Learning Scalable Coalition Formation in an Organizational Context*

Sherief Abdallah and Victor Lesser

University of Massachusetts Amherst shario,lesser@cs.umass.edu

1 Introduction

Agents can benefit by cooperating to solve a common problem [2, 11]. For example, several robots may cooperate to move a heavy object, sweep a specific area in short time, etc. However, as the number of agents increases, having all agents involved in a detailed coordination/negotiation process will limit the scalability of the system. It is better to first *form a coalition* of agents that has enough resources to undertake the common problem. Then only the agents in this coalition coordinate and negotiate among themselves.

This situation is common in domains where a task requires more than one agent and there are more than one task competing for resources. Computational grids and distributed sensor networks are examples of such domains. In computational grids a large number of computing systems are connected via a high-speed network. The goal of the grid is to meet the demands of new applications (tasks) that require large amounts of resources and reasonable responsiveness. Such requirements cannot be met by an individual computing system. Only subset of the available computing systems (aka a coalition) has enough resources to accomplish an incoming task.

The work in [8] defined the coalition formation problem as follows (a formal definition is given in Section 2). The input is a set of agents, each controlling some amount of resources, and a set of tasks, each requiring some amount

* This material is based upon work supported in part by the National Science Foundation under Grant No. IIS-9988784 and the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory Air Force Materiel Command, USAF, under agreement F30602-99-2-0525. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, the Defense Advanced Research Projects Agency (DARPA), Air Force Research Laboratory or the U.S. Government.

of resources and each worth some utility. The solution assigns a coalition of agents to each task, such that each task's requirements are satisfied and total utility is maximized. It should be noted that the coalition formation problem is not concerned with how agents in a coalition cooperate to actually executes its assigned task. Such cooperation can be achieved by other complementing frameworks such as teamwork [11].

In this chapter we propose a novel approach for solving the coalition formation problem approximately using an *underlying organization* to guide the formation process. The intuition here is to exploit whatever knowledge is known *a priori* in order to make the coalition formation process more efficient. For instance, in many domains, agents' capabilities remain the same throughout the lifetime of the system. Additionally, incoming tasks may follow some statistical pattern. Can we *organize* agents to exploit this knowledge (of their capabilities and task arrival patterns) to make the search for *future* coalitions more efficient? If so, will all organizations yield the same performance, or do some organizations perform better than others? In the remainder of this chapter we try to provide answers to these questions. The main contributions of this work are:

- an organization-based distributed algorithm for approximately solving the coalition formation problem
- the use of reinforcement learning to optimize the local allocation decisions made by agents in the underlying organization

The chapter is organized as follows. In Section 2 we define the problem formally, laying out the framework we will use throughout the chapter. In Section 4 we present our approach. Section 5 describes our experimental results. We compare our approach to similar work in Section 6. Conclusions and future work are discussed in Section 7.

2 Problem definition

To focus on the coalition formation problem, some simplifying assumptions are made to avoid adding the scheduling problem to it.[2] We assume time is divided into episodes. At the beginning of each episode each agent receives a sequence of tasks.[3] Once a task is allocated a coalition, agents in that coalition can not be assigned to another task until the end of the episode. At the end of every episode all agents are freed and ready to be allocated to the next sequence of tasks. More formally:

Let $T = \langle T_1, T_2, ..., T_q \rangle$ be the sequence of tasks arriving in an episode. Each task T_i is defined by the tuple $\langle u_i, rr_{i,1}, rr_{i,2}, ..., rr_{i,m} \rangle$, where u_i is the

[2] In future we plan to integrate scheduling in our framework.

[3] Note that the overall system may receive more than one task at the same time but at different agents.

utility gained if task T_i is accomplished; and $rr_{i,k}$ is the amount of resource k required by task T_i. Let $I = \{I_1, I_2, ..., I_n\}$ be the set of individual agents in the system. Each agent I_i is defined by the tuple $\langle cr_{i,1}, cr_{i,2}, ..., cr_{i,m}\rangle$, where $cr_{i,k}$ is the amount of resource k controlled by agent I_i.

The coalition formation problem is finding a subset of tasks $S \subseteq T$ that maximizes utility while satisfying the coalition constraints, i.e.:

- $\sum_{i|T_i \in S} u_i$ is maximized
- and there exists a set of coalitions $\bar{C} = \{C_1, ..., C_{|S|}\}$, where $C_i \subseteq I$ is the coalition assigned to task T_i, such that $\forall T_i \in S, \forall k : \sum_{I_j \in C_i} cr_{j,k} \geq rr_{i,k}$, and $\forall i \neq j : C_i \cap C_j = \emptyset$

In other words, each task is assigned a coalition capable of accomplishing it and any agent can join at most one coalition. This means if the resources controlled (collectively) by a coalition exceed the amount of resources required by the assigned task, the excess resources are wasted. Having more than one type of resource means that there will be trade-offs, where decreasing the excess of one resource type may increase the excess of another resource type. Next section shows that the coalition formation problem (as defined above) is NP-hard.

2.1 Complexity

In this section we prove that the Coalition Formation Problem (CFP), as we formulated it, is NP-hard. We do so by reducing the multidimensional knapsack problem, which is known to be NP-hard, to CFP.

The Multi-dimensional Knapsack Problem, MDKP

The input of this problem consists of a set of constraints $C = \{c_1, c_2, ..., c_m\}$ and a set of objects $O = \{o_1, o_2, ..., o_q\}$, where each object is defined by the tuple $o_i =< u_i, w_{i,1}, w_{i,2}, ..., w_{i,m} >$, where u_i is its value and $w_{i,j}$ is its weight for dimension j. The goal is to find a subset of objects $S \subset O$, s.t. $\sum_{o_i \in S} u_i$ is maximized, while $\forall c_j \in C, \sum_{o_i \in S} w_{i,j} \leq c_j$

Theorem 1. *Coalition Formation Problem, CFP, is NP-hard*

Proof. This is proved by reducing an MDKP instance to a CFP instance. This is done as follows. The decision version of the MDKP problem is:

Q1: given a set of objects O and a set of constraints C, is there a valid subset of objects S_k that satisfy the constraints and has total utility of k or more?

The mapping from MDKP to CFP is as follows. For each object $o_i =< u_i, w_{i,1}, ..., w_{i,m} >$ in MDKP, we define an agent $a_i =< w_{i,1}, ..., w_{i,m} >$ and a task $T_i =< u_i, w_{i,1}, ..., w_{i,m} >$. We also add task $T' =< U, W_1, ..., W_m >$, where $U = \sum_{o_i \in O} u_i$ and $W_j = (\sum_{o_i \in O} w_{i,j}) - c_j$ (this amount can be viewed

as the gap between the demand of a resource and its supply). As will be described shortly, T' encodes the constraints of the MDKP instance such that the coalition assigned to this task corresponds to the set of objects left **outside** the knapsack. The CFP decision problem then becomes:

Q2: given the set of tasks T and the set of agents A, is there a a valid set of Coalitions $\bar{C}$ that results in $U + k$ utility or more?

To prove the theorem, we need to show that the answer to Q1 is yes iff the answer to Q2 is yes. Let $\bar{C}_k = \{\{a_i\} : o_i \in S_k\}$ be the set of coalitions corresponding to S_k (i.e. $\bar{C}_k$ is a set of singular coalitions). Let $C_{-k} = \{a_i : o_i \notin S_k\}$, i.e. the coalition corresponding to all objects not in S_k. By definition, every coalition $\{a_i\} \in \bar{C}_k$ can be assigned to T_i, resulting in k utility. The hard part is to prove that the constraints of the MDKP problem is not violated by this assignment. This is where T' comes into play. If S_k satisfies the MDKP constraints, then

$$\forall j, \sum_{i:o_i \in S_k} w_{i,j} \leq c_j$$

$$\therefore \forall j \sum_{i:o_i \in O} w_{i,j} - \sum_{o_i \in S_k} w_{i,j} \geq \sum_{i:o_i \in O} w_{i,j} - c_j$$

$$\therefore \forall j \sum_{i:o_i \notin S_k} w_{i,j} \geq W_j$$

$$\therefore \forall j \sum_{i:a_i \in C_{-k}} w_{i,j} \geq W_j$$

i.e., C_{-k} is a valid coalition to undertake T'. This means there exists a set of coalitions $\bar{C} = \bar{C}_k \cup \{C_{-k}\}$ that yield $k + U$ utility.

3 Control

In a real multiagent system, which implements the coalition formation approach, a task may arrive at any agent. How can this agent *know* which agents have the right capabilities? We refer to the problem of *locating and assigning* an agent to a coalition as the control problem. While the control problem is crucial to the coalition formation process, it has received little attention in previous work that deals with coalition formation. This section tries to pin down the different approaches to solve this problem.

Figure 1 illustrates three possible approaches to the control problem. The first approach is having a fully distributed control paradigm where every agent is a manager. Each manager knows about and controls every other agent. The other extreme is the fully centralized approach, where there is only one manager in the system. The third approach is having a hierarchy. In this case there is a *tree* of managers. Each manager controls a *fixed* number of neighbors. The remainder of this section discusses the trade-offs between these

three approaches in light of the following issues: state consistency, scalability, and reliability.

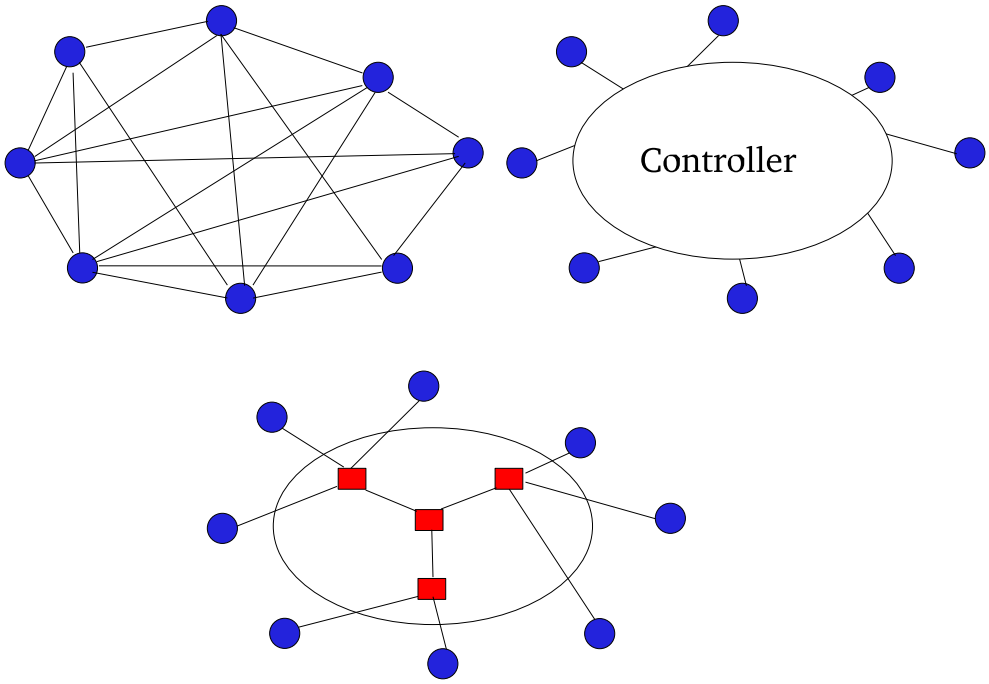

Fig. 1. Different control structures: fully distributed (top left), fully centralized (top right), and hierarchy (bottom).

Communication is needed when a new task arrives for two reasons. First to inform agents that are chosen in the coalition which task they are assigned to. Second, to inform other managers of the change of state in the system, i.e. maintaining *state consistency*. State consistency is the property that every manager in a system sees the same system's state. This is important to avoid conflicts among managers as early as possible. For example, assume manager m_1 asks agent a_1 to undertake task T_1. a_1 accepts and hence is no longer available to be assigned to another task. On the other hand, manager m_2 does not know of the change in a_1's state. m_2 receives another task, decomposes it, and starts contracting subtasks, relying on its incorrect system state. After contracting and committing some subtasks, m_2 asks a_1 to do subtask T_2. a_1 rejects the request as it is still working on T_1. m_2 fails to find a substitute for a_1 and starts decommitting the subtasks already committed.

Naturally, maintaining state consistency becomes a problem as the number of managers in a system increases. For example, in the fully distributed approach where every agent is itself a manager, the number of managers is maximum and maintaining state consistency needs a lot of communication overhead.[4] This makes the fully distributed approach the least efficient. On the other hand, in the fully centralized approach there is only one manager in the system that handles all allocations, therefore maintaining state consistency is free. The hierarchical approach strikes a balance between the other two approaches (the overhead for maintaining state consistency depends on the number of managers in the system).

Scalability is also an issue. A manager that needs to know the state of 1000 agents and control them is much more overloaded than a manager that needs to know the state of only 10 agents and control just these 10 agents. In both the centralized and the fully distributed approaches a manager is connected to all agents in the system, which significantly reduces these approaches scalability. The hierarchical approach that we adopt in this paper is more scalable than the other two approaches.

Reliability is how the failure of a manager affects the performance of the system as a whole. The fully distributed approach is the most reliable, where the failure of a manager minimally affect the system's performance. The centralized approach is naturally the least reliable, but it is possible to employ failure recovery mechanisms, e.g. electing a new manager to replace the one that failed. The hierarchical approach is still not as reliable as the fully distributed approach, but having multiple managers means the system will still be functional even if one manager fails. This may lead to having multiple *disconnected islands* of agents, but each island can still function independently.[5] Again failure recovery mechanisms can be employed in the hierarchical approach as well.

Another control architecture that is not mentioned above is the *network* architecture. This is a generalization of the hierarchy approach, where cycles may exist between managers. Having cycles in the control architecture introduces some problems. Figure 2 shows an example of that. Manager A asks managers B and C to report how many resources they have available. Manager B reports the resources of its neighbors, which include the resources available at C. Similarly, C reports the resources available at B. In the end, manager A will have a *wrong* view of what resources are available, because resources of both B and C are counted twice. Even worse, both B and C also have wrong view of the resources available, as they both may ask A for its state

[4] It is also possible to leave other managers have an *old* state of the system, hoping that no conflict will occur (e.g. they will never ask for the same agent, or even if they ask, the other agent will be already done from the old task.) This may lead to communication savings in some domains. We do not cover this approach in this chapter.

[5] Because islands are now disconnected, it is possible that some tasks that were achievable by the connected hierarchy are no longer achievable.

(which wrongfully indicates that A has a lot of resources). Having cycles also requires care with contracting tasks. Without careful protocol design, a task may circulate indefinitely being continuously contracted. For these reasons we did not consider control structures that include cycles.

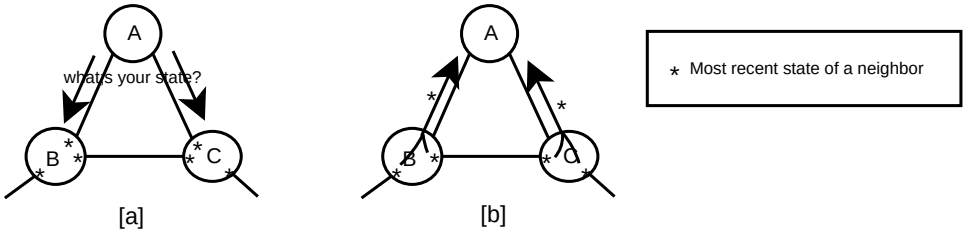

Fig. 2. Problems with cycles in control architectures.

4 Proposed Solution

Because the coalition formation problem is NP-hard, an optimal algorithm will need exponential time in the worst case (unless NP = P). An approximation algorithm, which can exploit information about the problem, is needed. If the environment (in terms of incoming task classes and patterns) does not follow any statistical model, and agents continually and rapidly enter and exit the system, there is little information to be exploited. Luckily, in many real applications the environment does follow a model, and the system can be assumed closed.

In such cases, it is intuitive to take advantage of this stability and *organize* the agents in order to guide the search for future coalitions. We chose to organize agents in a hierarchy, which is both distributed and scalable as discussed in Section 3. Figure 3 shows a sample hierarchical organization. An *individual* (the leaves in Figure 3) represents the resources controlled by a single agent. A *manager* (shown as a circle in Figure 3) is a computational role, which can be executed on any individual agent, or on dedicated computing systems. A manager *represents* agents beneath it when it comes to interaction with other parts of the organization.

Each manager M has a set of children, $children(M)$, which is the set of nodes directly linked below it. So for instance, in the organization shown in Figure 3, $children(M6) = \{I12, I13\}$, while $children(M3) = \{M4, M5, M6\}$. Conversely, each child C has a set of managers $managers(C)$. For example, $managers(M4) = \{M3\}$. For completeness, children of an individual are the empty set, and so are the managers of a root node.

Each agent A (either a manager or an individual) controls, either directly or indirectly, a set of individuals, $cluster(A)$ (i.e., the leaves reachable from

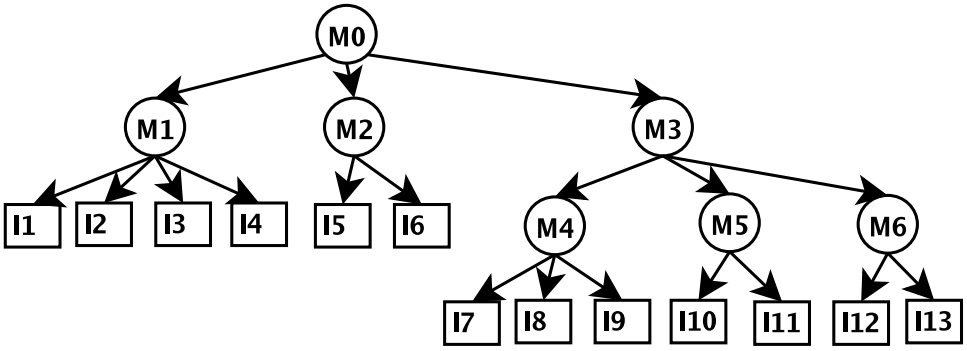

Fig. 3. An Organization Hierarchy

agent A). In the example above, $cluster(M6) = \{I12, I13\}$, $cluster(M3) = \{I7, I8, I9, I10, I11, I12, I13\}$, and $cluster(I6) = \{I6\}$. Also for each agent A, we define $members(A)$ to be the set of all agents reachable from A. In the above example, $members(M3) = \{M3, M4, M5, M6, I7, I8, I9, I10, I11, I12, I13\}$. Sections 4.3 and 4.6 show how agents in such organizations learn to work with each other.

4.1 Example

Figure 4 shows how a group of agents, organized in a hierarchy, can cooperate to form a coalition. A task $T = \langle u = 100, rr_1 = 50, rr_2 = 150\rangle$ is discovered by agent $M6$. Knowing that $members(M6)$ does not have enough resources to accomplish T, $M6$ sends task T to its manager $M3$. Since $members(M3)$ has enough resources to achieve T, $M3$ uses its local policy to chose the best child to contribute in achieving T, which is $M5$. $M3$ partially decomposes T into subtask $T_5 = \langle u_5 = 50, rr_{5,1} = 0, rr_{5,2} = 100\rangle$, and asks $M5$ to allocate a coalition for it. $M5$ returns a committed coalition $C_{T_5} = \{I10, I11\}$. The process continues until the whole task T is allocated. Finally, $M3$ integrates all sub-coalitions into C_T and sends it back to $M6$.

4.2 Architecture

In the system we developed, managers are concurrently and distributively learning their local policies. A local policy determines the order by which a manager decomposes a high-level task into subtasks and allocate these subtasks to its children. The combination of local policies constitutes a global hierarchical policy of the whole system. Figure 5 illustrates a block diagram of a manager's architecture in the system. There is a handler for every child. Each handler includes a neural net, which approximates the value of choosing the corresponding child to form a sub-coalition (for the task at hand). The

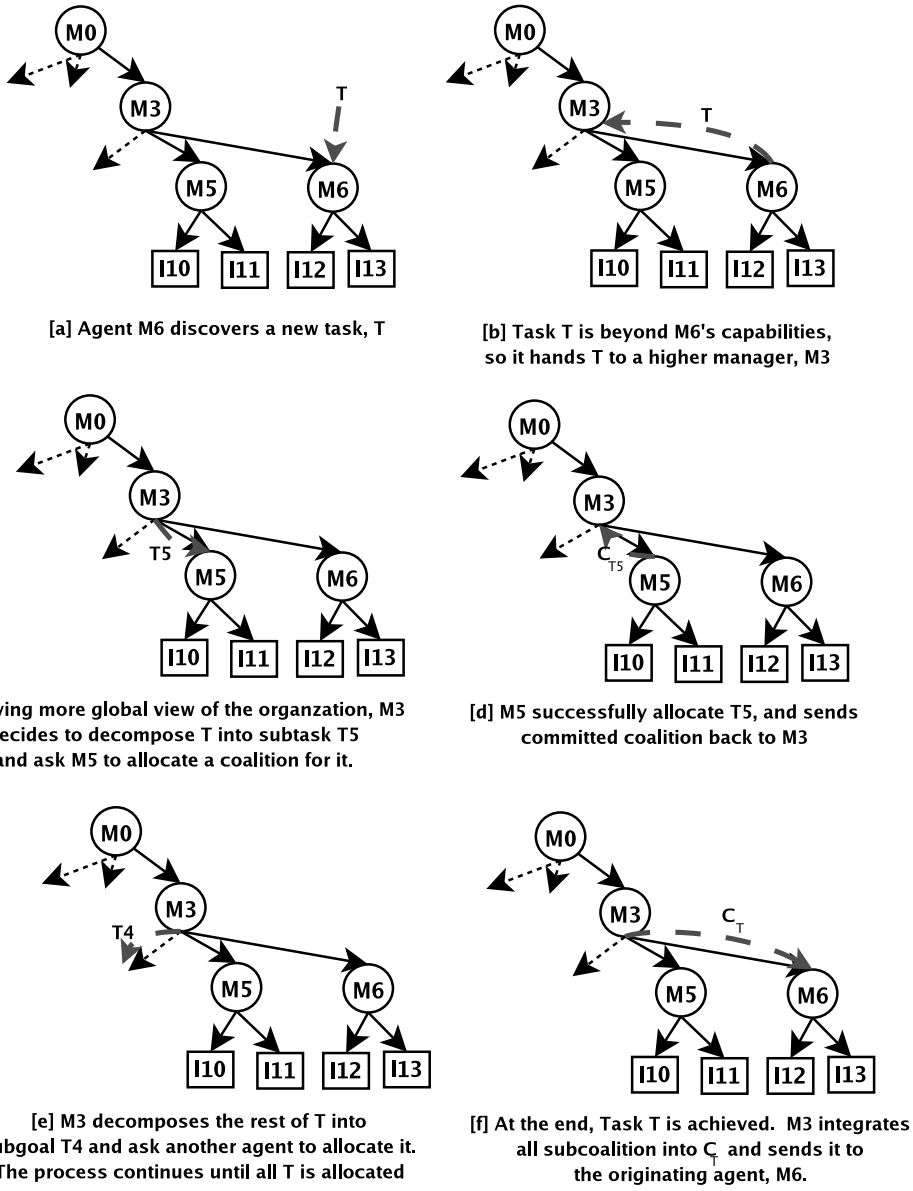

Fig. 4. An example of organization-based coalition formation.

weights of a neural net are optimized using reinforcement learning, as Algorithm 1 shows (described in Section 4.3). To speedup learning using neural nets, the state encoder encodes the current state differently for the neural net

of different children, depending on the amount of resources available at each child. More on learning in Section 4.6.

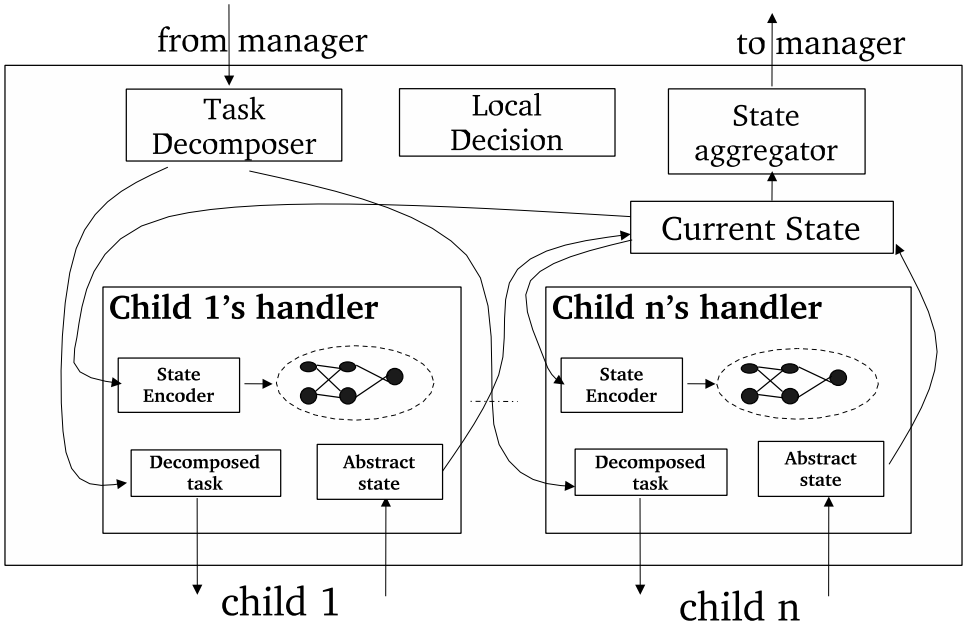

Fig. 5. A manager architecture.

The state aggregator aggregates the state of a manager m before it is sent to higher managers $managers(m)$. When a higher manager m_h receives the aggregated state from its child m, m_h will store the aggregated state in the *abstract state* field of child m's handler. The current state of a manager is a combination of the abstract states of its children and the current status of the task at hand (i.e. the resources the task requires and not yet allocated and the utility to be gained if the remainder of the task is completed). The task decomposer stores arriving tasks. When the local policy chooses a child to form a sub-coalition, the task decomposer decomposes the task for this child (storing it in the child's handler). More details of the operation of a manager in Section 4.3.

4.3 Local Decision

Algorithm 1 describes the decision process used by manager A in the organization once it receives a task T_A. Figure 6 illustrates the algorithm. Though in this figure T_A comes from another agent, T_A can also arrive directly from the environment as well. The algorithm works as follows.

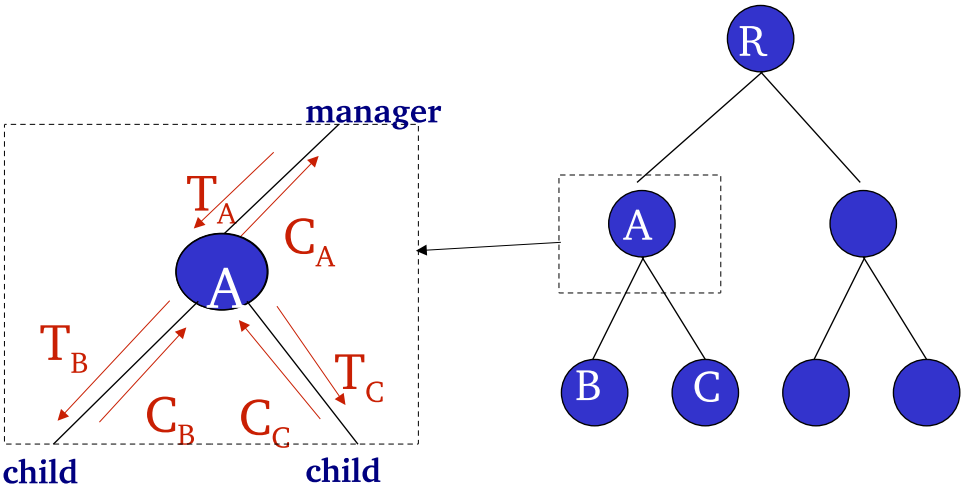

Fig. 6. The recursive decision process of a manager.

LOC_M is the list of coalitions committed by manager M for tasks that M received previously in the current episode. LOC_M is reset at the beginning of each episode. M evaluates its current state s_M (Section 4.4). M then selects an action a based on its policy (Section 4.6). Each action corresponds to a child $M_i \in children(M)$. Once a child is selected, a subtask T_i of T is dynamically created based on M_i's state (Section 4.5). M then asks M_i to form a sub-coalition capable of accomplishing T_i. (The notion $M_i.allocateCoalition(T_i)$ means that the function $allocateCoalition$ is called remotely on agent M_i). M_i forms a sub-coalition C_{T_i} and sends a commitment back to M. M updates C_T and learns about this action. M updates its state, including the amount of resources to be allocated (UR_M) and the corresponding utility to be gained (uu_M).

M selects the next best child and the process continues as long as the following conditions hold (step 3): T requires more resources than currently allocated AND M still controls some unallocated resources that are required by T. At the end, if enough resources are allocated to T, M adds the formed coalition C_T to its list of commitments LOC_M and returns C_T. Otherwise T is passed up the hierarchy. Also to simplify handling of multiple tasks, we do not allow coalition formation of a task to be interrupted. This means that if a new task T_{new} arrives at manager M while M is still forming a coalition for an older task T_{old}, then M will finish forming the coalition for T_{old} before considering T_{new}.

4.4 State Abstraction

For a manager M, the function $encodeState$ encodes the current state at manager M to produce the current state of $members(M)$. This encoding is

Algorithm 1 allocateCoalition(T)

INPUT: task $T = \langle u, rr_1, ..., rr_m \rangle$
OUTPUT: coalition $C_T = \{I_1, ..., I_{|C_T|}\}$

1: let $C_T = \{\}$, $uu \leftarrow u$, $UR \leftarrow \langle rr_1, ..., rr_m \rangle$, $stop \leftarrow$ false, $AR \leftarrow$ the amount of available resources controlled by $M = availableResources() = totalResources(M) - \sum_{C \in LOC} totalResources(C)$
2: $s \leftarrow$ encodeState(uu, UR)
3: **while** $UR > \overline{0}$ AND $UR.AR > 0$ AND $stop =$ false **do**
4: $a \leftarrow$ selectAction(s)
5: let M_i be the child corresponding to a.
6: $T_i \leftarrow$ decomposeTask($\langle UR, uu \rangle$, M_i)
7: $C_{T_i} \leftarrow M_i$.allocateCoalition(T_i)
8: $C_T \leftarrow C_T \cup C_{T_i}$
9: $UR \leftarrow UR - totalResources(C_{T_i})$, $uu \leftarrow uu - u_{T_i}$, and $AR \leftarrow AR - totalResources(C_{T_i})$
10: $r \leftarrow$ time and communication costs of forming C_{T_i}
11: **if** $UR = \overline{0}$ /* T does not need more resources */ **then**
12: $r \leftarrow r + u$
13: **end if**
14: $s' \leftarrow$ encodeState(uu, UR) /* the next state */
15: learn(s, a, r, s')
16: $s \leftarrow s'$
17: **end while**
18: **if** $UR > \overline{0}$ /* task T successfully allocated */ **then**
19: $LOC \leftarrow LOC \cup C_T$ /* to exclude agents in C_T from next allocations */
20: return C_T
21: **else**
22: **if** $\exists M' \in managers(M)$ /* if not root */ **then**
23: $M'.allocateCoalition(T)$ /* pass T up */
24: **else**
25: fail.
26: **end if**
27: **end if**

then fed to neural nets to get action values, as discussed in Section 4.6. Since the higher the manager in the hierarchy the exponentially more individuals it controls, state abstraction is necessary to achieve scalability. Otherwise, one is effectively centralizing the problem. In this work, each manager M abstracts the state of its organization, through the state aggregator (Section 4.2). This abstraction involves aggregating the states of underlying children recursively as described below.

Due to the large state space and to facilitate recursive abstraction, we defined the state by a set of features. Each state feature of manager M is defined recursively in terms of the features of M's children. For example, let the feature vector $totalResources(M) = \langle tr_1, ..., tr_m \rangle$ be the total amount of resources controlled by manager M (where m is the number of different

resource types). It can be defined recursively as follows: $totalResources(M) = \sum_{c \in children(M)} totalResources(c)$. That is, the total resources controlled by a manager is the sum of the total resources controlled by its children. For an individual I_i, $totalResources(I_i) = I_i$.

Some features cannot be defined recursively in a straightforward way. Instead, they are defined in terms of other recursive features. For example, let $averageResources(M)$ be a feature vector of the average amount of resources controlled by any individual in $members(M)$. This feature can be defined as $averageReources(M) = totalResources(M)/size(M)$. $totalResources(M)$ is described above, while $size(M)$ is the total number of individuals in an organization and is recursively defined as $size(M) = \sum_{c \in children(M)} size(c)$.

The recursive features defined above are assumed constant throughout the system lifetime. For example, $size(M)$ will return the number of individuals controlled by manager M, even if at a specific time none of these individuals is free. Clearly, for allocation purposes, one needs more *dynamic* features that reflect the *current* state of the system. For each static feature, a corresponding dynamic feature is defined preceded by the keyword *avail*. For example, the number of individuals not allocated to tasks $= availSize(M) = size(M) - \sum_{C \in LOC(M)} size(C)$, and their aggregated resources $=$

$$availTotalResources(M) = totalResources(M) - \sum_{C \in LOC(M)} totalResources(C)$$

In our implementation, tasks allocation always starts from the root manager (even if a task is received/discovered at a lower manager it is propagated up the hierarchy to the root manager). This restriction and the strict tree control architecture simplify communication and maintaining state consistency. The reason is that an agent can receive a request to do a task only from its manager. Since there is only one manager for each agent, each manager knows the state of its children through the request/response messages exchange. For example, manager M initially knows its child M_1 has 100 of CPU resource. M asks M_1 to form a coalition with at least 50 CPU resource. M_1 replies that it formed a coalition of 60 CPU resource (because, for example, M_1 controls 5 agents of 20 CPU resource each). M now knows that M_1 has only 40 CPU resource available.

As a result, when a manager asks a child to form a coalition, the manager knows *a priori* that a capable coalition will be formed. What is not known is how much resources will be wasted. In the last example, manager M knows that its child M_1 has 100 units of CPU resource. When M asks M_1 to form a sub-coalition with at least 50 CPU units, M knows that M_1 will commit a coalition with 50 or more CPU units, but M does not know exactly how much CPU units. For example, if M_1 controls only one agent with 100 CPU units then this agent will be the formed coalition and 50 extra CPU units might be wasted.

As usual, nothing comes for free. While abstraction significantly enhances the scalability of the system, the price of abstraction is loss of information.

A manager higher in the hierarchy "sees" fewer details about its organization. This leads to uncertainty in the manager state, and hence makes the local decision process more difficult to optimize. Section 4.7 discusses how the hierarchy affect the quality of abstraction.

4.5 Task Decomposition

When a manager M selects a child M_i to be asked for resources (for an incoming task T), M partially decomposes T to T_i (using heuristics that will be described shortly). As described in Section 4.4, a manager M only sees abstract features of its child M_i. Using this information, M needs to find T_i such that the expected excess of resources is minimized. What makes this difficult is that when a manager M decomposes T into T_i it does not know the exact state of M_i, but only an abstraction of it.

The partial decomposition heuristic we use, which is outlined in Algorithm 2, is to request from each child a multiple, α, of the average available resources it controls; i.e., $\alpha \times \frac{availTotalResources(M_i)}{availSize(M_i)}$. The intuition behind the heuristic is as follows. If all individuals controlled by M_i are identical, the heuristic is optimal. As individuals become more diverse in the resources they control, the heuristic still gives a good approximate decomposition that may succeed without wasting many resources.

Let us elaborate at Algorithm 2 in more detail. Because agents can not participate in more than one coalition, the minimum of the ratio l_j (in the algorithm) over all resource types is selected and used for all other resource types. Also to ensure progress, α is at least 1. Finally, the utility of the decomposed task is proportional to the total of the decomposed resources.

Algorithm 2 decomposeTask(T, M_i)

INPUT: task $T = \langle u, rr_1, ..., rr_m \rangle$ AND manager M_i
OUTPUT: task $T_i = \langle u_i, rr_{i,1}, ..., rr_{i,m} \rangle$

1: $AR_i \leftarrow availTotalResources(M_i) = \langle ar_{i,1}, ..., ar_{i,m} \rangle$
2: $z_i \leftarrow availSize(M_i)$
3: $\forall j : l_j \leftarrow \lfloor z_i \times \frac{rr_j}{ar_j} \rfloor$
4: $\alpha \leftarrow min_j(l_j)$
5: $\alpha \leftarrow max(\alpha, 1)$
6: $rr_{i,j} \leftarrow min(\alpha \times \frac{ar_{i,j}}{z_i}, rr_j)$
7: $u_i \leftarrow u \times \frac{\sum_j rr_{i,j}}{\sum_j rr_j}$
8: return T_i

For example, let $T = \langle u = 100, rr_1 = 50, rr_2 = 150 \rangle$, $availTotalResources(M_4) = \langle ar_{4,1} = 100, ar_{4,2} = 100 \rangle$, and $availSize(M_4) = 10$. Using the algorithm below we get $\alpha = 5$ and hence $T_4 = \langle u_4 = 50, rr_{4,1} = 50, rr_{4,2} = 50 \rangle$. Note that asking M_4 for as much as possible will result in wasted resources. For

example, the decomposed task $T_4' = \langle u_4' = 50, rr_{4,1}' = 50, rr_{4,2}' = 100 \rangle$ can only be satisfied if all individuals controlled by M_4 are allocated, resulting in 50 units of resource type 1 being wasted.

Note that the above heuristic algorithm is not optimal. In the previous example, if the whole organization only has 150 units of resource type 2 available, then the decomposed task T_4' may be better than T_4. Because of that, we allow each manager to select the same child more than once to fine tune the decomposition at the expense of more communication and time cost.

4.6 Learning

A key factor in the performance of our system is how a manager selects its actions (function *selectAction* in Algorithm 1). In particular, in what order a manager should ask each child for its contribution. We modeled this as a Markov Decision Process, MDP, then used reinforcement learning (RL) techniques to learn a good local policy for each manager. This section briefly describes the MDP model and the RL algorithm this work uses to learn the manager's policy. The section also describes how this work uses neural nets in conjunction with RL to cope with large state space. Before getting into the details of the model, some terms need to be defined:

System/Environment. These terms are used interchangeably to refer to anything outside the agent. A state of manager M (when it receives task T) consists of the abstract states of each child $M_i \in children(M)$, the resources required by T and its utility.

Action. Whatever an agent can do is an action. In a manager, there is an action corresponding to each child.

Reward. A real number indicating the quality of the last executed action. In other words, the agent executes an action and then receives its immediate reward (utility) from the system. From Algorithm 1, intermediate rewards are small negative rewards to reflect the communication and the processing costs of each additional step spent forming the coalition. Once a manager M successfully allocates a coalition to task T, it gains a reward equal to T's utility. Note that we can implicitly indicate our preferences by modifying the reward function. For example, in [8] the author prefers coalitions of smaller size. This can be achieved by adjusting the reward function accordingly (e.g., dividing the utility gained by the size of the coalition formed). Note that even if T is a subtask of another task T', the rewards received by M are independent of whether the coalition formation for T' will succeed or not. This *recursive optimality* speeds up learning, while not affecting the quality of the formed coalitions.

State. Ideally, the state of the system at a certain time should include every bit of detail about this system. However, for all practical purposes, only part of the system that would affect decision is important. If the state of an agent does not capture enough details of the real world, the agent may

fail to learn an optimal policy and the best it can do is to learn a near optimal policy.

Policy. The policy $\pi(s)$ is a table that specifies for every state the action that should be taken. The goal of a learning algorithm is to learn an *optimal* policy $\pi^*(s)$, i.e., a policy that specifies for every state the *best* action such that the total reward gained (by the agent) is maximized.

Decision Cycle. When an agent starts in a given state s, executes an action a, receives a reward r, and moves to the next state s', then this completes a decision cycle. This decision cycle is defined by the tuple $\langle s, a, r, s' \rangle$.

The model used in this work is Markov Decision Process, or MDP. In this model, the agent starts in a certain state s. The agent decides which *action* to execute. Upon executing an action a, the agent receives a *reward* r and the system moves to another state s'. The process continues until the system reaches a terminal state (if none exists, the process continues). An MDP model is completely defined by four components: $\langle S, A, P, R \rangle$, where

- S is the set of system states.
- A is the set of actions available for the agent to choose from.
- $P(s, a, s')$ is a transition probability function, i.e., the probability that the system will transit from state s to state s' if the agent executes action a. The uncertainty in coalition formation is due to the abstraction and the fact that a child might have allocated a task that its parent does not know about. Because it is difficult to analytically compute such transition probability, we used a model free learning algorithm as we discuss shortly.
- $R(s, a, s')$ is the expected reward function, if the system is in state s, the agent applies action a, and the system's next state is s'

In the field of operations research, it is assumed that all four components of the MDP are known and the optimal policy can be found using dynamic programming techniques [10]. However, in real domains, the P and R components are usually unknown. These two components together characterize the dynamics of the system in which the agent operates. They are called the *environment model.*

Reinforcement learning algorithms can be used in these cases as they make no assumptions regarding the environment model, and hence they are *model free.* These algorithms use *decision cycle tuples* to approximate the P and R functions. Decision cycle tuples can be obtained by executing actions in the system and receiving rewards. i.e., an agent observes its current state s and executes the best action a according to its policy, then observes the resulting reward r and next state s'. These four values constitute a decision cycle tuple.

This work uses a well-known algorithm, *Q-learn*[10] to automatically find the optimal policy. The main idea of the original algorithm is as follows. For every state and action pair $\langle s, a \rangle$, a real value $Q(s, a)$ is stored. These values are initialized randomly (or in any arbitrary way). They are then updated using the following equation (also known as Bellman's Equation):

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

Where $\langle s, a, r, s' \rangle$ is a decision cycle tuple; α and γ are learning parameters and are called *learning rate* and *discounting factor* respectively. As the agent moves from state to state, executing actions and receiving rewards, the values stored in Q converges and can be used to determine the optimal policy. Q-learning learns in an incremental and interactive manner; as an agent gains more experience, its performance improves. This is important in domains containing huge number of states, many of which will not be visited. The best action to perform in a given state s is $a^* = \pi^*(s) = argmax_a Q(s,a)$. The details explaining the intuition behind the algorithm and a proof of its correctness is beyond the scope of this chapter and can be found in [10]. We used the Q-learning algorithm with neural nets to approximate action values.

It should be noted that for this algorithm (and any other Reinforcement Learning algorithm) to work correctly, the agent needs to try all actions at every state "a large enough" number of times. One way to achieve this is to select an action randomly ϵ% of the time, and in the remaining $1-\epsilon$% pick the best action. This simple algorithm is called ϵ-*greedy exploration* algorithm [10] and ϵ is called the *exploration rate*. Typically, ϵ is initially large (to allow the agent to try more actions) and then decreases over time. We use a decaying exploration rate so that agents explore less as they gain more experience. We also tried using eligibility tracing, but the learning algorithm often diverged so this approach was dropped.

Neural Nets

Since the state of a manager includes the amount of available resources of each of its children and the amount of recourses required by the incoming task, the state space is very large. This prohibits the use of traditional Q-learning algorithm which uses a table to store the value of every state and action pair. Alternatively, functional approximators can be used. The idea here is to use a parametrized function instead of a table to approximate the values of actions (i.e. approximates $Q(s,a)$). In this case, Q-learning algorithm is used to update the parameters of the function, which implicitly updates the value of the action.

Here we use neural nets as the functional approximator. Q-learning is used to update the weights of the neural net. The details are beyond the scope of this document but can be found in [10]. A separate neural net is used to approximate the value of each action/child as shown in Figure 5. This uses more memory space (because of storing more neural nets), but provides better approximation as the weights of each neural net can be better fitted to the corresponding action/child.

We explored several techniques to speed up the learning further. One technique involved minimizing the input fed to each neural net. The key observation is that the value of choosing a child M_i depends mainly on M_i's state, and to a lesser extent on the other children's states.

4.7 Organization Structure

In this work, the underlying organization can be viewed as a search tree. Our distributed algorithm searches the same search tree several times for each task and for each episode. Each time, the search has a different start state (where and when the task is discovered) and different goal state (the set of individuals — leaves — that form the coalition.)

To optimize performance, not only does one need to learn a good *search mechanism*, as we do here, but also to find an *organization* that for a specific environment model and agent population yields the best performance. The interesting question is whether by modifying the search tree can the search mechanism perform better. The closest analogue in classical AI is the use of macro operators, which adds edges to the search tree to speedup the search. In our case there is more flexibility, as the search tree can be modified in whatever way.

While in this work we do not tackle the hard problem of optimizing the organization, we verify the effect of the underlying organization on the performance of the overall system. Our experiments verify this by testing different organization structures of the same agent population and same tasks distribution, as described in Section 5.

5 Experiments and Results

5.1 Setup

In our experiments we compare three possible policies: random, greedy, and learning. The random policy just picks a child at random. The greedy policy selects the child M_i with the highest preference value $p_i = \sum_{k=1}^{m} min(cr_{i,k}, rr_k)$, which measures how much resources M_i can contribute to the incoming task. For example, let the incoming task $T = \langle u = 100, rr_1 = 50, rr_2 = 150\rangle$ and let manager M has two possible children M_1 and M_2 where $availTotalResources(M_1) = \langle cr_{1,1} = 200, cr_{1,2} = 0\rangle$, $availTotalResources(M_2) = \langle cr_{2,1} = 0, cr_{2,2} = 200\rangle$, $p_1 = 50$ and $p_2 = 150$. Thus M will select M_2.

The experiments try to evaluate the effect of both the underlying organization and learning the local policy on the system's performance. To do so, we compared the performance of the same agent population under five organizations and the three local policies described above. In the tested agent population, agents control two types of resources, and the fall into 6 types of agents:

Type A controls $\langle cr_{A,1} = 2, cr_{A,2} = 2\rangle$ resources
Type B controls $\langle cr_{B,1} = 10, cr_{B,2} = 10\rangle$ resources
Type C controls $\langle cr_{C,1} = 0, cr_{C,2} = 30\rangle$ resources
Type D controls $\langle cr_{D,1} = 1, cr_{D,2} = 10\rangle$ resources
Type E controls $\langle cr_{E,1} = 20, cr_{E,2} = 2\rangle$ resources
Type F controls $\langle cr_{F,1} = 8, cr_{F,2} = 0\rangle$ resources

These classes represent different specializations among agents. Four of the studied organizations are shown in Figure 7. Organization $HOMOGEN$ is homogeneous. Agents of each type are clustered together, then similar types (e.g., A and B) are clustered together. Organization $SEMI-HOMOG$ is semi-homogeneous. Each couple of agents of similar types are clustered together, then similar clusters are clustered together. Organization $SHORT$ is similar to $HOMOGEN$, but one level of the hierarchy is omitted. Finally, organization $RANDOM$ has the same "structure" of $SHORT$, but individual agents are assigned randomly to each cluster.

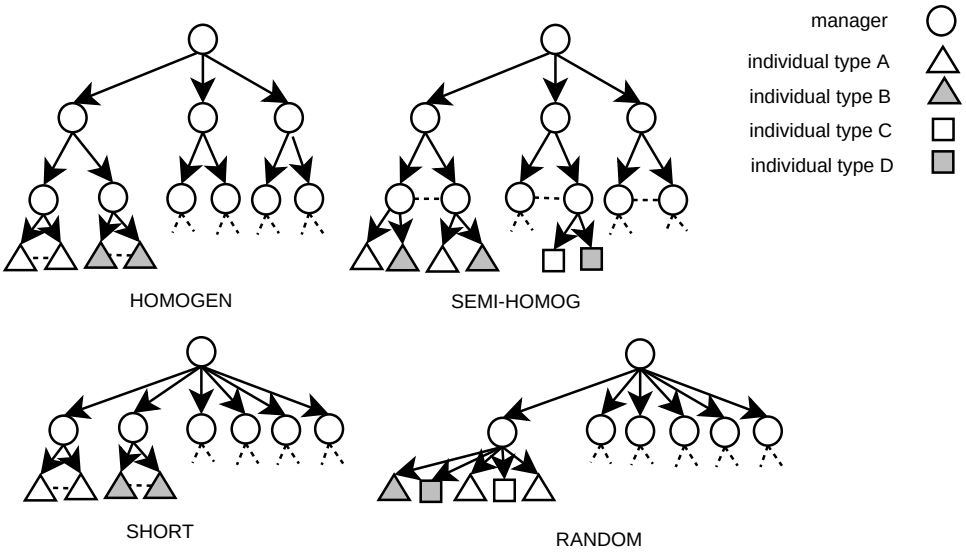

Fig. 7. Different Organization Structures.

Because the above four organizations (unlike the fifth organization described below) involve distributed decision making, we refer to the three local policies as: distributed learned policy (Distrib-Learn), distributed random policy (Distrib-Random), and distributed greedy policy (Distrib-Greedy), whenever we compare performance against the fifth organization.

The fifth organization is the centralized organization, $CENTRALIZED$, where there is only one manager connected to every other agent. This orga-

nization is tested using the random policy (Center-Random) and the greedy policy (Center-Greedy). The learned policy is not tested for this organization because the state of the centralized manager is exponential in the number of individuals, which is 40 in our experiments. We use this organization as a base line for comparison.

Results for every organization/technique combination were computed over 10 simulation runs. Each simulation run consisted of 30,000 episodes. Seven tasks arrive at every episode and are randomly picked from a bag of tasks. Tasks in the bag are generated randomly such that each task requires between 4 and 10 agents to be accomplished. Each task has an associated payoff, which is 1750 on average (it depends on the amount of resources each task requests). At any episode, the resources required by arriving tasks exceed the resources available to the system. The cost of every message (requesting a coalition or responding with a formed coalition) costs -1. Each Decision cycle (i.e. a time-step in forming a coalition) costs -1.

Our experiments focused only on 40 individuals and 10 managers so we can easily hand code different organization structures and study their effect. However, to verify the scalability of our approach, we tested it in a population of 90 agents and 13 managers. Agents were organized in a way similar to organization H and were randomly generated (using 9 distributions to represent 9 different classes of agents). Tasks were also randomly generated (from two different distributions). We plan further experiments on even larger populations and on the use of clustering techniques to automatically generate different organizations.

5.2 Results

Figures 8 shows the average utility for different organizations and policies when things have stabilized (i.e. learned policy converged). As expected, Center-Random performed worst. Distrib-Random performed better than Center-Random.[6] Center-Greedy is better than both. Our approach, Distrib-Learn, outperformed all other policies for all organization structures, except when using a random organization structure.

Figure 9 illustrates how the performance of our system improves as agents gain more experience (i.e., witness more episodes). Interestingly, Distrib-Greedy, performed worse than Distrib-Random and Distrib-Learn in all organizations except RANDOM, where it performed better than both. We think this is due to the fact that the greedy policy is based on a heuristic, which might perform best in some contexts and worst in others. That is also why the greedy policy has the highest deviation. In our experiments with larger agent population (90 agents), Distrib-Learn was better than other policies, achieving 35% more utility than Center-Random and at least 20% better than Distrib-Random and Distrib-Greedy.

[6] We believe this is due to the goal decomposition component of the organization, which encodes part of the domain knowledge.

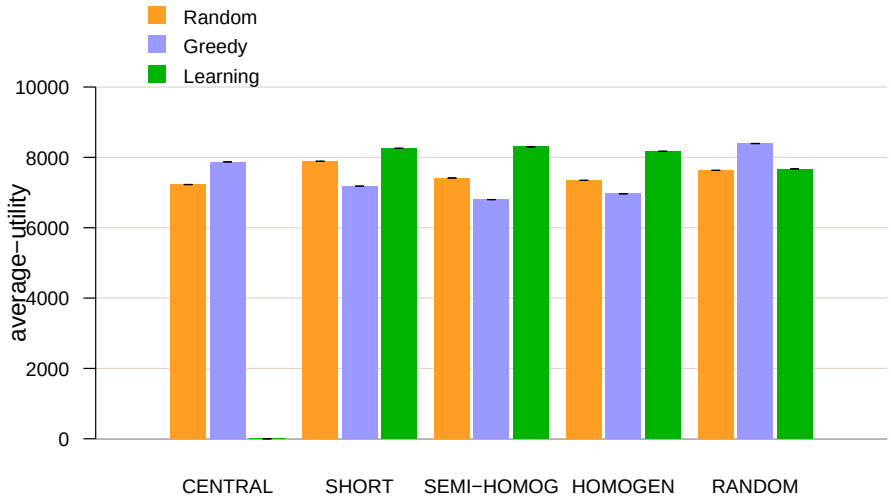

Fig. 8. Average utility for random, greedy and learned policies and for different organizations.

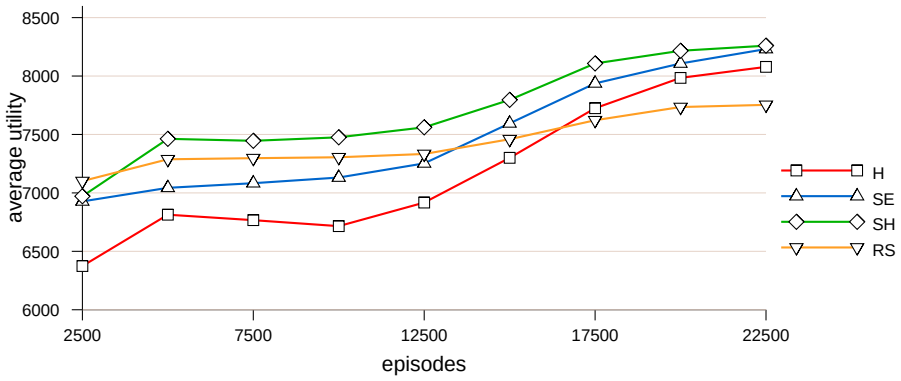

Fig. 9. Learning curve.

More importantly, Distrib-Learn is more stable than other approaches as Figure 10 shows. The standard deviation (of achieved utility) using Center-Greedy is 70% worse than Distrib-Learn with SE organization. Center-Random is 30% worse than Distrib-Learn. Also Distrib-Greedy was the worst for all organizations except $RANDOM$. We had similar results with the larger agent population. Distrib-Learn had the least standard deviation, which was around one third that of Distrib-Greedy.

While it is expected that our approach performs better than distributed random and greedy policies, one might expect centralized policies to perform better than our approach, due to the inaccuracies (incurred by abstraction) and the limitations on managers' choices (imposed by the organization). We believe the reason our system performed better is the underlying organization, which implicitly encodes domain knowledge. In other words, limiting the actions of a manager is actually a good thing if these actions are the best actions this manager can take. This is also why a bad organization may lower performance. The underlying organization also affects the abstraction quality. A random organization contains more information, hence it will be abstracted poorly (the entropy principle).

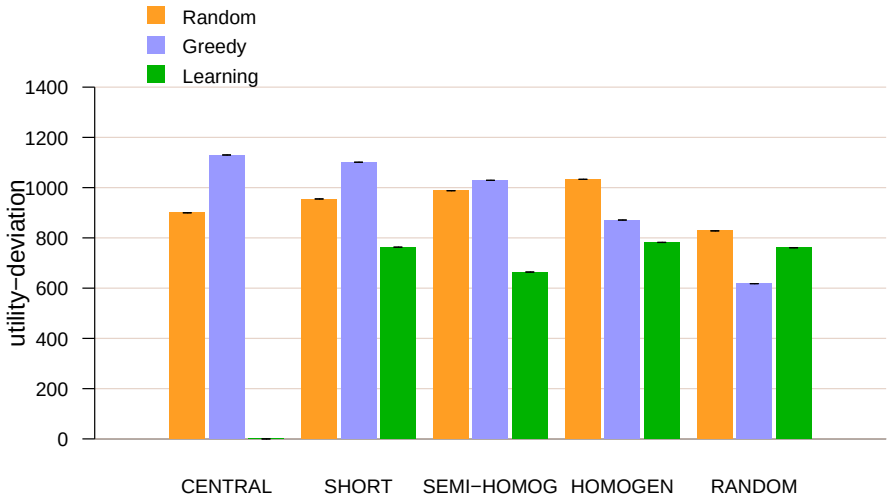

Fig. 10. Utility standard deviation for random, greedy and learned policies and for different organizations.

Figure 11 compares the average number of exchanged messages per task. As expected this measure increases as the organization hierarchy gets taller. Centralized approaches exchange fewer messages. Still, learning the local decision reduces the number of exchanged messages. Finally, Figure 12 shows the average resources wasted. Center-Greedy wasted 20% more resources than Distrib-Learn, while Center-Random wasted 40% more. We got similar results for the larger agent population.

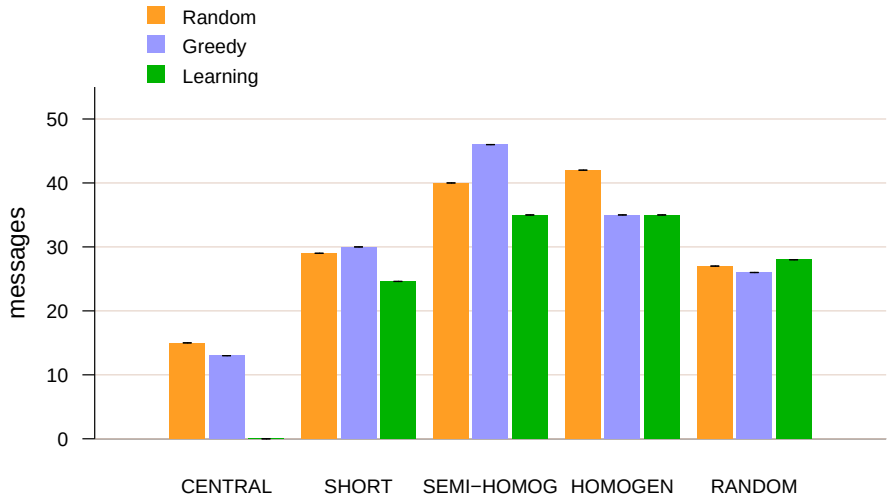

Fig. 11. Messages average for random, greedy and learned policies and for different organizations.

6 Related work

In [8], the authors presented a distributed algorithm for solving the coalition formation problem. The algorithm requires exponential time but is optimal. It neither used learning nor an underlying organization. Our algorithm is an approximation algorithm that returns a solution in polynomial time.

The work in [6] introduced an anytime coalition structure generation algorithm (the term *coalition structure* refers to the solution of the coalition formation problem). As in [8], the work did not use any organization for guiding the coalition formation search and assumed a black box function that given a feasible solution returns the *value* of such solution, while we evaluate the solution based on the total utility of the allocated tasks.

The work in [9] used a multi-leveled learning scheme to form coalitions. Both reinforcement learning and case based reasoning were used. Unlike our work, they do not use an underlying organization, which limits the scalability of their approach (their experiments were limited to 4 agents).

Though some extensions to the original contract net protocol [12] proposed the use of an underlying organization, none of these extensions (to our knowledge) provided a formal model of such an organization, nor evaluated the performance for different organizations, unlike our work here.

In the brokering research area [4] not enough attention is given to scalability or coalition formation, the main focus of our work. In some sense, our use of an underlying organization can be viewed as a hierarchy of brokering

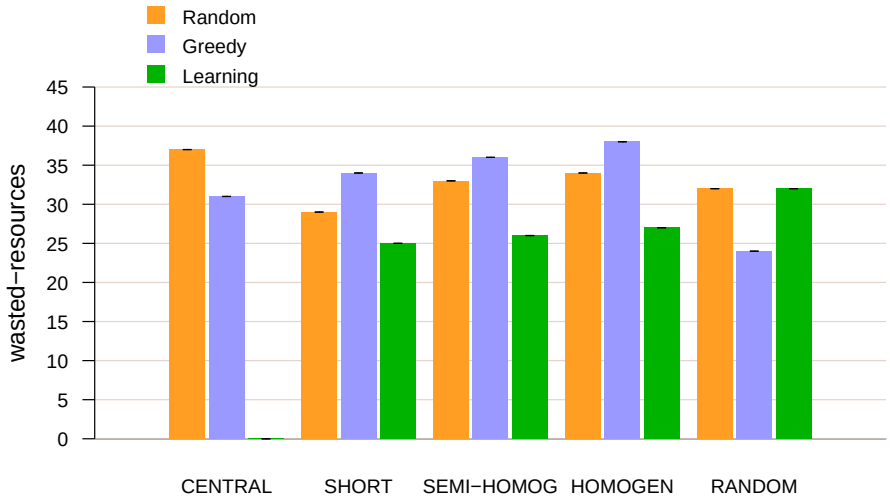

Fig. 12. Average percentage of wasted resources for random, greedy and learned policies and for different organizations.

agents. Integrating brokering techniques into our framework is an interesting future work direction.

The coalition formation problem can be mapped to a multi-unit combinatorial auction. However, none of the algorithms developed for combinatorial auctions [7] make use of *stable knowledge*, which remains relatively unchanged throughout the system lifetime. This includes agents' capabilities (e.g., same bids repeat for consecutive auctions) and task patterns (e.g., consecutive auctions follow some statistical model). We on the other hand try to exploit this knowledge implicitly using an underlying organization and learning the local decision of each agent in the organization.

The work in [3] tried to provide a unified framework for coordination in MAS. In this framework each agent follows a set of behaviors that differ in their level of abstraction. As behaviors become more and more abstract, an (implicit) underlying organization becomes more and more apparent. The goal of such an organization is to optimize the immediate individual goals. In our work, the goal of the organization is to optimize the coalition formation process, which *indirectly* optimizes the performance of the MAS as a whole.

In [5], the authors proposed and analyzed a simplified and restricted model of an organization, which takes only processing and communication costs into account. While they tried also to analyze the performance of different organizations, unlike our work there was no notion of resources, individual capabilities, coalition capabilities, task requirements, coalition formation, and learning.

In our approach a group of agents co-learn to work together in an organization. This can be viewed as distributed learning of a hierarchical policy that targets recursive optimality [1]. However, none of the work in hierarchical learning area (HRL) introduced the concepts of quantitative/dynamic state *abstraction* and task *decomposition*. We defined these concepts to decouple agents' local decision problems while minimizing communication, and hence achieve scalability. Our work also quantitatively evaluates how different action hierarchies affect the learning performance. Figure 13 illustrates the relationship between our work and HRL. HRL learns a policy for the whole action hierarchy in a single agent. In our approach each agent concurrently learn a *sub-policy*. Collectively, these sub-policies constitute a global hierarchical policy, but learning of sub-policies is distributed.

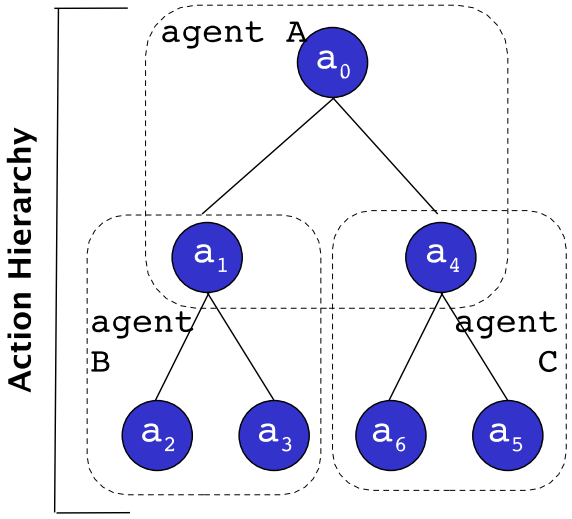

Fig. 13. Relationship between Hierarchical Reinforcement Learning and our approach.

7 Conclusions and Future work

In this work we defined a generic problem solving framework using an underlying organization, and applied it to the coalition formation problem. We provided an algorithm for the local decision to be made by each agent, given state abstractions from other agents and its decomposed task. We used Q-learning with neural nets as functional approximators to improve the local decision. Our initial results show that our approach outperformed both random and greedy policies for most of the organizations we studied. It achieved

higher utility and more stability with a smaller percentage of wasted resources and fewer exchanged messages. The results also verify the scalability of our approach as it still outperforms the other approaches we studied for larger systems.

In future, we aim to study a wider variety of organizations for different types of environments. We will also investigate further our abstraction and decomposition schemes, as we believe better schemes can considerably improve the learned policy performance. We also plan to study the optimization of the underlying organization and how this interacts with optimizing the hierarchical policy.

References

1. A. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. In *Discrete Event Systems journal*, volume 13, pages 41–77, 2003.
2. K. Decker and V. Lesser. Designing a family of coordination algorithms. In *1st International Conference on Multi-Agent Systems*, 1995.
3. E. Durfee and T. Montgomery. Coordination as distributed search in a hierarchical behavior space. *IEEE Trans. on Systems, Man, and Cybernetics*, 21:1363–1378, 1991.
4. M. Klusch and K. Sycara. Brokering and matchmaking for coordination of agent societies: A survey. chapter 8, pages 197–224. 2001.
5. Y. pa So and E. Durfee. Designing tree-structured organizations for computational agents. *Computational and Mathematical Organization Theory*, 2(3):219–246, 1996.
6. T. Sandholm and et al. Coalition structure generation with worst case guarantee. *Proceedings of the 3rd Internation Conference on Autonomous Agents*, 1999.
7. T. Sandholm and et al. Winner determination in combinatorial auction generalizations. *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems*, 2002.
8. O. Shehory. Methods for task allocation via agent coalition formation. *Artificial Intelligence Journal*, 101(1–2):165–200, 1998.
9. K. Soh and X. Li. An integrated multilevel learning approach to multiagent coalition formation. *International Joint Conference on Artificial Intelligence*, pages 619–624, August 2003.
10. R. Sutton and A. Barto. *Reinforcment Learning: An Introduction*. MIT Press, 1999.
11. M. Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, 1997.
12. D. N. W. Shen. An agent-based approach for dynamic manufacturing scheduling. *Proceedings of the 3rd International Conference on the Practical Applications of Agents and Multi-Agent Systems*, 1998.