Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021)

Decentralized Refinement Planning and Acting

Ruoxi Li, Sunandita Patra, Dana S. Nau
Dept. of Computer Science, and Institute for Systems Research
University of Maryland, College Park, MD 20742, USA
rli12314@cs.umd.edu, patras@umd.edu, nau@cs.umd.edu

Abstract

We describe Dec-RPAE, a system for decentralized multi-agent acting and planning in partially observable and non-deterministic environments. The system includes both an acting component and an online planning component. The acting component is similar to RAE, a well-known acting engine, but incorporates changes that enable it to be used by multiple autonomous agents working independently in a collaborative setting. Each agent runs a local copy of Dec-RPAE, with a set of hierarchical refinement methods using operational models that specify various ways to accomplish its designated tasks. To perform actions, the agent uses Dec-RPAE's acting component to execute the methods in the agent's environment. To advise the acting component on *which* method to execute, the planning component repeatedly does Monte Carlo simulations of the methods to estimate their potential outcomes. Agents can communicate with each other to exchange information about their states, tasks, goals, and plans in order to cooperatively succeed in their respective missions. Our experimental results demonstrate that Dec-RPAE is useful for improving the agents' performances.

1 Introduction

Recent work on the integration of acting and planning has advocated a hierarchical organization of an actor's deliberation functions, with online planning throughout the acting process. This view has led to the development of the RAE acting algorithm (Ghallab, Nau, and Traverso 2016) and the RAE+UPOM integrated planning-and-acting system (Patra et al. 2020). A key limitation of the above work is that it is essentially *single-agent* planning and acting. Although several of the test domains for RAE+UPOM involved multiple robots, in each case the planning and acting were done by a single centralized system.

In this paper, we extend the above approach to accommodate multiple agents that do their planning and acting in a decentralized fashion. Our contributions are as follows:

- We introduce Dec-RPAE, a decentralized multi-agent planning and acting engine that uses operational models like the ones used in RAE. It consists of two components, Dec-RAE and D-UPOM:
  - Dec-RAE, the decentralized acting component, is a generalization of RAE. Multiple agents can run Dec-RAE concurrently in a decentralized fashion, and can use it to perform actions, communicate and delegate tasks among each other.
  - D-UPOM, the decentralized planning component, uses a Monte-Carlo rollout technique based on the well-known UCT algorithm (Kocsis and Szepesvári 2006). D-UPOM is a decentralized adaptation of UPOM (Patra et al. 2020). In D-UPOM, if an agent i needs to delegate a task τ to some other agent j, i can ask j to predict how well they can accomplish τ, and delegate τ to the agent that can do the best job.

  Dec-RAE can also be used with RAE's UPOM planner (Patra et al. 2020) instead of D-UPOM, but D-UPOM has the advantage of supporting inter-agent plan coordination. We prove that D-UPOM's Monte Carlo rollouts will converge to optimal choices of methods for Dec-RAE to use.
- We present experimental evaluations of Dec-RPAE in two domains. The results show that additional Monte-Carlo rollouts in the planning component improve the performance of the acting component in both single-agent and multi-agent settings. We observe that communication enables coordination between agents thereby improves their performance to a large extent. Our experiments also show that agents can successfully coordinate their actions, and D-UPOM works in a setting where tasks need to be delegated among each other recursively.

The rest of this paper is organized as follows: (1) background, (2) definitions, (3) Dec-RPAE, (4) experimental results, (5) discussion, (6) related work, and (7) conclusion.

2 Background

In RAE, *Refinement methods* and *commands* are *operational models* of actions that tell the agent what to do. A refinement method for a task t gives a procedure for accomplishing t. The procedure may include any of the usual programming constructs (if-then-else, loops, etc.), as well as commands to send to the actor's execution platform, and subtasks to refine further using the actor's refinement methods. RAE may have several methods available for the same task, in which case it can consult the UPOM planner to get a recommendation of which method to try first. UPOM uses the same operational model that the RAE uses and performs UCT-like Monte Carlo Tree Search (MCTS) in the space of operational models in a simulated environment to predict the actions' outcomes.

Monte Carlo tree search (MCTS) is a promising approach for online planning because it efficiently searches over long planning horizons and is anytime (Browne et al. 2012). In treating the choice of child node to expand in the MCT as a multi-armed bandit problem, the UCT algorithm balances the tradeoff between exploration and exploitation, to find a near-optimal plan. Yao et al. (2020) proposes a MCTS-based solver for hierarchically organized intention progression problems, and Wichlacz et al. (2020) applies MCTS in HTN Planning, but they only works in a single-agent, deterministic and static task environment. Dec-MCTS (Best et al. 2018) features decentralized multi-agent MCTS, but it does not have a hierarchically organized deliberation that also allows task allocation.

Extended from the above work, our work features decentralized online multi-agent MCTS in the space of hierarchical operational models that enables inter-agent communication and task allocation.

3 Definitions

We now describe the parameters for our decentralized multi-agent refinement planning and acting domain, and give some examples. $\mathcal{I}$ is the set of all agents. Each agent $a \in \mathcal{I}$ has its own local knowledge $\Sigma_a = \langle S_a, \mathcal{T}_a, \mathcal{M}_a, \mathcal{C}_a \rangle$, where

- S_a is a set of local states an agent a may be in; each state $s \in S_a$ is represented using a state-variable formulation similar to the one in (Ghallab, Nau, and Traverso 2016).
- $\mathcal{T}_a$ is a finite set of tasks and events that agent a may have to deal with.
- $\mathcal{M}_a$ is the set of refinement methods, each of which gives a way for a to perform a task $\tau \in \mathcal{T}_a$.
- $\mathcal{C}_a$ is a finite set of primitive actions (commands) that can be carried out by the execution platform of agent a. The actions can have non-deterministic outcomes.

A refinement method is composed of 4 elements, where 1) $head$ specifies the name and parameters of the method, where the number of parameters could be arbitrary greater than that in the task which it is related to, 2) $tasks$ indicates the task that the method is capable of refining, 3) $body$ gives a procedure to accomplish a the task by performing subtasks, commands and state variable assignments. We get a refinement method instance by assigning values to the free parameters of a method.

Example Let us illustrate the above definitions by an example domain, where several robots including *drones* and *roombas* (robot vacuum cleaners) forage for some target objects (e.g., dirt) in an initially unknown terrain. This domain includes but is not limited to:

- a set of agents $\mathcal{I} = \{d_1, r_1, r_2\}$, where d_1 is a drone, and r_1 and r_2 are roombas,
- a set of refinement methods $\mathcal{M}_{r_1}$ for agent r_1 that includes $\{$m1-cleanSet(s), m1-clean(s), m1-broadcastGoal$(g)\}$,
- a set of refinement methods $\mathcal{M}_{r_2}$ for agent r_2 that includes $\{$m2-cleanSet$(s, l)\}$,
- a set of refinement methods $\mathcal{M}_{d_1}$ for agent d_1 that includes $\{$m1-search(a), m1-planTrajectory(a), m1-flyTo$(l)\}$,
- a set of commands $\mathcal{C}_{d_1}$ for agent d_1 that includes $\{$observe$(l)\}$,
- a set of tasks $\mathcal{T}_{d_1}$ for agent d_1 that includes $\{$flyTo(l), planTrajectory(a), search$(a)\}$.
- sets of tasks $\mathcal{T}_{r_1}, \mathcal{T}_{r_2}$ for agents r_1 and r_2 that both include $\{$cleanSet(s), clean$(l)\}$.

Below we show the pseudo code for three example refinement methods: m1-search(a), cleanSet(s), and cleanSet(s).

```
m1-search(a):
   task: search(a)
   body: trajectory ← do task planTrajectory(a)
         for l in trajectory:
            do task flyTo(l)
            execute command observe(l)
            if l has dirt:
               delegate task clean(l) to agent r ∈ {r1, r2}
```

m1-search(a) is a method for the drone d_1 to search area a along a trajectory, perform the command observe(l) to check if an intermediate location l is dirty before *delegating* a task to a either roomba r_1 or r_2 to clean up the location.

```
m1-cleanSet(s): # greedy method
   task: cleanSet(s)
   body: if s is ∅ then return
         l ← closest l ∈ s
         do task broadcastGoal(l)
         do task clean(l); remove l from s
         do task cleanSet(s)
```

Task cleanSet(s) requires a set of locations s to be cleaned. Roomba r_1 has one method for this task, m1-cleanSet(s). m1-cleanSet(s) is a *greedy* method that cleans the closest location in s, then calls cleanSet recursively for the other locations.

Roomba r_2 does not have the above method, but instead has a *simple* method m2-cleanSet(s, l) that refines task cleanSet(s), where l indicates the first location to clean:

```
m2-cleanSet(s, l): # simple method
   task: cleanSet(s)
   body: if s is ∅ then return
         do task clean(l); remove l from s
         do task cleanSet(s)
```

l's value is automatically assigned with some predefined rules (e.g, $l \in s$). In this case, there are $|s|$ method instances applicable to task cleanSet(s) for agent r_2.

The current context for an incoming external task τ is represented via a *refinement stack* σ which keeps track of how far RAE has progressed in the refinement method that is refining τ. E.g., the initial refinement stack is

$$\sigma_0 = \langle(\tau_0, m_0, 1)\rangle, \tag{1}$$

where τ_0 is a task, m_0 is a method that is relevant for τ_0 and applicable in s_0, and 1 indicates that the current progress is the on the first step of method m_0.

Now we define the *refinement planning problem* for each agent $a \in \mathcal{I}$ as $\Pi_a = (\Sigma_a, s_0, \sigma_0, U_a)$, where s_0 is the initial state, σ_0 is the initial refinement stack, and U_a is a's utility function. The planner's objective is to optimize the utility function U_a.

Figure 1 illustrates the *space of refinement trees* (Patra et al. 2020) which is composed of 3 types of nodes: 1) a *disjunction* node is a task followed by its applicable method instances; 2) a *sequence* node is a method instance m followed by all the steps; and 3) a *sampling* node for an action a has the possible nondeterministic outcomes of a as its children.

Decentralized Refinement planning is essentially a series of independent tree search procedures by a team of communicating agents $\mathcal{I}$ that over their local spaces of refinement trees in order for the agent to find a near-optimal method to use for refining a task τ. The ultimate objective is to optimize $f(U_{a_1}, U_{a_2}, ..., U_{a_n})$, where $a_1, a_2, ..., a_n \in \mathcal{I}_\tau$, and $\mathcal{I}_\tau \subseteq \mathcal{I}$. $\mathcal{I}_\tau$ is the set of agents that are relevant to task τ. We use summation as $f()$ in our experiments.

4 Dec-RPAE

Dec-RPAE is a decentralized refinement planning and acting engine that enables heterogeneous robots to cooperatively operate in a partially-observable, non-deterministic environment. Dec-RPAE consists primarily of Dec-RAE (the acting engine), and D-UPOM (the Monte-Carlo rollout algorithm that is used for planning). Each agent has its own copies of Dec-RAE and D-UPOM, and its own domain knowledge, execution platform, and internal state.

Dec-RAE is a modified version of RAE (Patra et al. 2020), modified to run concurrently on multiple agents and to enable communication among those agents. Here we specify 4 types of communication messages: 1) local *state information* obtained from the agent's action and observation history; 2) a *goal* that the agent is actively pursuing; 3) a *task* that the agent needs one of the other agents to accomplish (e.g. a subtask τ in agent A's method that needs to be delegated to agent B or C); 4) *plan information* at any abstraction levels (e.g., estimated utility/reward/efficiency of the plan, estimated state change resulted from the plan, or the explicit plan).

Our agents are built with both actuators and sensors to send and receive communication signals. Commands are given to agents to sense the communication network, send messages, or read messages. Received messages are buffered in memory waiting for the agent to read. We acknowledge the fact that communication is neither free, nor guaranteed to succeed. Therefore, each communication command is associated with a cost and a probability of success just like other commands.

Like RAE, Dec-RAE uses method instances to perform tasks, and when multiple method instances are available for the same task, it either arbitrarily chooses the method instance or consults its planner to get information about which method instance to use. Unlike RAE, Dec-RAE may have refinement methods that specify tasks to be delegated to other agents (see m1-search(a) in Section 3). In such a case, Dec-RAE either arbitrarily chooses the delegatee from among a set of candidate agents, or requests plan information from among the candidate agents to get advice about which candidate to choose. To provide the task delegator with plan information, each candidate agent obtains the estimated utility of doing the delegated task by calling its local planner using its local state information. Dead cycles caused by recursive task delegation can be prevented by specifying the rules of task delegation in the methods.

<u>Select-Method$(s, \tau, \sigma, d_{max}, n)$:</u>
$\tilde{m} \leftarrow failure$; $d \leftarrow 0$
$global\ Q_{s,\sigma}$ # global for the agent
1 $s' \leftarrow$ Abstraction(s)
for *n times* **do**
| D-UPOM $(s', \text{push}((\tau, nil, nil), \sigma), d_{max})$
$\tilde{m} \leftarrow \text{argmax}_{m \in M} Q_{s,\sigma}(m)$ return $\tilde{m}$, $Q_{s,\sigma}(\tilde{m})$

Algorithm 1: Select-Method returns the method instance with the highest estimated utility and the estimated utility by performing n D-UPOM rollouts.

The planner is Select-Method (Algorithm 1), a wrapper around the D-UPOM algorithm (see Section 4.1 for details). During the acting phase, when an agent (e.g., agent i) needs to witfully select the method instance to refine the task τ in its local state s and a refinement stack σ, it calls Select-Method with two control parameters: n, the number of rollouts, and d_{max}, the maximum rollout length (which dictates the total number of sub-tasks and actions in a rollout).

Select-Method calls D-UPOM n times in a simulated environment, each call to D-UPOM proceeds until the rollout length reaches d_{max}. Abstraction(s) (line 1) is the abstracted agent state that is used in D-UPOM's simulated environment.

4.1 D-UPOM

If no tasks are delegated, D-UPOM is essentially UPOM. However, suppose agent i is performing task τ that has a subtask τ_2 that needs to be delegated to some other agent (e.g., agent j or k), as illustrated in Figure 1. Without $plan$ communication, agent i has no idea how well the other agent can accomplish τ_2 or what effect the other agent will have on the environment after finishing task τ_2. Thus, agent i is only able to delegate the task to an agent that is selected arbitrarily or based on agent i's subjective heuristics. In order to generate more optimal plans, agents need to coordinate with each other in the planning process by communicating their local plans with each other. With D-UPOM, agent i can ask the candidate delegatees to predict how well they can accomplish τ_2 as well as the resulting change to the state of the environment, then delegate τ_2 to the agent that expects to do the best job and leave an ideal state of the environment so agent i can successfully perform the rest of the task τ.

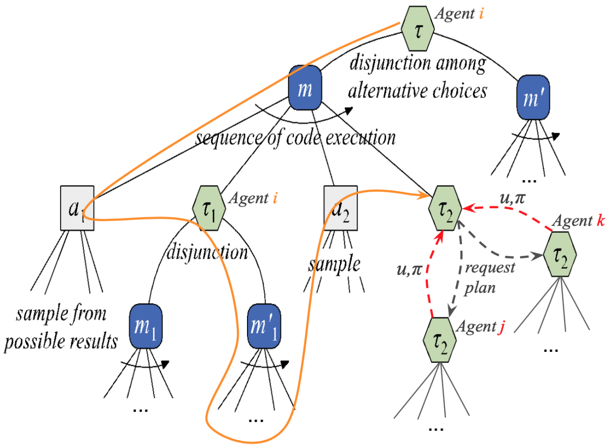

Figure 1: The space of the refinement trees for a decentralized planning problem that requires cooperation among agents i, j, and k. Agent i is supposed to delegate task τ_2 to agent j or agent k. The orange arrow indicates a D-UPOM rollout for agent i.

D-UPOM naturally supports market-based task allocation during the acting time. When the task τ' is potentially delegated to different agents who are capable of accomplishing it, each agent plans for τ' using its methods and returns the estimated rewards. The agent who has a method that obtains the highest reward will be chosen to accomplish τ'.

Specifically, when task τ' is delegated to another agent $a \in \mathcal{I}$ (Algorithm 2, line 1), one needs to request those agents to plan for τ' by calling their local planners (line 2) in parallel. Agent a is supposed to receive the request to plan for τ', feed its locally observed state information into its local copy of Select-Method to get the optimal method $\tilde{m}$ and its corresponding plan utility $Q_{s,\sigma}(\tilde{m})$, and send the plan utility $Q_{s,\sigma}(\tilde{m})$ as well as the abstract plan π back to the agent who requests it. The estimated utility of task τ' is the largest plan utility u returned from agent a. Then state s is updated according to the corresponding abstract plan π (line 3). Since the plan utilities are generated by complete rollouts from other agents, it is unnecessary to make the same requests, In which case, Request-Plan(τ',a) will return the stored value.

next(σ, s) (line 4) is the refinement stack resulting from performing $m[i]$ in state s, where $(\tau, m, i) = top(\sigma)$. Applicable$(s, \tau)$ (line 5) is the set of method instances applicable to τ in state s. U(s, c, s') (line 7) is the utility obtained from entering state s' by executing action c on state s.

5 Convergence of D-UPOM Rollouts

In this section, we discuss the convergence of Select-Method and D-UPOM. As we know, UCT is demonstrated to converge on a finite horizon MDP with a probability of not finding the optimal action at the root node that goes to zero at a polynomial rate as the number of rollouts grows to infinity (Kocsis and Szepesvári 2006, Theorem 6). Since we can map the search strategy of each agent's D-UPOM to UCT[1], and map the search space of each agent's D-UPOM to a MDP (Section 5.1), given that the task delegation among $a \in \mathcal{I}$ is not cyclic or infinitely recursive, the *decentralized refinement planning* process among $a \in \mathcal{I}$ is equivalent to a finite number of MDPs being solved using UCT. Thus Select-Method should converge with monotonic utility functions.

<u>D-UPOM(s, σ, d):</u>
if $\sigma = \langle\rangle$ *or* $d = 0$ **then** return 0
$(\tau, m, i) \leftarrow$ top(σ)
if $m = nil$ *or* $m[i]$ *is a task* τ' **then**
    if $m = nil$ **then** $\tau' \leftarrow \tau$ # for the first task
1   **if** τ' *is to be delegated* **then**
        $\mathcal{I} \leftarrow$ set of candidate agents
2        $a \leftarrow$ argmax$_{a \in \mathcal{I}}$Request-Plan $(\tau', a)[u]$
3        $s' \leftarrow$ state s updated according to Request-Plan $(\tau', a)[\pi]$
4   return Request-Plan $(\tau', a)[u]$ + D-UPOM$(s', \text{next}(\sigma, s'), d-1)$
    else
        if $N_{s,\sigma}(\tau')$ *is not initialized yet* **then**
5            $M' \leftarrow$ Applicable(s, τ')
            if $M' = 0$ **then** return 0
            $N_{s,\sigma}(\tau') \leftarrow 0$
            for $m' \in M'$ **do**
                $N_{s,\sigma}(m') \leftarrow 0$, $Q_{s,\sigma}(m') \leftarrow 0$
        $Untried_m \leftarrow \{m' \in M' | N_{s,\sigma}(m') = 0\}$
        if $Untried_m \neq \emptyset$ **then**
            $m_c \leftarrow$ randomly select from $Untried_m$
6   **else** $m_c \leftarrow$ argmax$_{m \in M'} \phi(m, \tau')$
        $\sigma' \leftarrow$ push$((\tau', m_c, 1), \text{next}(\sigma, s))$
        $u \leftarrow$ D-UPOM$(s, \sigma', d-1)$
        $Q_{s,\sigma}(m_c) \leftarrow \frac{N_{s,\sigma}(m_c) \times Q_{s,\sigma}(m_c) + u}{1 + N_{s,\sigma}(m_c)}$
        $N_{s,\sigma}(m_c) \leftarrow N_{s,\sigma}(m_c) + 1$
        return u
if $m[i]$ *is an assignment* **then**
    $s' \leftarrow$ state s updated according to $m[i]$
    return D-UPOM$(s', \text{next}(\sigma, s'), d)$
if $m[i]$ *is a command* c **then**
    $s' \leftarrow$ Sample(s, c)
7   return $U(s, c, s')$ + D-UPOM$(s', \text{next}(\sigma, s'), d-1)$

<u>Request-Plan(τ, a):</u>
If never requested, request agent a to call its own copy of Select-Method to plan for τ and obtain the estimated utility u and abstract plan π. Otherwise, use the previously obtained values. Return u and π.

Algorithm 2: D-UPOM and Request-Plan. In line 6, $\phi(m, \tau) = Q_{s,\sigma}(m) + C\sqrt{\log N_{s,\sigma}(\tau) / N_{s,\sigma}(m)}$, where $C > 0$.

[1]The mapping of D-UPOM's search strategy to an equivalent UCT and with non-additive utility functions is available at ⟨https://www.cs.umd.edu/%7epatras/DecRPAETheory.pdf⟩.

Search Space for D-UPOM rollouts Let $\Sigma_a = \langle S_a, \mathcal{T}_a, \mathcal{M}_a, \mathcal{C}_a \rangle$ be the local knowledge of agent $a \in \mathcal{I}$. Select-Method searches over Σ_a in a simulator. For each $a \in \mathcal{I}$, $\mathcal{T}_a$, $\mathcal{M}_a$, and $\mathcal{C}_a$ are all finite, and every sequence of steps generated by the methods in $\mathcal{M}_a$ (including task delegation) is finite. For $s \in S_a$ and $c \in \mathcal{C}_a$, we let $\gamma_a(s, c) \subseteq S$ be the set of all states that may be produced by simulating c's execution in s. For each $s' \in \gamma_a(s, c)$, we let $\mathcal{P}(s, c, s')$ be the probability that state s' will be produced if we simulate c's execution in state s. The *refinement planning problem* is $\Pi_a = (\Sigma_a, s_0, \sigma_0, U_a)$, $a \in \mathcal{I}$.

Rollouts A *rollout* in Σ_a is a sequence of pairs

$$\rho = \langle (\sigma_0, s_0), (\sigma_1, s_1), \ldots, (\sigma_n, s_n) \rangle \quad (2)$$

satisfying the following properties:

- each s_i is a state, and each σ_i is a refinement stack;
- $\forall i > 0$ there is a nonzero probability that s_j and σ_j are the next state and refinement stack after s_{i-1} and σ_{i-1};
- (σ_n, s_n) is a termination point for D-UPOM.

If the final refinement stack is the empty stack $\sigma_n = \langle\rangle$ then rollout ρ is successful; otherwise ρ fails. In a top-level call to D-UPOM, the initial refinement stack is σ_0. In all subsequent refinement stacks produced by D-UPOM, we will say that a refinement stack σ is *reachable* in Σ_a (i.e., reachable from a top-level call to D-UPOM) if there exists a rollout $\rho = \langle (\sigma_0, s_0), (\sigma_1, s_1), \ldots, (\sigma_n, s_n) \rangle$, such that σ_0 satisfies (1) and $\sigma \in \{\sigma_0, \ldots, \sigma_n\}$. We let $\mathcal{R}(\Sigma_a)$ be the set of all refinement stacks that are reachable in Σ_a. Since every sequence of steps generated by the methods (including task delegations to other agents) in $\mathcal{M}_a$ is finite, it follows that $\mathcal{R}(\Sigma_a)$ is also finite.

For each pair (σ_j, s_j) in ρ, let (τ_j, m_j, i_j) be the top element of σ_j. If $m_j[i_j]$ is an action, then the next element of ρ is a pair (σ_{j+1}, s_{j+1}) in which s_{j+1} is the state produced by executing the action $m_j[i_j]$. In Σ_a, this corresponds to the state transition $(s_j, m_j[i_j], s_{j+1})$. Thus the set of state transitions in ρ is $t_\rho = \{(s_j, m_j[i_j], s_{j+1}) \mid$ (σ_j, s_j) and (σ_{j+1}, s_{j+1}) are members of ρ, $(\tau_j, m_j, i_j) = \mathsf{top}(\sigma_j)$, and $m_j[i_j]$ is an action $\}$.

Thus if U is additive, then

$$U(\rho) = \sum_{(s,c,s') \in t_\rho} U(s, c, s'). \quad (3)$$

5.1 Defining the MDP for Each Agent

We want to define a MDP Ψ for each agent a such that choosing among methods in Σ_a corresponds to choosing among actions in Ψ. The easiest way to do this is to let all of Σ_a's actions, methods and delegated tasks $\mathcal{T}_{del}$ (to other agents) be actions in Ψ. We will write Ψ as

$$\Psi = (S^\Psi, \mathcal{C}^\Psi, s_0^\Psi, S_g^\Psi, \gamma^\Psi, \mathcal{P}^\Psi, U^\Psi) \quad (4)$$

where

$S^\Psi = \mathcal{R}(\Sigma_a) \times S_a$ is the set of states,

$\mathcal{C}^\Psi = \mathcal{M}_a \cup \mathcal{C}_a \cup \mathcal{T}_{del}$ is the set of actions,

$s_0^\Psi = (\sigma_0, s_0)$ is the initial state,

$S_g^\Psi = \{(\langle\rangle, s) \mid s \in S\}$ is the set of goal states,

and the state-transition function γ^Ψ, state-transition probability function $\mathcal{P}^\Psi$, and utility function U^Ψ are as follows.

State transitions To define γ^Ψ and $\mathcal{P}^\Psi$, we must first define which actions are applicable in each state. Let $(\sigma, s) \in S^\Psi$, and $(\tau, m, i) = \mathsf{top}(\sigma)$. Then the set of actions that are applicable to (σ, s) in Ψ is $\mathsf{Applicable}^\Psi((\sigma, s))$

$$= \begin{cases} \mathsf{Instances}(\mathcal{M}, m[i], s), & \text{if } m[i] \text{ is a task,} \\ \{m[i]\}, & \text{if } m[i] \text{ is an action,} \\ \{\text{Delegate to agent } i \in \mathcal{I}\}, & \text{if } m[i] \in \mathcal{T}_{del}. \end{cases} \quad (5)$$

Thus if $c \in \mathsf{Applicable}^\Psi((\sigma, s))$, then there are three cases for what $\gamma^\Psi(s^\Psi, c)$ and $\mathcal{P}^\Psi(s, c, s')$ might be:

- **Case 1:** $m[i]$ is a task in $\mathcal{M}_a$, and $c \in \mathsf{Instances}(\mathcal{M}_a, m[i], s)$. In this case, the next refinement stack will be produced by pushing a new stack frame $\phi = (m[i], c, 1)$ onto σ. The state s will remain unchanged. Thus the next state in Ψ will be $(\phi + \sigma, s)$, where '+' denotes concatenation. Thus, $\gamma^\Psi((\sigma, s), c) = \{(\phi + \sigma, s)\}$, $\mathcal{P}^\Psi[(\sigma, s), c, (\phi + \sigma, s)] = 1$.

- **Case 2:** $m[i]$ is an action in $\mathcal{C}_a$, and $c = m[i]$. Then c's possible outcomes in Ψ correspond one-to-one to its possible outcomes in Σ_a. More specifically, if γ_a is the state-transition function for Σ_a, then
$\gamma^\Psi((\sigma, s), c) = \{(Next(\sigma, s'), s') \mid s' \in \gamma_a(s, c)\}$, and,
$$\mathcal{P}^\Psi((\sigma, s), c, (\sigma', s'))) = \begin{cases} \mathcal{P}_a(s, c, s'), & \text{if } (\sigma', s') \in \gamma^\Psi((\sigma, s), c), \\ 0, & \text{otherwise.} \end{cases}$$

- **Case 3:** $m[i]$ is a task $\tau_d \in \mathcal{T}_{del}$ delegated to other agents. Let $\sigma = (m, \tau_d, j) + \sigma'$. Let $b \in \mathcal{I}$ be a chosen agent for delegation and s' be the state resulting from b accomplishing τ_d. Then, $\gamma^\Psi((\sigma, s), b) = \{(m, \mathit{Next}(m, j), j + 1) + \sigma', s')\}$. $\mathcal{P}^\Psi((\sigma, s), b, (\sigma', s')) =$
$$\begin{cases} \mathcal{P}_b(s, \tau_d, s'), & \text{if } (\sigma', s') \in \gamma^\Psi((\sigma, s), b), \\ 0, & \text{otherwise.} \end{cases}$$

Rollouts A *rollout* of Π^Ψ is any sequence of states and actions of Ψ, $\rho^\Psi = \langle (\sigma_0, s_0), c_1, (\sigma_1, s_1), c_2, \ldots, (\sigma_{n-1}, s_{n-1}), c_n, (\sigma_n, s_n) \rangle$, such that for $i = 1, \ldots, n$, $c_i \in \mathsf{Applicable}^\Psi(\sigma_{i-1}, s_{i-1})$ and $\mathcal{P}^\Psi((\sigma_{i-1}, s_{i-1}), c_i, (\sigma_i, s_i)) > 0$. The rollout is *successful* if $(\sigma_n, s_n) \in S_g^\Psi$, and unsuccessful otherwise.

Utility We can define U^Ψ directly from U. If ρ^Ψ is the rollout given above, then the corresponding rollout in Σ_a is $\rho = \langle (\sigma_0, s_0), (\sigma_1, s_1), \ldots, (\sigma_{n-1}, s_{n-1}), (\sigma_n, s_n) \rangle$, and $U^\Psi(\rho^\Psi) = U(\rho)$. If U is additive, then so is U^Ψ. In this case, Ψ satisfies the definition of an MDP with initial state (Mausam 2012).

6 Experimental Evaluation

We evaluate Dec-RPAE in two different domains, the Dirt Collection domain, and the Spring Door domain. Experimental results are illustrated and discussed in this section.

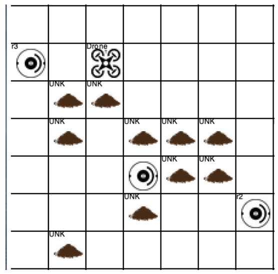

Figure 2: The Dirt Collection Simulator.

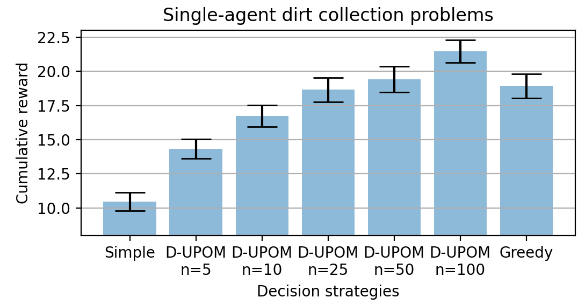

Figure 3: In each experiment there are 1 *roomba* agent and 16 Dirt objects in a 7×7 grid. Each *roomba* agent type's average cumulative reward and standard error is obtained and plotted from solving 50 randomly generated problems, each problem runs 5 times.

6.1 Dirt Collection Domain

Multi-agent Foraging is a canonical testbed for cooperative multi-agent systems, in which a collection of robots has to search and transport objects to specific locations (Zedadra et al. 2017). As a special case of this problem, we developed a Dirt Collection Simulator based on the code from Russell and Norvig (2009), where multiple roombas and drones cooperatively clean up a finite amount of dirt objects scattered randomly within an $N \times N$ grid. Each dirt object is associated with a value, which corresponds to the reward for the roomba when the roomba collects it. Each roomba has a limited amount of time budget to carry out actions including moving forward, turning left, turning right, picking up the dirt right beneath it, and communicating with other agents. A drone can detect the locations of dirt, communicate with roombas and delegate cleaning tasks to roombas. Each action takes a certain time period to complete. The domain is nondeterministic, because each action (command) has a small probability (2 - 4%) of failing. The objective is for the roomba team to maximize the cumulative reward from collecting dirt objects with a limited time budget.

The roombas in our experiments have several different types of decision strategies. In Figures 3, Table 1 and 2, these are denoted by the following labels:

- The label *greedy* means that m1-cleanSet(s, l) (see Section 3) is the only method that the agent has for the *cleanSet* task (though it also has methods for other tasks). A greedy roomba always pursues the closet target.
- The label *simple* means that m2-cleanSet(s, l) (see Section 3) is the agent's only method for the *cleanSet* task. A simple roomba cleans the dirt in an arbitrary order.
- The label D-UPOM indicates that the agent has the same methods as a simple agent, but uses D-UPOM to plan for the choice of the method instances.
- The label *n* is the number of UCT rollouts that is configured in a D-UPOM agent.
- The label *comm* indicates that goal communication is enabled using a task broadcastGoal(g) in which the agent broadcasts information about the target it is pursuing.

Within each experiment, all roomba agents (if there are more than one) use the same strategy.

Each of the first set of experiments (Figure 3) involves only one agent but no tasks being delegated. D-UPOM in this case is essentially single-agent UPOM. The experiments show that a D-UPOM agent performs much better than a simple agent, since a reactive simple agent would clean the set of locations in an arbitrary sequence, while the D-UPOM agent tries to plan for the optimal sequence. The performance of a D-UPOM agent further improves as the number of UCT rollouts increases, which surpasses a greedy agent's performance with 50 rollouts.

Multi-agent dirt collection problems

Roomba	Greedy	Greedy	Simple	D-UPOM
Comm	No	Yes	Yes	Yes
Reward	27.39	33.41	11.92	42.80
SE	2.27	2.61	1.62	2.33

Table 1: In each experiment, there are 4 *roomba* agents and 16 *dirt* objects in a 10×10 grid. D-UPOM agents has n = 50. Each *roomba* team's average cumulative reward and standard error (SE) is obtained from solving 30 randomly generated problems, each problem runs 5 times.

The second set of experiments (Table 1) involves multiple communicating roombas but no tasks being delegated. With goal communication enabled, agents would be aware of each other's goals, thus, are able to adjust their own goals accordingly to avoid duplication of efforts. We observe a 48.5% improvement in the greedy agent team's performance with goal communication, compared to the performance without any communication. The D-UPOM agent team with communication performs slightly better than a greedy agent team with communication, which is consistent with the result from the single-agent experiments.

The third set of experiments (Table 2) shows the performance of D-UPOM in situations where tasks are delegated among heterogeneous agents. In each experiment, 4 roombas with the same decision strategy (same denotation as is shown in previous experiments) await for a drone to locate a cluster of dirt and delegate the cleaning tasks to one of them. A reactive drone randomly assigns the cleaning task to one of the roombas. On the contrary, a D-UPOM drone has the same methods as a reactive drone does, but uses D-

Multi-agent dirt collection problems (task delegation)

Roomba	D-UPOM	D-UPOM	Simple	Greedy
Drone	D-UPOM	Reactive	Reactive	Reactive
Reward	24.12	21.83	10.28	19.81
SE	0.44	0.41	0.23	0.41
Act #	42.05	40.27	32.44	31.50
Plan time	16.19	7.77	0.00	0.00

Table 2: Each experiment involves 4 *roombas* and 1 *drone*, and approximately 12 *dirt* in a 10×10 grid. D-UPOM agents has n = 100. Each team's average cumulative reward and its standard error (SE), total number of actions (act #), and cumulative planning time (plan time) are obtained from 150 randomly generated problems, each runs 5 times.

UPOM to choose the optimal method instances. It tends to assign the cleaning task to the roomba that is closest to the dirt cluster (i.e., the roomba that has the highest estimated utility of cleaning the cluster). The result shows that a drone can improve the task performance by using D-UPOM. We also observe that the team of greedy roombas and a reactive drone score 114% more than the team of naive roombas and a reactive drone, but 11.7% less than the team of D-UPOM (n=100) roombas and a reactive drone. Although the greedy approach runs instantly since no simulation-based planning process is needed, the average cumulative planning time in a task performed by the team of greedy roombas and a reactive drone is only 7.77 seconds on a 2.3 GHz Dual-Core Intel Core i5 processor. Considering that on average 40.27 actions are executed in a task, each action only needs 0.19 second of the planning time. In our experience, the time spent on planning is negligible in real-world applications.

6.2 Spring Door Domain

The Spring Door domain has several *robots* trying to move objects from one room to another in a facility with a mixture of spring doors and ordinary doors. Spring doors close themselves unless they are held by a robot. A robot cannot simultaneously carry an object and hold a spring door open, so it must ask for help from another robot in this situation. Any robot that's free can be the helper. Specifically, a *manager* agent delegates to some robot r_1 the task of moving an object to some location. If r_1 needs to pass a spring door while carrying the object, it will delegate some other robot r_2 to hold the door. The Spring Door domain has 7 tasks, 11 methods, and 9 actions. Each action takes a specific time period to complete. To add nondeterminism to the domain, each action has a small probability (3 - 5%) of failing. The utility of a solution is its efficiency (roughly, 1/cost). As discussed in (Patra et al. 2019), this requires a minor modification to the "+" operator in Algorithm 2, line 7.

The result (Figure 4) shows that the average efficiency of relocating an object is the lowest when the agents choose methods and delegatees reactively (i.e., the number of D-UPOM rollouts is 0). As the number of rollouts increases, they make more informed choices, thus achieve higher efficiency.

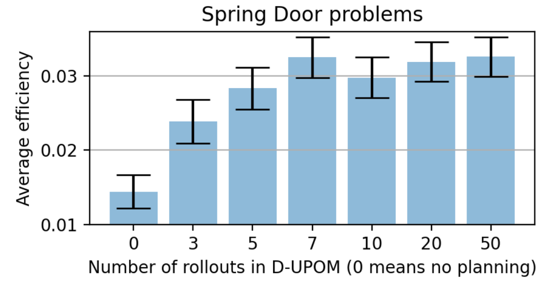

Figure 4: Each experiment involves one *manager* and 2 to 3 *robots* in a facility with 3 to 7 rooms. Each data point is the average efficiency for 50 randomly generated problems, running each problem 5 times. The vertical lines indicate standard error.

This experiment demonstrates Dec-RPAE's capability of handling recursively delegated tasks, as the *manager* delegates a task to a robot r_1, and r_1 delegates a subtask to another robot r_2. We also show that Dec-RPAE can plan for situations where decentralized agents need to coordinate their actions. This is made possible by r_2 communicating the expected change of the environment state to r_1, so r_1 will expect that the spring door will be held open by r_2.

7 Discussion

We don't yet support asynchronous decentralization (Kuter and Hamell 2018), instead, only one task is assigned to the robot team at a time. If multiple tasks are assigned to an agent, we can easily modify the code to make each agent buffer those tasks in a queue and process them one after another. Ideally, The task delegator should take into consideration that a busy delegatee may not be able to help with the delegated task immediately. The most naive way to deal with such a situation is for the delegator not to prioritize any candidate agent that is buffered with other tasks. However, sometimes the busy candidate is so capable that given the tasks at hand, it can still accomplish the delegated task efficiently in a timely manner. In order to let the delegator know so, the busy candidate needs to: 1) estimate when and on what state will its currently buffered tasks will finish, 2) plan for the delegated task supposing that the task begins at that time and on that state, and 3) send the estimated utility of the plan and the abstract plan with timestamps to the delegatee. We intend to explore this in our future work.

It is also possible that different candidate delegatees may leave the state of the environment different after finishing the delegated task, which might affect the delegator's performance for the rest of the task (if any) after the delegation. In that case we need to sample different candidate delegtees in D-UPOM, just like we sample non-deterministic actions.

In our experiments, communication commands are guaranteed to succeed. We have not done enough investigations in cases where communication is not always guaranteed, and agents might need to proactively look for communication signals (e.g., by going to a high ground where there is a better chance to re-establish communication with others).

A broader question that automotive agents need to decide is who, when, how, and what to communicate (Balch and Arkin 1995; Wei, Hindriks, and Jonker 2014). In our future work, we hope to make our system more resilient and intelligent in terms of communication.

8 Related Work

The multi-agent systems based on hierarchical task networks (HTN) (Obst and Boedecker 2006; Dix et al. 2003; Clement, Durfee, and Barrett 2007; Pellier and Fiorino 2007; Cardoso and Bordini 2019; Kuter and Hamell 2018), although have hierarchical deliberations, use abstract *descriptive models*. Compared to operational models that are used in Dec-RPAE, descriptive models (e.g., a classical precondition-and-effects action models) tell what the action will do, but not how to do it.

Auctions are the most common task-allocation mechanisms used in market-Based multi-robot coordination (Dias et al. 2006). Among studies in decentralized hierarchical planning systems that use market-Based task allocation, Zlot and Stentz (2006) focuses on how to do auctions of tasks, and it does not include a planning algorithm to produce the agents' bids for those tasks. DOMAP (Cardoso and Bordini 2019) has separate phases for goal allocation and individual HTN planning, while our approach integrates those phases by enabling recursive allocation of subtasks.

A Decentralized partially-observable Markov decision process (Dec-POMDP) is a framework for a team of collaborative agents to maximize a global reward based on local information. Each agent's individual policy maps from its action and observation histories to actions (Oliehoek 2012). Unfortunately, optimally solving Dec-POMDPs is NEXP-complete (Bernstein, Zilberstein, and Immerman 2013). In single-agent (i.e., MDP) domains, the options framework (SMDP) proposed by Sutton, Precup, and Singh (1999) uses higher-level, temporally extended macro-actions (or options) to represent and solve problems. Amato et al. (2019) extend the framework to the multi-agent case by introducing a Macro Dec-POMDP formulation with macro-actions modeled as options. It is an offline planner that can generate a joint policy to select the best option on each state for each agent, while our approach is a planning and acting engine that selects the best refinement method for each task online using operational models.

Our approach is essentially simulation-based planning, which shares some similarities with reinforcement learning (RL) (Kaelbling, Littman, and Moore 1996; Sutton and Barto 1998; Geffner and Bonet 2013; Leonetti, Iocchi, and Stone 2016; Garnelo, Arulkumaran, and Shanahan 2016), and MCTS is also a typical technique in RL to increase sample efficiency in simulation. In model-based RL, the model (e.g., system dynamics) is learned from real experience and gives rise to simulated experience. In our work, the simulator is given, and the operational models are much more complex than the actions used in model-based RL.

Both RAE and architectures based on BDI (Belief-Desire-Intention) models (De Silva, Meneguzzi, and Logan 2020; Yao et al. 2020; De Silva, Meneguzzi, and Logan 2018) rely on a reactive system, but with differences regarding their primitives as well as their methods or plan-rules. BDI systems rely on PDDL-like representations (eg. add or del operators) but RAE can handle any type of skill (e.g., physics-based simulators) with nondeterministic effects.

We know of no prior work on decentralized refinement (hierarchical) acting and online planning using operational models.

9 Conclusion

We have described Dec-RPAE, a system for decentralized multi-agent refinement planning and acting that uses operational models. We prove that if there are no exogenous events, D-UPOM's Monte Carlo rollouts will converge to optimal choices of methods for Dec-RAE to use. In our empirical evaluations of Dec-RPAE's performance in two domains, the results show that the system's performance is improved by performing additional Monte-Carlo rollouts in D-UPOM, and allowing agents to communicate. Our experiments also show D-UPOM's capability of handling recursive task delegation and action coordination.

Acknowledgments

This work has been supported in part by DARPA task order HR001119F0057, Lockheed Martin research agreement MRA17001006, NRL grant N00173191G001, and ONR grant N000142012257. The information in this paper does not necessarily reflect the position or policy of the funders, and no official endorsement should be inferred. We thank Phillip J. Dibona, William C. Regli, Paolo Traverso, and Malik Ghallab for their inspirations.

References

Amato, C.; Konidaris, G.; Kaelbling, L.; and How, J. 2019. Modeling and Planning with Macro-Actions in Decentralized POMDPs. *JAIR* 64: 817–859.

Balch, T.; and Arkin, R. C. 1995. Communication in Reactive Multiagent Robotic Systems. *Auton. Robots* 1(1): 27–52.

Bernstein, D. S.; Zilberstein, S.; and Immerman, N. 2013. The Complexity of Decentralized Control of Markov Decision Processes.

Best, G.; Forrai, M.; Mettu, R. R.; and Fitch, R. 2018. Planning-Aware Communication for Decentralised Multi-Robot Coordination. In *ICRA*, 1050–1057.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A Survey of Monte Carlo Tree Search Methods .

Cardoso, R. C.; and Bordini, R. H. 2019. Decentralised Planning for Multi-Agent Programming Platforms. In *AAMAS*, 799–807.

Clement, B.; Durfee, E.; and Barrett, A. 2007. Abstract Reasoning for Planning and Coordination. *J. Artificial Intelligence Research (JAIR)* 28: 453–515.

De Silva, L.; Meneguzzi, F.; and Logan, B. 2020. BDI Agent Architectures: A Survey. International Joint Conferences on Artificial Intelligence.

De Silva, L.; Meneguzzi, F. R.; and Logan, B. 2018. An operational semantics for a fragment of PRS. In *IJCAI*.

Dias, M. B.; Zlot, R.; Kalra, N.; and Stentz, A. 2006. Market-Based Multirobot Coordination: A Survey and Analysis. *Proceedings of the IEEE* 94(7): 1257–1270. doi: 10.1109/JPROC.2006.876939.

Dix, J.; Muñoz-Avila, H.; Nau, D. S.; and Zhang, L. 2003. IMPACTing SHOP: Putting an AI planner into a multi-agent environment. *Annals of Mathematics and Artificial Intelligence* 37(4): 381–407.

Garnelo, M.; Arulkumaran, K.; and Shanahan, M. 2016. Towards Deep Symbolic Reinforcement Learning. *CoRR* abs/1609.05518.

Geffner, H.; and Bonet, B. 2013. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool.

Ghallab, M.; Nau, D.; and Traverso, P. 2016. *Automated Planning and Acting*.

Kaelbling, L. P.; Littman, M. L.; and Moore, A. W. 1996. Reinforcement Learning: A Survey. *JAIR* 4: 237–285.

Kocsis, L.; and Szepesvári, C. 2006. Bandit based Monte-Carlo Planning. In *ECML*, 282–293.

Kuter, U.; and Hamell, J. 2018. Assumption-based Decentralized HTN Planning. In *Proceedings of the 1st ICAPS Workshop on Hierarchical Planning*.

Leonetti, M.; Iocchi, L.; and Stone, P. 2016. A synthesis of automated planning and reinforcement learning for efficient, robust decision-making. *Artificial Intelligence* 241: 103–130.

Mausam, A. K. 2012. *Planning with Markov decision processes: an AI perspective*. Morgan & Claypool Publishers.

Obst, O.; and Boedecker, J. 2006. Flexible Coordination of Multiagent Team Behavior Using HTN Planning. In *RoboCup 2005*, 521–528.

Oliehoek, F. A. 2012. *Decentralized POMDPs*, 471–503.

Patra, S.; Ghallab, M.; Nau, D.; and Traverso, P. 2019. Acting and Planning Using Operational Models. In *AAAI*, 7691–7698.

Patra, S.; Mason, J.; Kumar, A.; Ghallab, M.; Traverso, P.; and Nau, D. 2020. Integrating Acting, Planning and Learning in Hierarchical Operational Models. In *ICAPS*, 478–487.

Pellier, D.; and Fiorino, H. 2007. A Unified Framework Based on HTN and POP Approaches for Multi-Agent Planning. In *IAT*, 285–288.

Russell, S.; and Norvig, P. 2009. *Artificial Intelligence: A Modern Approach*. 3rd edition.

Sutton, R. S.; and Barto, A. G. 1998. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press.

Sutton, R. S.; Precup, D.; and Singh, S. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence* .

Wei, C.; Hindriks, K.; and Jonker, C. 2014. The Role of Communication in Coordination Protocols for Cooperative Robot Teams. In *ICAART*, volume 2.

Wichlacz, J.; Höller, D.; Torralba, Á.; and Hoffmann, J. 2020. Applying Monte-Carlo Tree Search in HTN Planning. In *SOCS*.

Yao, Y.; Alechina, N.; Logan, B.; and Thangarajah, J. 2020. Intention Progression under Uncertainty. In Bessiere, C., ed., *IJCAI*, 10–16.

Zedadra, O.; Jouandeau, N.; Seridi, H.; and Fortino, G. 2017. Multi-Agent Foraging: state-of-the-art and research challenges. *Complex Adaptive Systems Modeling* 5: 1–24.

Zlot, R.; and Stentz, A. 2006. Market-based Multirobot Coordination for Complex Tasks. *IJRR* 25(1): 73–101.