true that $\exists a_j \in N_i : a_j \in D_\ell$, which contradicts $N_i \cap D_\ell = \emptyset$. □

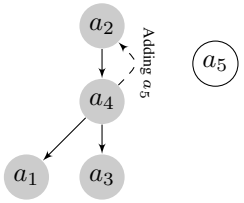
## 4.2 General Principles

Based upon the results of the previous section, Mobed adds an agent $a_i$ to an existing hierarchy by the following procedure:

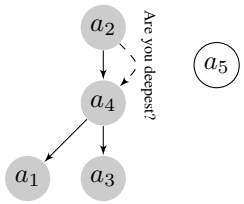
1: find a valid insertion point $a_\ell$ /* one must exist according to Lemmas 1 and 2 */
2: **if** $D_\ell \cap N_i = \emptyset$ **then** /* Lemma 4 */
3: $\quad \pi(a_i) \leftarrow a_\ell$
4: **else** /* Lemma 3 */
5: $\quad \pi(a_i) \leftarrow \pi(a_\ell)$
6: $\quad \pi(a_\ell) \leftarrow a_i$



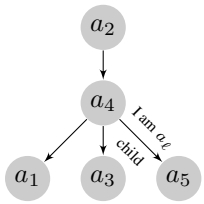

(a) $a_5$ initiates its addition to the existing hierarchy by sending messages to all $a_j \in N_5$.




(b) $a_4$ forwards the message on to its parent, $a_2$.




(c) $a_2$ has received two messages regarding the addition of $a_5$ and $|N_5| = 2$, however, $a_2$ has a single child, so ask that child if it is deepest...




(d) $a_4$ did not receive any messages from its children ($D_4 \cap N_5 = \emptyset$), so add $a_5$ as a child.

Figure 2: An execution of the algorithm for addition of a agent $a_5$ to an existing hierarchy. Solid edges represent edges in the existing hierarchy. Dashed edges represent message passing.

Determining $a_\ell$ (step one in the procedure) can be performed by recursively passing messages up the tree starting from all agents in $N_i$. Without loss of generality, let us assume that all agents in $N_i$ are already present in the same hierarchy; this assures that there must be some agent $a_j$ that will eventually receive $|N_i|$ messages from its children. Then,

1: **while** $a_j$ received exactly one message regarding the addition of $a_i$ **do**
2: $\quad a_j \leftarrow$ the child from which $a_j$ received the message
3: $a_\ell \leftarrow a_j$

Using this method it is trivial to check whether $D_\ell \cap N_i = \emptyset$ (step two in the procedure): if $a_\ell$ did not receive any messages from its children then we know $D_\ell \cap N_i = \emptyset$ is true. Figure 2 provides an example execution of this algorithm.

### 4.3 Merging Hierarchies

We shall now consider the case when a new agent's neighborhood contains agents in disjoint hierarchies. This may happen if $a_i$ is an articulation point of the interaction graph. The approach for this case is simply to add $a_i$ as the new parent of the roots of $T_1$ and $T_2$. The problem, however, is that no agent in $N_i$ necessarily knows that they are in disjoint hierarchies and the addition process as described above will deadlock.

The solution is as follows: whenever a root of a hierarchy (*e.g.*, $a_1$ and $a_3$) receives an addition request regarding a new agent $a_i$—regardless of whether that addition request was sent directly to the root or whether it was propagated up the hierarchy—and that root has received fewer than $|N_i|$ such addition requests, then that root will additionally send a `Root` message to $a_i$ stating that it is the root of a hierarchy. If $a_i$ ever receives $|N_i|$ such messages then $a_i$ will become the new parent of all agents from whom it received the messages.

### 4.4 Preventing Race Conditions

The algorithm as it is presented above will work correctly if there is exactly one agent being added at a time. Due to the possibility of arbitrary message latency, there is a chance that concurrent additions could result in inconsistent modifications to the hierarchy among the agents. To address this we introduce a concept of *active blocks* of the hierarchy. When an agent $a_j$ receives an add request regarding a new agent $a_i$, then $a_j$ goes into *active* mode and proceeds as normal. If $a_j$ is already in active mode, however, it will immediately reply to $a_i$ with an `AlreadyActive` message. Such an error condition implies that another agent in the subtree rooted at $a_j$ (or an agent on the path from $a_j$ to the root) is in the process of either being added or removed; we shall call this agent $a_k$. $a_j$ will also send an `AlreadyActive` message to $a_k$. These messages contain a field stating whether $k > i$. If an agent ever receives such a message it will inform all agents in its neighborhood that it is cancelling its addition. If the agent's identifier is lower priority than the other sent in the `AlreadyActive` message then that agent will perform an exponential backoff before restarting its addition. Otherwise, the agent will only sleep for a constant time period. Once the algorithm is complete, all agents that received an addition message regarding $a_i$ become inactive.

### 4.5 Initial Generation

The special cases addressed in the previous sections are sufficient to generate and maintain hierarchies *as long as* a hierarchy already exists. How should the initial hierarchy be generated? Which agent should become the first root?

The solution is to construct the initial hierarchy semi-synchronously: an agent will not attempt addition of itself to a hierarchy until all of its higher-priority neighbors are already members of a valid hierarchy. A new agent to be added to the hierarchy, $a_i$, will send an `AddMe` to all $a_j \in N_i$, as described above, however, $a_i$ will send them one-by-one in order of decreasing priority of $j$. After each `AddMe` is sent, $a_i$ will block until receipt of a reply from $a_j$ before proceeding to the next, lower-priority neighbor. The neighbor's reply can be one of three possibilities: an `AlreadyActive` message as described in §4.4, a `NoTree` message (meaning the neighbor has not yet been added to a tree), or a `AdditionStarted` message (meaning that the neighbor does have a tree and has started the addition process for $a_i$ as described above). If $a_i$ receives a `NoTree` message from $a_j$ and $j > i$ then $a_i$ will send a `CancelSearch` message to all $a_k \in N_i$ where $k > j$ and $a_i$ will block until it receives an `AddRequest` message from another agent. If, on the other hand, a `NoTree` message is received from an $a_j$ where $j < i$ then it is ignored. The addition will then proceed as in §4.2. Pseudocode for the entire algorithm—implementing the constructs in §4.2–4.5—is given in Algorithm 1.

**Algorithm 1** Mobed's distributed addition of a new agent, regardless of whether or not any of the agent's neighbors are already in hierarchies or whether those hierarchies are disjoint.

Note that the $t$ argument is used as a type of logical clock to avoid processing messages that have expired. If a message is ever received with a $t$ value that is lower than any other message that has been received from the sender then the message is discarded.

**procedure** ADD-AGENT($a_i, t = 0$)
**Require:** $a_i$ is the agent to be added. $t$ is a counter for this addition attempt, initially set to zero.
  $H \leftarrow N_i$
  **for all** $a_j \in N_i$ in order of descending $j$ **do**
    SEND-MESSAGE(`HasTree?`) to $a_j$
    wait for a reply from $a_j$
    **if** the reply is `AlreadyActive` **then**
      handle as described in §4.4.
    **else if** the reply is `NoTree` **then**
      **if** $j > i$ **then**
        send a `CancelSearch`($t$) message to all $a_m \in N_i$ where $a_m > a_j$.
        wait to receive an `AddRequest` message
        **return** ADD-AGENT($a_i, t + 1$)
      **else**
        $H \leftarrow H \setminus \{a_j\}$
    **else if** the reply is a success **then**
      do nothing
  **for all** $a_j \in H$ **do**
    SEND-MESSAGE(`AddMe`, $a_i, |H|, t$) to $a_j$
  $R \leftarrow \emptyset$ /* $R$ is a set and therefore does not allow duplicates */
  **while** $|R| < |H|$ **do**
    **if** the next message is a `Root` message from $a_j$ **then**
      $R \leftarrow R \cup \{a_j\}$
      **if** $|R| = |H|$ **then** /* $|N_i|$ contains agents in disjoint hierarchies */
        **for all** $a_r \in R$ **do**
          $\pi(a_r) \leftarrow a_r$ /* Become the new parent of $a_r$ */
          Tell $a_r$ that we are its new parent and that it can become inactive.
    **else if** the next message is `Added`($p, c$) from $a_j$ **then**
      $\pi(a_i) \leftarrow p$ /* our new parent is $p$ */
      $\pi(c) \leftarrow a_i$ /* our new child is $c$ */
      $R \leftarrow H$
  send an `AddRequest` to all lower-priority agents to whom $a_i$ has ever sent a `NoTree` message.

**procedure** HANDLE-ADDME($a_n, q, t$) sent from $a_j$
**Require:** $a_n$ is the agent requesting to be added and $a_i$ is the agent that received (and is processing) the message. $m : V \to \mathbb{N}$, $r : V \to \mathcal{P}(\pi^{-1}(a_i))$, and $s : V \to \mathbb{B}$ are all maps retained in memory. $m$ maps variables to an integer, $r$ maps variables to the power set of $a_i$'s children, and $s$ maps variables to a boolean. If a key does not exist in $s$ then its value is taken to be FALSE. $a_v$ is the agent for whom $a_i$ is currently active, or $\emptyset$ if $a_i$ is inactive.
  **if** $a_i$ is not yet in the hierarchy **then**
    SEND-MESSAGE(`NoTree`, $t$) to $a_n$
    **return**
  **else if** $a_v \neq \emptyset \neq a_n$ **then**
    SEND-MESSAGE(`AlreadyActive`, $v > n, t$) to $a_n$ and $a_v$
    $a_v \leftarrow \emptyset$
    Clear all of the maps in memory and tell $a_j$ to cancel its search, forwarding the message to all agents in the current active block.
    **return**
  **else**
    $a_v \leftarrow a_n$ /* make $a_i$ active for $a_n$ */
  **if** $a_j = a_n$ **then** /* $a_j$ is the variable requesting to be added */
    $m(a_n) \leftarrow 1$, $r(a_n) \leftarrow \emptyset$, and $s(a_n) \leftarrow$TRUE
  **else if** $a_j \in \pi^{-1}(a_i)$ **then** /* $a_j$ is one of our children */
    $m(a_n) \leftarrow m(a_n) + 1$
    $r(a_n) \leftarrow r(a_n) \cup \{a_j\}$
  **if** $m(a_n) = q$ **then** /* $a_i$ satisfies Equation 1 for $N_n$ */
    **if** $|r(a_n)| = 1$ **then** /* $a_i$ is not the deepest */
      $q' \leftarrow q$
      **if** $s(a_n) \mapsto$TRUE **then** /* $a_i$ was originally sent a message from $a_n$ (meaning $a_i \in N_n$) */
        $q' \leftarrow q' - 1$
      $a_k \leftarrow$ the single variable in $r(a_n)$
      SEND-MESSAGE(`AddMe`, $a_n, q', t$) to $a_k$
      $a_v \leftarrow \emptyset$
      remove $m(a_n)$, $r(a_n)$, and $s(a_n)$ from memory
    **else** /* $a_i$ is the deepest vertex satisfying Equation 1 */
      **if** $r(a_n) \neq \emptyset$ **then** /* we have at least one descendant that is in $N_n$ (Lemma 3) */
        $\pi(a_n) \leftarrow \pi(a_i)$
        $\pi(a_i) \leftarrow a_n$
        SEND-MESSAGE(`Added`, $\pi(a_n), a_i$) to $a_n$
      **else** /* Lemma 4 */
        $\pi(a_n) \leftarrow a_i$
        SEND-MESSAGE(`Added`, $a_i, \emptyset$) to $a_n$
      $a_v \leftarrow \emptyset$
      Tell all of the agents in $r(a_n)$ that the search is over and clear all of the maps in memory.
  **else if** our vertex $a_i$ is not the root of the pseudotree **then**
    SEND-MESSAGE(`AddMe`, $a_n, q, t$) to $\pi(a_i)$ /* Forward the message to our parent */
  **else** /* This situation *may* occur if there are two or more variables in $N_n$ from disjoint hierarchies */
    SEND-MESSAGE(`Root`, $a_i, t$) to $a_n$

### 4.6 Changes to Constraints and Agent Removal

Changes to constraints can be handled by removing and re-adding all affected agents. Removal of an agent $a_i$ can be accomplished by making $a_i$, $\pi(a_i)$, and all $a_j \in \pi^{-1}(a_i)$ active. All of the children are then made the children of $\pi(a_i)$ and $a_i$ is removed. The hierarchy will remain valid.

## 5 Analysis

This section analyzes—both theoretically and empirically—the performance of Mobed. For the empirical analysis, a series of random, connected graphs were randomly generated from a uniform distribution with varying numbers of vertices and edge densities[1]. 100 such random graphs were generated for each pair of number of vertices and edge density. For each graph both DFS and Mobed were run to generate a valid hierarchies. A new vertex was then randomly added in such a way as to maintain the given edge density. DFS and Mobed were then re-run to produce a new valid hierarchy. Various metrics including the average number of rounds of computation (*i.e.*, the length of the longest causal chain of synchronous messages required for the algorithm to reach quiescence) and the edit distance between hierarchy perturbations were recorded.

### 5.1 Computational Complexity

Let us consider the computational complexity of adding a new agent $a_i$ to an existing valid hierarchy $T = \langle A_T, \pi \rangle$. We assume that $N_i \subseteq A_T$ and there are no other agents attempting to be concurrently added. $a_i$ will first send one message to each neighbor in $N_i$. Each $a_j \in N_i$ will then forward the message up the tree. In the worst case in terms of synchrony, each of the $|N_i|$ messages will have to traverse up the tree to the root, followed by the root propagating a single message back down to the insertion point. The addition of a single agent therefore requires worst case $O(|N_i|d_T)$ rounds, where $d_T$ is the depth of the deepest agent in $T$. In the worst case in terms of existing hierarchy topology, $T$ will be a chain and the worst case number of rounds for a single insertion is $O(|N_i||A_T|)$. Construction of a hierarchy from scratch for a worst-case fully-connected interaction graph therefore requires $O\left(\sum_{i=1}^{|A|} i \cdot |A|\right) = O\left(|A|^3\right)$ rounds. The best-case runtime, however, is $O(|N_i|)$, which in the real world will often be quite small. Therefore, the runtime in terms of number of rounds will always be polynomial and—if $N_i$ is bounded—individual additions will run in amortized linear time. This theory is supported by the empirical results (Figure 3).

### 5.2 Edit Distance

The *edit distance* between two hierarchies, $T_1 = \langle A_1, \pi_1 \rangle$ and $T_2 = \langle A_2, \pi_2 \rangle$, is defined as[2]

$$|\{\langle a_i, \pi_1(a_i)\rangle : a_i \in A_1\} \Delta \{\langle a_j, \pi_2(a_j)\rangle : a_j \in A_2\}|.$$

[1]Edge density, ranging in $[0, 1]$, is the percentage of possible edges that exist in the graph. A density of $0.0$ means that the graph has no edges while a density of $1.0$ means that the graph is complete.

[2]*i.e.*, the minimum number of child-parent relationships that must be reassigned in order for the two trees to become isomorphic.

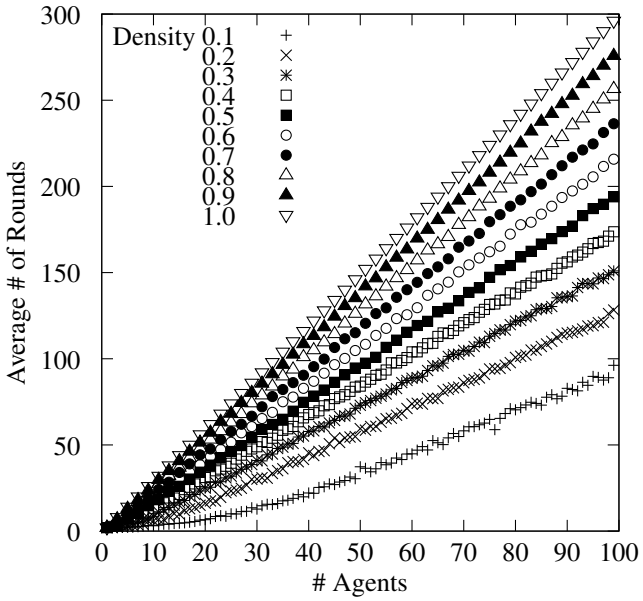



Figure 3: Average number of rounds for Mobed to reach quiescence for a single agent addition.

Ideally, after a single addition of an agent the edit distance between the original hierarchy and the resulting hierarchy will be minimized. The worst case edit distance for DFS will occur whenever the existing hierarchy $T = \langle A_T, \pi \rangle$ consists of a root with $|A_T| - 1$ children and the agent being added, $a_i \notin A_T$, has the property $N_i = A_T$. In this case $|A|$ edits may occur. In contrast, Mobed bounds the number of edits for each agent addition at two.

During our empirical analysis, edit distance according to the metric formalized above was noted for both DFS and Mobed between the initial and post-vertex-addition hierarchies. Figure 4 gives the percentage difference between the edit distance of DFS and Mobed; positive percentages mean that DFS had a worse edit distance. DFS performed worse for sparse graphs (density $\leq 0.5$). Although DFS performed better on dense graphs, it was only ever one edit better.

Our definition of edit distance is quite favorable to DFS; in some domains a better metric may be the number of ancestor-descendant relationships that are modified as a result of each change to the interaction graph. Such perturbations in the context of DCR might cause large portions of the previously explored search space to be expanded again. With this stricter metric, Mobed still has a bounded edit distance of two, while DFS may perform much worse.

## 6 Conclusions and Future Work

This paper has two primary contributions: a formal generalization of the DynDisMHG problem and introduction of Mobed: an algorithm for distributedly constructing and maintaining multiagent hierarchies with bounded edit distance between hierarchy perturbations. Mobed was compared—both theoretically and empirically—to the only other viable solution to the DynDisMHG problem: distributed DFS. It was

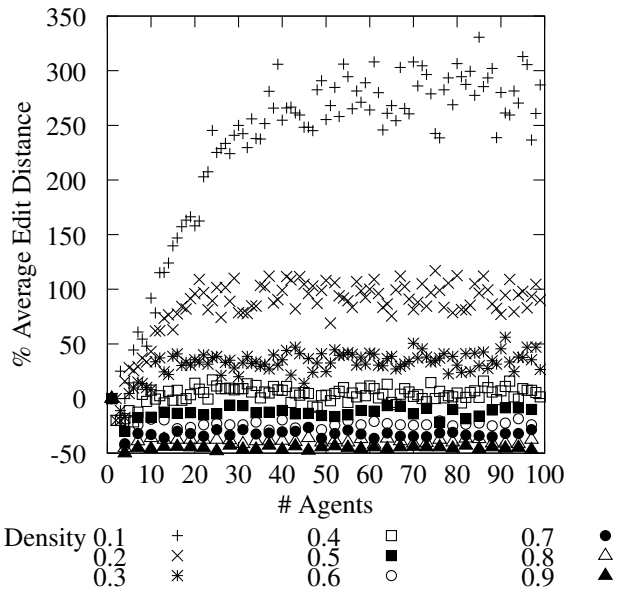



Figure 4: Comparison of the edit distance of DFS to Mobed. The Y-axis is the percentage difference between DFS and Mobed; positive values imply DFS performed worse.

shown that Mobed will always reach quiescence in a linear number of rounds for each agent addition with respect to the total number of agents, but under certain circumstances it can theoretically run in constant time. Re-running DFS after such a perturbation would also require a linear number of rounds, but may have arbitrarily bad edit distance. The edit distance of Mobed is always bounded at two edits, which is very low. For sparse graphs (density less than $0.5$) Mobed has at least as good an edit distance as DFS, and exhibited as much as a 300% benefit. For those instances when Mobed exhibited a higher edit distance than DFS (*i.e.*, when the graph is dense) its edit distance was no more than one edit worse. Privacy is also maintained in Mobed insofar as agents only ever have knowledge of their interaction graph neighbors, hierarchy parents, and hierarchy children. Therefore Mobed is a viable replacement for DFS as a solution to the DynDisMHG problem, especially for sparse interaction graphs.

In the future we will empirically analyze the use of Mobed in distributed problem solving algorithms. There is also much work to be done in studying hierarchy generation techniques that better balance the tradeoff between computational efficiency/messaging, edit distance, and privacy, and also methods to maintain other invariants on the hierarchy's topology.

## References

[Arnborg, 1985] Stefan Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability—a survey. *BIT Numerical Mathematics*, 25(1):1–23, March 1985.

[Chechetka and Sycara, 2005] Anton Chechetka and Katia Sycara. A decentralized variable ordering method for distributed constraint optimization. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, July 2005.

[Collin and Dolev, 1994] Zeev Collin and Shlomi Dolev. Self-stabilizing depth-first search. *Information Processing Letters*, 49(6):297–301, 1994.

[Fischer *et al.*, 2004] Klaus Fischer, Michael Schillo, and Jörg Siekmann. *Holonic and Multi-Agent Systems for Manufacturing*, volume 2744 of *Lecture Notes in Computer Science*, chapter Holonic Multiagent Systems: A Foundation for the Organisation of Multiagent Systems, pages 1083–1084. Springer, 2004.

[Hamadi *et al.*, 1998] Youssef Hamadi, Christian Bessière, and Joël Quinqueton. Backtracking in distributed constraint networks. In *Proceedings of the European Conference on Artificial Intelligence*, pages 219–223, 1998.

[Lass *et al.*, 2008] Robert N. Lass, Evan A. Sultanik, and William C. Regli. Dynamic distributed constraint reasoning. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, pages 1886–1887, 2008.

[Law *et al.*, 2007] Lap Kong Law, Srikanth V. Krishnamurthy, and Michalis Faloutsos. Understanding and exploiting the trade-offs between broadcasting and multicasting in mobile ad hoc networks. *IEEE Transactions on Mobile Computing*, 6(3):264–279, 2007.

[Lynch, 1997] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1997.

[Petcu and Faltings, 2007] Adrian Petcu and Boi Faltings. R-DPOP: Optimal solution stability in continuous-time optimization. In *Proceedings of the International Conference on Intelligent Agent Technology*, November 2007.

[Shehory and Kraus, 1998] Onn Shehory and Sarit Kraus. Methods for task allocation via agent coalition formation. *Artificial Intelligence*, 101(1-2):165–200, 1998.

[Silaghi and Yokoo, 2008] Marius Calin Silaghi and Makoto Yokoo. *Encyclopedia of Artificial Intelligence*, chapter on Distributed Constraint Reasoning, pages 507–513. Information Science Reference, 2008.

[Sycara *et al.*, 1996] Katia Sycara, Keith Decker, Anandeep Pannu, Mike Williamson, and Dajun Zeng. Distributed intelligent agents. *IEEE Expert: Intelligent Systems and Their Applications*, 11(6):36–46, 1996.

[van Aart *et al.*, 2004] Chris J. van Aart, Bob Wielinga, and Guus Schreiber. Organizational building blocks for design of distributed intelligent system. *International Journal of Human-Computer Studies*, 61(5):567–599, 2004.

[Yokoo, 1995] Makoto Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming*, pages 407–422, 1995.

[Zivan and Meisels, 2006] Roie Zivan and Amnon Meisels. Dynamic ordering for asynchronous backtracking on DisCSPs. *Constraints*, 11:179–197, 2006.