International Journal on Artificial Intelligence Tools
Vol. 21, No. 1 (2012) 1250003 (25 pages)

DOI: 10.1142/S0218213012500030

World Scientific
www.worldscientific.com

STOCHASTIC RESOURCE ALLOCATION IN MULTIAGENT ENVIRONMENTS: AN APPROACH BASED ON DISTRIBUTED Q-VALUES AND BOUNDED REAL-TIME DYNAMIC PROGRAMMING

PIERRICK PLAMONDON and BRAHIM CHAIB-DRAA
DAMAS Laboratory, Laval University, G1K 7P4, Québec, Canada
chaib@ift.ulaval.ca

Received 17 May 2010
Accepted 10 October 2011

This paper contributes to solve effectively stochastic resource allocation problems in multiagent environments. To address it, a distributed Q-values approach is proposed when the resources are distributed among agents a priori, but the actions made by an agent may influence the reward obtained by at least another agent. This distributed Q-values approach allows to coordinate agents' reward and thus permits to reduce the set of states and actions to consider. On the other hand, when the resources are available to all agents, no distributed Q-values is possible and tight lower and upper bounds are proposed for existing heuristic search algorithms.

Our experimental results demonstrate the efficiency of our distributed Q-values in terms of planning time as well as our tight bounds in terms of fast convergence and reduction of backups.

Keywords: Stochastic resource allocation; real-time dynamic programming; Markov decision processing; distributed Q-values.

1. Introduction

Dynamic resource allocation problems lie in general at the core of many real-world scheduling problems. In such problems, a scheduling process suggests the action (i.e. resources to allocate) to undertake to accomplish certain tasks, according to the perfectly observable state of the environment. When executing an action to realize a set of tasks, the stochastic nature of the problem induces probabilities on the next visited state. In general, the number of states is the combination of all possible specific states of each task and available resources. In this case, the number of possible actions in a state is the combination of each individual possible resource assignment to the tasks. The very high number of states and actions in this type of problem makes it very complex to solve.

In principle, this type of problems can be treated as Markov decision processes and solved using approximate dynamic programming (DP) algorithms.[1] In dis-

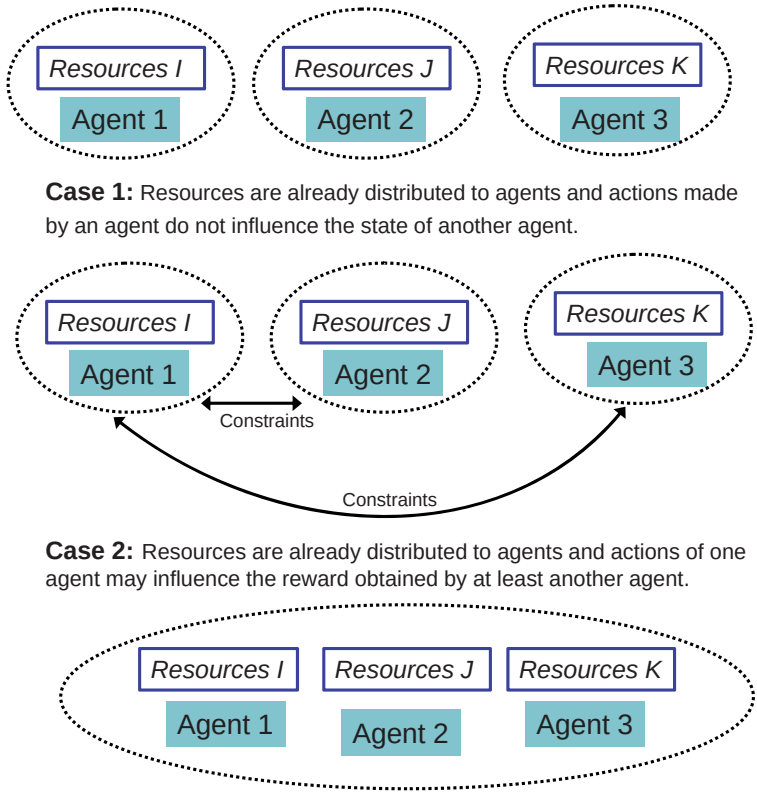


Fig. 1. Different types of resource allocation problems.

tributed environment inhabited by several agents, those agents can be face to three types of resource allocation problems, as can be seen on Figure 1. Firstly, the Case 1 where the resources are distributed among agents a priori, and the actions made by an agent do not influence the state of another agent, the globally optimal policy can be computed by finding a policy[a] separately for each agent. This problem can be solved using existing approaches of approximate dynamic programming as suggested for instance by Refs. 1–3. Examples of this case include: a network of computers where each has its own resources and does not impact others through its actions; robots having their predetermined tasks in a factory, etc.

The Case 2 of Figure 1 presents a second type of resource allocation problem. It is the case where the resources are distributed among agents a priori, but the actions made by an agent may influence the reward (or the cost) obtained by at least another agent. Examples here include major railroads, truck companies, etc. To solve this

[a]Finding a policy is also called "planning" and it consists (in a stochastic domain) to find a function which associates an action to a state.

problem efficiently, we propose a distributed Q-values approach where a planning agent manages each task and all agents have to share the limited resources. The planning process starts with the initial state s_0, in which, each agent computes her respective Q-value. Then, the planning agents are coordinated through an arbitrator to find the highest global Q-value by adding the respective possible Q-values of each agent. Doing so, allows to reduce the number of states and actions and consequently to get an approximate solution to the resource allocation problem.

Finally, when the resources are available to all agents, as in Case 3 of Figure 1, no distributed Q-values is possible. Examples here are robots in manufacturing, software agents through the web, etc. A common way of addressing this large stochastic problem is by using Markov Decision Processes (MDPs), and in particular real-time search where many algorithms have been developed recently. For instance, Real-Time Dynamic Programming (RTDP)[4] and its variants,[5–7] HDP (a combination of an heuristic search with dynamic programming),[8] and LAO* (a version of AO* with loops)[9] are all state-of-the-art heuristic search approaches in a stochastic environment. RTDP is an algorithm which updates states in trajectories from an initial state s_0 to a goal state s_g. The original RTDP approach is much more effective if the action space can be pruned of sub-optimal actions. To do this, McMahan,[6] Smith *et al.*,[7] and Singh and Cohn[10] proposed solving a stochastic problem using a RTDP type heuristic search with upper and lower bounds on the value of states. McMahan[6] and Smith *et al.*[7] suggested, in particular, an efficient trajectory of state updates to further speed up the convergence, when upper and lower bounds are given.

In this paper, we propose a new algorithm to define upper and lower bounds in the context of a RTDP heuristic search approach. We use, in particular, the concept of *marginal revenue*[11] to elaborate tight bounds. Our marginal revenue bounds are compared theoretically and empirically to the bounds proposed by Singh and Cohn.[10] Also, even if the algorithm used to obtain the optimal policy is RTDP, our bounds can be used with any other algorithm to solve an MDP. The only condition on the use of our bounds is to be in the context of stochastic constrained resource allocation.

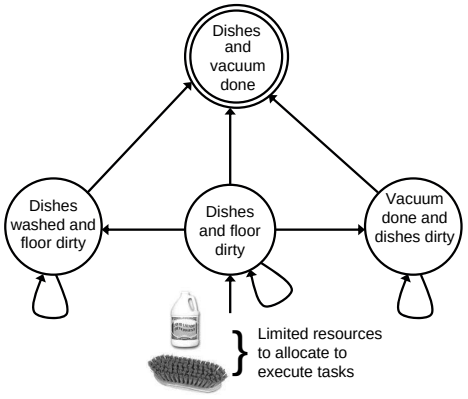
To sum up, this paper describes two contributions for solving stochastic resource allocation problems in multiagent environments. In the case where the resources are distributed among agents a priori but rewards depend on actions of other agents, an algorithm based on distributed Q-values is proposed. In the case where the resources are not distributed among agents a priori, the use of an existing search procedure is proposed. However this procedure is enforced by new lower/upper bounds which improve the performance. The next section describes a general view of our problem of interest. Section 3 depicts Markov Decision Processes (MDPs) which are used to model our problem. Afterwards, Section 4 sketches the main contributions of this paper. Section 5 details the experiments made and, finally, Section 6 concludes this paper.

2. Problem Formulation

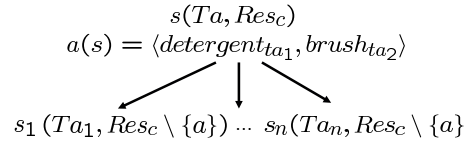
Our problem of interest are naval maritime environments which are known to be very complex environments with tight real-time constraints. In case of an own platform attack by incoming missiles, the commander must make fast decisions by considering several factors to ensure himself of the best possible survival of the own platform and its crew. Under such real-time constraints, it can often happen that the commander makes errors because of the complexity of the environment or the stress which the situation can generate. In these conditions, a computer is tremendously faster than a human and consequently, it can suggest decisions in time, thus, facilitating the task of resource allocation by a commander. In this context, we are developing a Decision-Support System (DSS) that focusses specifically on some particular aspects of such maritime environments, in order to reduce the complexity of the domain. Our primary focus is on the *Resource allocation* process.

A simple resource allocation problem, as illustrated on Figure 2(a), is one where there are the following two tasks to realize: $ta_1 = \{\text{wash the dishes}\}$, and $ta_2 = \{\text{clean the floor}\}$. These two tasks are either, or not, in the realized state. To realize them, two types of resources are assumed: $res_1 = \{\text{brush}\}$, and $res_2 = \{\text{detergent}\}$. A computer has to compute the optimal allocation of these resources to cleaning robot agents which are in charge of the two tasks. In this problem, a state represents a conjunction of the particular state of each task, and the available resources. The resources may be constrained by the amount that may be used simultaneously (local constraint), and in total (global constraint).

When executing an action a (an action here is a resource allocation) in state s, the specific states of the tasks change stochastically, and the remaining resources are determined as the resources available in s, minus the resources used by action a, if the resource is consumable. In our example, a brush is a non-consumable resource, while the detergent is a consumable resource.



(a) An example of resource allocation.



(b) task transition graph.

Fig. 2. An example of resource allocation (a) and its task transition graph (b).

As shown on Figure 2(b), the system is in a state s with a set of tasks Ta to realize, and a set Res_c of consumable resource available. A possible action, as illustrated on Figure 2(b), in this state may be to allocate one unit of detergent to task ta_1, and one brush to task ta_2. The state of the system changes stochastically, as each task's state does. For example, the floor may be clean or not with a certain probability, after having allocated the brush to clean it. In this example, the state of the tasks may change, for example, in n new possible states. For all these n possible state transitions after s, the consumable resources available (Res_c) are $Res_c \setminus res(a)$ (Res_c minus $res(a)$), where $res(a)$ is the consumable resources used by action a.

3. Markov Decision Processes (MDPs): A Slight Overview

There is an extremely large body of research studying MDPs, and the basic algorithmic techniques are presented in some detail in Section 3.1. The most commonly used formulation of MDPs assumes full observability and stationarity, and uses as its optimality criterion the maximization of expected total reward over a finite horizon, maximization of expected total discounted reward over an infinite horizon, or minimization of the expected cost to go. MDPs were introduced by Ref. 12 and have been studied in depth in the fields of decision analysis and Operation Research, including the seminal work of Ref. 13. During the recent decades, they have been extensively studied in many fields including Operation Research and Management,[1,14] Control[15] and Machine Learning.[2,3] MDPs are very suitable to model a stochastic environment where the outcome of an agent's action is probabilistic and the environment is modified by some unpredictable exogenous events. For example, suppose that a detergent resource has been allocated to clean the floor. After executing this action, the floor may be cleaned or not. Part of the MDP process for this scenario is illustrated in Figure 3.

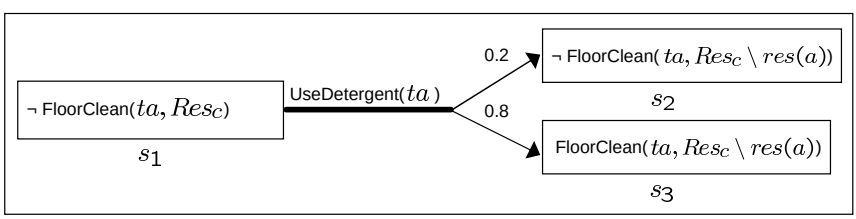


Fig. 3. A fragment of an MDP.

A Markov decision process is 4-tuple $(S, A, P.(.|.), R(.))$, where

- S is a finite set of states,
- A is a finite set of action, and $A(s) \subseteq A$ is the finite set of actions applicable in state $s \in S$.
- $P_a(s'|s) = Pr(s_{t+1} = s'|s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time $t + 1$.

- $R(s')$ is the immediate reward received after transition to state s' from state s with transition probability $P_a(s'|s)$.

A discount factor γ, a real number between 0 and 1, complete these tuples and it describes the preference of an agent for current rewards over future rewards. In MDPs, the state s' that results from an action a is not predictable but is observable, providing feedback for the selection of the next action a'. As a result, a solution of an MDP is not an action sequence, but a function π mapping state s into actions $a \in A(s)$. Such a function is called a *policy*. These policies are defined to be applicable no matter what state (or distribution over states) one finds oneself in — action choices are defined for every possible state or history.

3.1. *Dynamic programming approaches*

Suppose that an MDP is given with a state space S, action space A, a transition matrix $P_a(s'|s)$ for each action a, a reward function r. The main problem is to find the policy that maximizes the expected total reward for the planning horizon. Bellman's *principle of optimality*[12] forms the basis of the stochastic dynamic programming algorithms used to solve MDPs. In particular, the optimal value of a state is the immediate reward for that state plus the expected discounted value of the next state transition probability, assuming that the agent chooses the optimal action. That is, the value of a state when its expected reward is maximized, is given by:

$$V(s) = R(s) + \max_{a \in A(s)} \gamma \sum_{s' \in S} P_a(s'|s) V(s') \,. \tag{1}$$

A concept that is often useful is that of a Q-function (or Q-value). Intuitively, it denotes the expected value of performing action a at state s.[16] Given an arbitrary value function V, $Q(a, s)$ is defined as

$$Q(a, s) = R(s) + \gamma \sum_{s' \in S} P_a(s|s') \max_{a' \in A(s')} Q(a', s') \tag{2}$$

where $V(s') = \max_{a' \in A(s')} Q(a', s')$.

The V (or Q-value) functions are usually computed using dynamic programming (DP). DP is said to be an *implicit-enumeration* approach because it finds an optimal solution, to a given problem, without evaluating all possible solutions. Once an optimal solution for a state is found, Bellman's principle of optimality allows us to infer that an optimal solution that reaches this state must include the solution that is optimal for this state. Using Bellman's principle of optimality to avoid enumerating all possible solutions is sometimes called *pruning by dominance*. In particular, the standards dynamic programming algorithms to solve an MDP are *value iteration* and *policy iteration*. We now detail these two iterations.

3.1.1. *Policy and value iteration*

General Policy iteration proceeds as follows: at each iteration consider changing the action at each state while keeping the actions for all the other states fixed to the current policy. Some such single-state action changes will improve upon the current policy. Different variants of policy iteration differ in which single-action improvement they adopt at each step. Usually, we also alternate the following two steps, beginning from some initial policy π_0:

- *Policy evaluation:* given a policy π_i, calculate $V_i = V^{\pi_i}$, that is the value of each state if π_i is adopted;
- *Policy improvement:* calculate a new Policy π_{i+1} according to $\pi_{i+1} \leftarrow \text{argmax}_{a \in A(s)} \sum_{s'} P_a(s'|s) V^{\pi_i}(s')$

This algorithm terminates when the policy improvement step yields no change in the utilities.

Equation (1) forms the basis of the value iteration algorithm for solving MDPs. In value iteration, called also backward induction, π is not used, it is calculated just whenever it needed. To iterate values, we reformulate equation Equation (1) as the following iteration

$$V(s) \longleftarrow R(s) + \max_{a \in A(s)} \gamma \sum_{s' \in S} P_a(s'|s) V(s') \,.$$

Thus, value iteration propagates utilities from a state s to its neighbor states s' iteratively. Indeed, value iteration can be viewed as *propagating information* through the state space by means of local updates. Value iteration terminates when the value change between two iterations is bounded by an indifference constant ϵ. Notice that the model may also include a time horizon which might be finite or infinite.

Real-Time search which has been proven more efficient than value iteration is now introduced.

3.1.2. *Real-time search*

Russel and Norvig[17] distinguish two types of search. Firstly, an *offline search* algorithm computes a complete solution before setting foot in the real world, and then executes the solution without recourse to their percept. In contrast, an *online search* agent operates by interleaving computation and action: first it takes an action, then it observes the environment and computes the next action. Online search is a good idea in dynamic domains when there is a penalty for computing too long. Online search is an even better idea for stochastic domains. In general, an offline search would have to come up with an exponentially large contingency plan that considers all possible happenings, while an online search needs only consider what actually does happens.

An optimal policy can be found using an offline dynamic programming algorithm such as policy iteration or value iteration. But a disadvantage of dynamic

programming is that it evaluates the entire state space. In effect, it finds a policy for every possible starting state. By contrast, heuristic search algorithms solve a problem for a particular starting state and use an admissible heuristic to focus the search, and remove from consideration regions of the state space that cannot be reached from the start state by an optimal solution. For problems with large state spaces, heuristic search has an advantage over dynamic programming because it can find an optimal solution for a start state without evaluating the entire state space.

This advantage is well-known for problems that can be solved by A*.[18] In fact, an important theorem about the behavior of A* is that (under certain conditions) it evaluates the minimal number of states among all algorithms that find an optimal solution.[19] The A* algorithm lies the foundation of the online Real-time Dynamic Programming (RTDP), Labelled RTDP (LRTDP), Bounded RTDP (BRTDP), and Focused RTDP (FRTDP) algorithms. These algorithms reflecting real-time dynamic programming are now presented.

RTDP. In the value iteration algorithm, the updates are performed over all states in parallel. On the other hand, in the Real-Time Dynamic Programming (RTDP) algorithm,[4] the updates are performed on the states visited by a greedy search guided by the value function V.

Real-Time Dynamic Programming (RTDP) is a probabilistic version of LRTA*,[20] which in turn has emerged from A*.[18] A good advantage of RTDP, just like all MDP algorithms, is that it is an anytime algorithm. This algorithm can also be viewed as a greedy version of the dynamic programming algorithms for solving MDPs. Indeed, at each state s, a local choice is made based on the selection of the optimal action a based on a selection function $h(s)$ to search in the state space S. Notice that $h(s)$ should be an admissible heuristic — that is, the value given by the heuristic has to overestimate (or underestimate) the optimal value when the objective function is maximized (or minimized) — heuristic which defines an initial value for state s'. Furthermore, RTDP is an on-line algorithm because it interleaves planning and execution.

LRTDP. Bonnet[5] proposed LRTDP (Algorithm 1) as an improvement to RTDP.[4] LRTDP is a simple dynamic programming algorithm that involves a sequence of trial runs, each starting in the initial state s_0 and ending in a goal or a *solved* state. Each LRTDP trial (Line 6 to 11) is the result of simulating the policy π, through the PICK-NEXT-STATE(a) function, while updating the values $V(s)$ using a Bellman backup (Eq. (1)) over the states s that are visited.

It has been proven that LRTDP, given an admissible initial heuristic on the value of states cannot be trapped in loops, and eventually yields optimal values.[5] The convergence is accomplished by means of a labeling procedure called CHECK-SOLVED(s, ϵ) (Line 14 of the algorithm). This procedure tries to label as solved each traversed state in the current trajectory. When the initial state is labelled as

Algorithm 1 The LRTDP algorithm.[5]

```
1: Function LRTDP(S)
2: returns a value function V
3: repeat
4:    s ← s_0
5:    visited ← null
6:    repeat
7:       visited.push(s)
8:       V(s) ← R(s) + max_{a∈A(s)} γ Σ_{s'∈S} P_a(s'|s)V(s') {where V(s') = h(s') when s' is
         not yet visited}
9:       a = s.GREEDY-ACTION()
10:      s ← s.PICK-NEXT-STATE(a)
11:   until s is a goal
12:   while visited ≠ null do
13:      s ← visited.pop()
14:      if ¬ CHECK-SOLVED(s, ε) then
15:         break
16:      end if
17:   end while
18: until s_0 is solved
19: return V
```

solved, the algorithm has converged. LRTDP uses only an upper bound ($h(s)$) to guide the search.

Using both an upper bound and a lower bound permits to prune the action space and to guide the search more effectively. The next sections describe BRTDP and FRTDP which both use an upper bound and a lower bound.

BRTDP. The pseudocode for Bounded RTDP (BRTDP)[6] is given in Algorithm 2. BRTDP has many differences from RTDP: The first is when a policy is requested for BRTDP (before or after convergence), it is returned based on the lower bound L. The second difference is that L helps guide exploration in simulation, as computed in Lines 11 to 16 of the algorithm. In particular, when trajectories are sampled in simulation, the outcome distribution is biased to prefer transitions to states with a large gap: $U(x') - L(x')$, where $U(x')$ is the upper bound of state x', and $L(x')$ is the lower bound of state x'. Furthermore, BRTDP maintains a list of states on the current trajectory, and when the trajectory terminates, it does backups in reverse order along the stored trajectory (Lines 18 to 22 of the BRTDP function). Finally, like LRTDP, simulated trajectories terminate when they reach a state that has a "well-known" value, rather than when they reach the goal. The adaptive criterion τ is a constant > 1.

Algorithm 2 The BRTDP algorithm.[6]

Function BRTDP(S)
returns a value function V
while $U(s_0) - L(s_0) > \epsilon$ **do**
  $x \leftarrow s_0$
  $traj \leftarrow$ EMPTYSTACK
  while true **do**
    $traj$.PUSH(x)
    $U(x) \leftarrow R(x) + \gamma \sum_{x' \in S} P_a(x'|x)U(x')$ {where $U(x) \leftarrow h_U(x)$ when x is not yet visited}
    $a \leftarrow \arg\max_{a \in A(s)} Q_L(x, a)$
    $L(x) \leftarrow R(x) + \gamma \sum_{x' \in S} P_a(x'|x)L(x')$ {where $L(x) \leftarrow h_L(x)$ when x is not yet visited}
    $\forall x', b(x') \leftarrow P_a(x'|x)(U(x') - L(x'))$
    $B \leftarrow \sum_{x' \in S} b(x')$
    if $B < (U(s_0) - L(s_0))/\tau$ **then**
      break
    end if
    $x \leftarrow$ sample from distribution $b(x')/B$
  end while
  while $traj \neq null$ **do**
    $x \leftarrow traj$.POP()
    $U(x) \leftarrow R(x) + \gamma \sum_{x' \in S} P_a(x'|x)U(x')$
    $L(x) \leftarrow R(x) + \gamma \sum_{x' \in S} P_a(x'|x)L(x')$
  end while
end while
return V

FRTDP. Focused RTDP is also an RTDP based algorithm proposed by Ref. 7. As in RTDP, FRTDP's execution consists in trials that begin in a given initial state s_0 and then explore reachable states of the state space, selecting actions according to an upper bound. Once a final state is reached, it performs Bellman updates on the way back to s_0. FRTDP uses a priority value for selecting actions outcomes and detecting trial termination. The lower bound ($L(s)$) is used to establish the policy by contributing in the priority calculation of states to expand on the fringe of the search tree. Furthermore, in this algorithm, trial termination detection has been modified and improved from RTDP by adding an adaptive maximum depth D in the search tree in order to avoid over-committing to long trials early on. More

Table 1. General comparison between RTDP, LRTDP, BRTDP and FRTDP.

	RTDP	LRTDP	FRTDP	BRTDP
Optimality	Infinite	Yes	Yes	Yes
Upper bound	Yes	Yes	Yes	Yes
Lower bound	No	No	Yes	Yes
Speed of convergence	+	++	+++	+++

Note: Here, "+" should be read as follows[6,7]: LRTDP converges faster than RTDP; FRTDP converges faster than LRTDP; FRTDP and BRTDP are comparable in terms of speed of convergence.

precisely, the maximum depth D is updated by $k_D \times D$ (where k_D is the increasing ratio) each time the trial is not useful enough. This usefulness is represented by δW where δ measures how much the update changed the upper bound value of s and W the expected amount of time the current policy spends in s, adding up all possible paths from s_0 to s.

Table 1 presents a broad comparison of RTDP, LRTDP, BRTDP, and FRTDP. All these approaches converge to an optimal policy, however, RTDP does it in an infinite amount of time. All these algorithms implement a lower bound, but only BRTDP and FRTDP implement an upper bound. BRTDP and FRTDP are also the ones that converge the fastest to the optimal solution.[6,7]

One should however emphasized that the choice of the initial value function (heuristic) is essential for the convergence of RTDP type solutions.

4. Resource Allocation using MDPs

An MDP in the context of our resource allocation problem with limited resources is defined as a tuple $\langle Res, Ta, S, A, P, W, R\rangle$, where:

- $Res = \langle res_1, \ldots, res_{|Res|}\rangle$ is a finite set of resource types available for a planning process. These resources might be discrete (machine, processor, etc.) or continuously divisible (power, detergent, paint, etc.). Each resource type may have a local resource constraint L_{res} on the number that may be used in a single step, and a global resource constraint G_{res} on the number that may be used in total. The global constraint only applies for consumable resource types (Res_c) and the local constraints always apply to consumable and non-consumable resource types.
- Ta is a finite set of tasks with $ta \in Ta$ to be accomplished.
- S is a finite set of states with $s \in S$. A state s is a tuple $\langle Ta, \langle res_1, ..., res_{|Res_c|}\rangle\rangle$, which is the characteristic of each unaccomplished task $ta \in Ta$ in the environment, and the available consumable resources. s_{ta} is the specific state of task ta. Furthermore, S contains a non empty set $s_g \subseteq S$ of goal states. A goal state is a sink state where an agent stays forever.
- A is a finite set of actions (or assignments). The actions $a \in A(s)$ applicable in a state are the combination of all resource assignments that may be executed, according to the state s. In particular, a is simply an allocation of resources

to the current tasks, and a_{ta} is the resource allocation to task ta. The possible allocations are limited by L_{res} and G_{res}.
- Transition probabilities $P_a(s'|s)$ for $s \in S$ and $a \in A(s)$.
- $W = [w_{ta}]$ is the relative weight (criticality) of each task.
- State rewards $R = [r_s] : \sum_{ta \in Ta} r_{s_{ta}} \leftarrow \Re_{s_{ta}} \times w_{ta}$. The relative reward of the state of a task $r_{s_{ta}}$ is the product of a real number $\Re_{s_{ta}}$ by the weight factor w_{ta}. For our problem, a reward of $1 \times w_{ta}$ is given when the state of a task (s_{ta}) is in an achieved state, and 0 in all other cases.
- A discount (preference) factor γ, which is a real number between 0 and 1.

A solution of an MDP is a policy π mapping state s into actions $a \in A(s)$. In particular, $\pi_{ta}(s)$ is the action that should be executed on task ta, considering the global state s. In this case, an optimal policy is one that maximizes the expected total reward for accomplishing all tasks. The optimal value of a state, $V(s)$, is given by previous equation (1) in which the remaining consumable resources in state s' are $Res_c \setminus res(a)$ and where $res(a)$ are the consumable resources used by action a. Similarly, one may compute the Q-Values $Q(a, s)$ of each state action pair using the equation (2).

The policy is subjected to the local resource constraints $res(\pi(s)) \leq L_{res}$; $\forall\ s \in S$, and $\forall\ res \in Res$. The global constraint is defined according to all system trajectories $tra \in TRA$. A system trajectory tra is a possible sequence of state-action pairs, until a goal state is reached under the optimal policy π. For example, state s is entered, which may transit to s' or to s'', according to action a. The two possible system trajectories are $\langle (s, a), (s') \rangle$ and $\langle (s, a), (s'') \rangle$. If we express $res(tra)$ as a function which returns the resources used by trajectory tra, then the global resource constraint is $res(tra) \leq G_{res}; \forall\ tra \in TRA$, and $\forall\ res \in Res_c$. Furthermore, the model is Markovian and consequently the history need not be considered in the state space.

Now we investigate the distributed Q-values approach.

4.1. *Distributed Q-values for resource allocation*

4.1.1. *The distributed Q-values approach*

In the Case 2 (as depicted in Figure 1) , resources are already distributed to agents and actions of ones may influence the reward obtained by others. This is for instance the case when agents are in charge of tasks which are not complectly independent and they have precedence constraint between them as for instance: "Task1 should be executed before Task2" or "Task1 facilitates Task3" or "It would be better to do Task2 and Task5 in parallel", etc. To use efficiently resources in this case, we use distributed Q-values (adapted from Ref. 21). The primary assumption underlying distributed Q-values is that the overall reward function R can be additively decomposed into separate rewards R_i for each distinct agent $i \in Ag$, where $|Ag|$ is the number of agents. That is, $R = \sum_{i \in Ag} R_i$. In this case, it requires each agent

to compute a value, from its perspective, for every action. To coordinate with each other, each agent i reports its action values $Q_i(a_i, s_i)$ for each state $s_i \in S_i$ to an arbitrator at each planning iteration. The arbitrator then chooses an action maximizing the sum of the agent Q-values for each global state $s \in S$. The next time state s is then updated, an agent i considers its Q-value as its respective contribution to the global maximal Q-value. That is, $Q_i(a_i, s_i)$ is the value of a state such that it maximizes $\max_{a \in A(s)} \sum_{i \in Ag} Q_i(a_i, s_i)$.

The fact that the agents use a determined Q-value as the value of a state is an extension of the Sarsa on-policy algorithm[22] to distributed Q-values. In this way, an ideal compromise can be found for the agents to reach a global optimum. Indeed, rather than allowing each agent to choose the successor action, each agent i uses the action a'_i chosen by the arbitrator in the successor state s'_i:

$$Q_i(a_i, s_i) = R_i(s_i) + \gamma \sum_{s'_i \in S_i} P_{a_i}(s'_i|s_i) Q_i(a'_i, s'_i) \tag{3}$$

where the remaining consumable resources in state s'_i are $Res_{c_i} \setminus res_i(a_i)$ for a resource allocation problem.

We can then apply an optimal Bellman backup in a state as in Algorithm 3. In Line 5 of the DISTQ-BACKUP function, each agent managing a task computes its respective Q-value. Here, $Q^\star_i(a'_i, s')$ determines the optimal Q-value of agent i in state s'. An agent i uses as the value of a possible state transition s' its Q-value which is part of the maximal global Q-value for state s'. In brief, for each visited state $s \in S$, each agent computes its respective Q-values with respect to the global state s. So the state space is the joint state space of all agents.

The arbitrator functionalities are depicted in Lines 8 to 20 of Algorithm 3. The global Q-value is the sum of the Q-values produced by each agent managing each task as shown in Line 11, considering the global action a. In this case, when an action of an agent i cannot be executed simultaneously with an action of another agent i', the global action is simply discarded from the action space $A(s)$. Line 14 simply allocates the current value with respect to the highest global Q-value, as in a standard Bellman backup. Then, the optimal policy and Q-value of each agent is updated in Lines 16 and 17 to the sub-actions a_i and specific Q-values $Q_i(a_i, s)$ of each agent for action a.

The behavior of DISTQ-BACKUP is now discussed and we start it by proving the optimality of DISTQ-BACKUP. This proof requires to demonstrate that the DISTQ-BACKUP function outputs the same optimal value as a standard Bellman backup.

Lemma 4.1. *A state for DISTQ-BACKUP is updated in the same manner as for a standard Bellman backup.*

Proof. The following equation is used in DISTQ-BACKUP to compute a Q-value:

$$Q(a, s) = \sum_{i \in Ag} R_i(s) + \max_{a_i \in A_i(s)} \gamma \sum_{s'_i \in S'_i} P_{a_i}(s'_i|s) Q^\star_i(a'_i, s')\,. \tag{4}$$

Algorithm 3 The distributed Q-values Bellman Backup algorithm.[23]

Function DISTQ-BACKUP(s)
$V(s) \leftarrow 0$
for all $i \in Ag$ **do**
  for all $a_i \in A_i(s)$ **do**
    $Q_i(a_i, s) \leftarrow R_i(s) + \gamma \sum_{s'_i \in S_i} P_{a_i}(s'_i|s) Q^\star_i(a'_i, s')$ {where $Q^\star_i(a'_i, s') = h_i(s')$ when s' is not yet visited, and s' has $Res_{c_i} \setminus res_i(a_i)$ remaining consumable resources for each agent i}
  end for
end for
for all $a \in A(s)$ **do**
  $Q(a, s) \leftarrow \sum_{i \in Ag} Q_i(a_i, s)$
  if $Q(a, s) > V(s)$ **then**
    $V(s) \leftarrow Q(a, s)$
    for all $i \in Ag$ **do**
      $\pi_i(s) \leftarrow a_i$
      $Q^\star_i(a_i, s) \leftarrow Q_i(a_i, s)$
    end for
  end if
end for

Since the reward can be additively decomposed for each task, Eq. (4) may be rewritten as:

$$Q(a, s) = R(s) + \sum_{i \in Ag} \max_{a_i \in A_i(s)} \gamma \sum_{s'_i \in S'_i} P_{a_i}(s'_i|s) Q^\star_i(a'_i, s') \,. \tag{5}$$

Since $Q(a, s) = \sum_{i \in Ag} Q_i(a, s)$ when the transition probability of each task considers the actions performed on other tasks, Eq. (5) may be rewritten as:

$$Q(a, s) = R(s) + \max_{a \in A(s)} \gamma \sum_{s' \in S'} P_a(s'|s) Q(a', s') \tag{6}$$

where $Q(a', s')$ is the maximal Q-value for state s'. Indeed, since the arbitrator determines the maximal Q-value for a state, $Q(a', s') = V(s')$. Since Eq. (6) is the same as a Bellman backup, and $Q(a', s')$ is the same as $V(s)$, a Q-value is updated in the same manner in DISTQ-BACKUP as for a standard Bellman backup. □

4.1.2. *Complexity of distributed Q-values*

Some of the gain in complexity to use distributed Q-values lies in the $\sum_{s'_i \in S_i} P_{a_i}(s'_i|s)$ part of the equation. An agent considers as a possible state transition only the possible states of the set of tasks it manages. Since the number of states is exponential with the number of tasks, using distributed Q-values should

reduce the planning time significantly. Furthermore, the action space of the agents takes into account only their available resources which is much less complex than a standard action space, which is the combination of all possible resource allocation in a state for all agents.

More formally, a standard "multiagent" Bellman backup has a complexity of $O(|A| \times |S_{Ag}|)$, where $|S_{Ag}|$ is the number of joint states for all agents excluding the resources, and $|A|$ is the number of joint actions. On the other hand, the distributed Q-values Bellman backup has a complexity of $O((|Ag| \times |A_i| \times |S_i|) + (|A| \times |Ag|))$, where $|S_i|$ is the highest number of states for an agent i, excluding the resources, and $|A_i|$ is the highest number of actions for an agent i. Since $|S_{Ag}|$ is combinatorial with the number of tasks, so $|S_i| \ll |S|$. Also, $|A|$ is combinatorial with the number of resource types. If the resources are already shared among the agents, the number of resource types for each agent will usually be lower than the set of all available resource types for all agents. In these circumstances, $|A_i| \ll |A|$. In a standard Bellman backup, $|A|$ is multiplied by $|S_{Ag}|$, which is much more complex than multiplying $|A|$ by $|Ag|$ with the distributed Q-values Bellman backup. Thus, the distributed Q-values Bellman backup is much less complex than a standard Bellman backup.

In fact even with the cost of communication induced, the distributed Q-values remains much better than the global Q based on the complete standard Bellman backup, as shown by Ref. 21.

We now present our tight bounds for improving the existing real-time dynamic programming algorithms.

4.2. *A new bounded RTDP approach*

4.2.1. *Singh and Cohn bounds*

In the third case when all resources are available to all agents, then they can perform tasks simultaneously. We then need a modified value iteration algorithm for dynamically merging MDPs. The approach proposed by Singh and Cohn[10] is suitable for this. These authors defined lower and upper bounds to prune the action space in the context of dynamically merging multiple MDPs. In their approach, a value function is computed for all tasks to realize, using a standard RTDP approach. Then, using these *task*-value functions, a lower bound h_L, and upper bound h_U are defined as follows :

$$h_L(s) = \max_{ta \in Ta} V_{ta}(s_{ta}), \text{ and } h_U(s) = \sum_{ta \in Ta} V_{ta}(s_{ta}) .$$

Indeed, for any composite state, the sum of the optimal values of the component states is an upper bound to the optimal value of the composite states, i.e., $V(s_{ta}) \leq \sum_{ta \in Ta} V_{ta}(s_{ta})$. Similarly, for any composite state, the maximum of the optimal values of the component states is a lower bound to the optimal value of the composite states, i.e., $V(s_{ta}) \geq \max_{ta \in Ta} V_{ta}(s_{ta})$.

The admissibility of these bounds has been proven by Singh and Cohn. Thus, the upper bound always overestimates the optimal value of each state, while the lower bound always underestimates the optimal value of each state. However, these bounds are somehow large and we propose to tight them in the next two sections.

Notice that for readability, we have named the upper bound and the lower bound of Singh and Cohn, respectively: SinghU and SinghL. Furthermore, the bounds defined by Singh and Cohn that we have implemented using BOUNDED-RTDP is named SINGH-RTDP.

4.2.2. *A tight upper bound*

The SinghU bound as proposed by Singh and Cohn[10] includes actions which may not be possible to execute because of resource constraints and which outputs too high values for the upper bound. To consider only possible actions (i.e, $a \in S(s)$), our upper bound,[23] named maxU is introduced:

$$h_U(s) = \max_{a \in A(s)} \sum_{ta \in Ta} Q_{ta}(a_{ta}, s_{ta}) \tag{7}$$

where $Q_{ta}(a_{ta}, s_{ta})$ is the Q-value of task ta for state s_{ta}, and action a_{ta} computed using a standard LRTDP approach.

Theorem 4.1. The upper bound defined by Eq. (7) is admissible.

Proof. The local resource constraints are satisfied because the upper bound is computed using all global possible actions a. However, $h_U(s)$ (as reflected by equation (7)) still overestimates $V^\star(s)$ because the global resource constraint is not enforced. Indeed, each task may use all consumable resources for its own purpose. Doing this produces a higher value for each task, than the one obtained when planning for all tasks globally with the shared limited resources. □

Computing our maxU bound in a state has a complexity of $O(|A| \times |Ta|)$. A standard Bellman backup has a complexity of $O(|A| \times |S_{Ta}|)$, where S_{Ta} is the number of joint states for all the agents excluding the resources. Since $|A| \times |Ta| \ll |A| \times |S_{Ta}|$, the computation time to determine the upper bound of a state, which is done one time for each visited state, is much less than the computation time required to compute a standard Bellman backup for a state, which is usually done many times for each visited state. Thus, the computation time induced by our upper bound is negligible.

4.2.3. *A tight lower bound*

Our tight lower bound is formulated by allocating the resources a priori among the tasks.[23] When each task has its own set of resources, each task may be solved independently. The allocation a priori of all the resources is made using *marginal*

revenue, which is a highly used concept in microeconomics,[11] and has recently been used for coordination of a decentralized MDP.[24] In brief, marginal revenue is the extra revenue that an additional unit of product will bring to a firm. Thus, for a stochastic resource allocation problem, the marginal revenue of a resource is the additional expected value it involves. The marginal revenue of a resource res; for a task ta in a state s_{ta} is defined as follows:

$$mr_{ta}(s_{ta}) \leftarrow V_{ta}(s_{ta}(Res)) - V_{ta}(s_{ta}(Res \setminus res)) . \tag{8}$$

To determine a lower bound, here how we proceed. At the beginning, the REVENUE-BOUND function is called (Algorithm 4) with the set of tasks to execute and the set of available resources. Then, the ASSIGN-RESOURCE function (Algorithm 5) assigns each resource type to a task using the concept of marginal revenue. Finally, with the resource being shared, we compute, using a LRTDP algorithm, the value function of each task with its designated set of resource. A lower bound for a state can then be obtained by summing the respective value functions.

Algorithm 4 The marginal revenue lower bound algorithm.[23]

1: **Function** REVENUE-BOUND(s)
2: **returns** a lower bound Low_{Ta}
3: **for all** $ta \in Ta$ **do**
4: $V_{ta} \leftarrow$LRTDP(S_{ta}) {Same value functions as used by the upper bound.}
5: $value_{ta} \leftarrow 0$
6: **end for**
7: $Res_{Ta} \leftarrow$ASSIGN-RESOURCES($S, value_{Ta}$)
8: **for all** $ta \in Ta$ **do**
9: $Low_{ta} \leftarrow$LRTDP(S_{ta})
10: **end for**
11: **return** Low_{Ta}

We now describe the Algorithm 4, which uses the concept of marginal revenue of a resource to allocate the resources a priori among the tasks, thus enabling to define the lower bound value of a state. In Line 4 of the algorithm, a value function is computed for all tasks in the environment using a standard LRTDP[5] approach. This value function, which is also used for the upper bound, is computed considering that each task may use all available resources. The Line 5 initializes the $value_{ta}$ variable. This variable is the estimated value of each task $ta \in Ta$. In the beginning of the algorithm, no resources are allocated to a specific task, thus the $value_{ta}$ variable is initialized to 0 for all $ta \in Ta$.

Then, the execution shifts to the ASSIGN-RESOURCES function (Algorithm 5). In Line 5, a resource type res (consumable or non-consumable) is selected to be allocated. Here, a domain expert may separate all available resources in many types or

Algorithm 5 The assign resource algorithm.[23]

1: **Function** ASSIGN-RESOURCES($S, value_{Ta}$)
2: **returns** a lower bound Low_{Ta}
3: $s \leftarrow s_0$
4: **repeat**
5: $res \leftarrow$ Select a resource type $res \in Res$
6: **for all** $ta \in Ta$ **do**
7: **if** res is consumable **then**
8: $mr_{ta}(s_{ta}) \leftarrow V_{ta}(s_{ta}) - V_{ta}(s_{ta}(Res \setminus res))$
9: **else**
10: $mr_{ta}(s_{ta}) \leftarrow 0$
11: **repeat**
12: $mr_{ta}(s_{ta}) \leftarrow mr_{ta}(s_{ta}) + V_{ta}(s_{ta}) - \max_{(a_{ta} \in A(s_{ta}) | res \notin a_{ta})} Q_{ta}(a_{ta}, s_{ta})$
13: $Res_{c_{ta}} \leftarrow Res_{c_{ta}} \setminus res(\pi(s_{ta}))$
14: $s_{ta} \leftarrow s_{ta}$.PICK-NEXT-STATE($Res_{c_{ta}}$)
15: **until** s_{ta} is a goal
16: $s \leftarrow s_0$
17: **end if**
18: $mrrv_{ta}(s_{ta}) \leftarrow mr_{ta}(s_{ta}) \times \frac{V_{ta}(s_{ta}) - value_{ta}}{R(s_{g_{ta}})}$
19: **end for**
20: $ta \leftarrow$ Task $ta \in Ta$ which maximizes $mrrv_{ta}(s_{ta})$
21: $Res_{ta} \leftarrow Res_{ta} \bigcup \{res\}$
22: $temp \leftarrow \emptyset$
23: **if** res is consumable **then**
24: $temp \leftarrow res$
25: **end if**
26: $value_{ta} \leftarrow value_{ta} + ((V_{ta}(s_{ta}) - value_{ta}) \times \frac{\max_{a_{ta} \in A(s_{ta}, res)} Q_{ta}(a_{ta}, s_{ta}(temp))}{V_{ta}(s_{ta})})$
27: **until** all resource types $res \in Res$ are assigned
28: **return** Res_{Ta}

parts to be allocated. The resources are allocated in the order of their specialization. In other words, the more a resource is efficient on a small group of tasks, the earlier it is allocated. Allocating the resources in this order improves the quality of the resulting lower bound. The Line 8 computes the marginal revenue of a consumable resource res for each task $ta \in Ta$.

The approach adopted here is to sum the difference between the real value of a state to the maximal Q-value of this state if resource res cannot be used for all states in a trajectory given by the policy of task ta. This heuristic proved to obtain good results, but other ones may be tried, for example Monte-Carlo simulation. In Line 18, the marginal revenue is updated in function of the resources already allocated

to each task. $R(s_{g_{ta}})$ is the reward to realize task ta. Thus, $\frac{V_{ta}(s_{ta})-value_{ta}}{R(s_{g_{ta}})}$ is the residual expected value that remains to be achieved, knowing current allocation to task ta, and normalized by the reward of realizing the tasks. The marginal revenue is multiplied by this term to indicate that, the higher the task's residual value, the higher its marginal revenue.

Then, a task ta is selected in Line 20 with the highest marginal revenue, adjusted with residual value. In Line 21, the resource type res is allocated to the group of resources Res_{ta} of task ta. Afterwards, Line 26 recomputes $value_{ta}$. The first part of the equation to compute $value_{ta}$ represents the expected residual value for task ta. This term is multiplied by

$$\frac{\max\limits_{a_{ta}\in A(s_{ta})} Q_{ta}(a_{ta}, s_{ta}(res))}{V_{ta}(s_{ta})},$$

which is the ratio of the efficiency of resource type res. In other words, $value_{ta}$ is assigned to $value_{ta}$ + (*the residual value* × *the value ratio of resource type* res). For a consumable resource, the Q-value considers only resource res in the state space, while for a non-consumable resource, no resources are available.

All consumable and non-consumable resource types are allocated in this manner until Res is empty. When all resources are allocated, the lower bound components Low_{ta} of each task are computed in Line 9 of the REVENUE-BOUND function. When the global solution is computed, the lower bound is as follow:

$$h_L(s) = \max(\max_{ta\in Ta} V_{ta}(s_{ta}),\ \sum_{ta\in Ta} Low_{ta}(s_{ta}))\,. \tag{9}$$

We use the maximum of the maximal value functions and the sum of the lower bound components Low_{ta}, thus $h_L(s) \geq$ SinghL. We call this new bound RbL.

The main difference of complexity between SinghL and RbL comes from $\sum_{ta\in Ta} Low_{ta}(s_{ta})$ which is computed via REVENUE-BOUND via Line 9 where a value for each task has to be computed with the shared resources. However, since the resources are shared, the state and action space is greatly reduced for each task, reducing greatly the calculus compared to the value functions computed in Line 4 which is done for both SinghL and REVENUE-BOUND.

Theorem 4.2. The lower bound of Eq. (9) is admissible.

Proof. $Low_{ta}(s_{ta})$ is computed with the resources being shared. Summing the $Low_{ta}(s_{ta})$ value functions for each $ta \in Ta$ does not violates the local and global resource constraints. Indeed, as the resources are shared, the tasks cannot overuse them. Thus, $h_L(s)$ is a realizable policy, and consequently an admissible lower bound. □

5. Discussion and Experiments

In our case, the experimental test have performed in the domain of a naval platform which must counter incoming missiles (i.e. tasks) by using its resources (i.e.,

weapons, movements). For the experiments, 100 random resource allocation problems were generated for each approach, and possible number of tasks. In our problem, $|S_{ta}| = 4$, thus each task can be in four distinct states, which are: (1) the incoming missile is searching the platform; (2) the incoming missile is locked on the platform; (3) the incoming missile is countered; and (4) the incoming missile hit the platform. There are two types of states, firstly, states where actions modify the transition probabilities; and then, there are goal states. The state transitions are all stochastic because when a missile is in a given state, it may always transit in many possible states. In particular, each resource type has a probability to counter a missile between 45% and 65% depending on the state of the task.

When a missile is not countered, it transits to another state, which may be preferred or not to the current state, where the most preferred state for a task is when it is countered. The effectiveness of each resource is modified randomly by $\pm 15\%$ at the start of a scenario. There are also local and global resource constraints on the amount that may be used. For the local constraints, at most 1 resource of each type can be allocated to execute tasks in a specific state. This constraint is also present on a real naval platform because of sensor and launcher constraints and engagement policies. Furthermore, for consumable resources, the total amount of available consumable resource is between 1 and 2 for each type. The global constraint is generated randomly at the start of a scenario for each consumable resource type. The number of resource types has been fixed to 5, where there are 3 consumable resource types and 2 nonconsumable resources types.

For this problem the LRTDP, FRTDP and BRTDP approaches have been implemented using different bounds. For FRTDP, the initial length D of a trajectory is 3 and the increasing ratio (k_D) is 1.2. For BRTDP the constant τ was set to 10. We tried different variations of settings and this one provided a fast convergence. Also, as Singh and Cohn[10] proposed, we pruned the action space when $Q_U(a, s) < L(s)$ for the implementations of FRTDP and BRTDP. Lets summarize the implemented approaches here:

5.1. *Experiments for Case 2 of Figure 1*

The following approaches were implemented considering that each task possesses its own set of resources:

- LRTDP: The upper bound of MAXU has been used for LRTDP.
- R-BRTDP: The RBL and MAXU bounds have been used for BRTDP.
- R-FRTDP: The RBL and MAXU bounds have been used for FRTDP.
- DISTQ-LRTDP: The backups have been computed using the DISTQ-BACKUP function (Algorithm 3), in a LRTDP context. The upper bound used was MAXU.
- DISTQ-FRTDP: The backups has been computed using the DISTQ-BACKUP function, but in a FRTDP context. The RBL and MAXU bounds have been used.
- DISTQ-BRTDP: The backups have been computed using the DISTQ-BACKUP function, but in a FRTDP context. The RBL and MAXU bounds have been used.

Table 2. Planning time in seconds for our resource allocation problem where each task possesses its own set of resource.

$\|Ta\|$	5	6	7	8	9	10	11	12
LRTDP	36.5	614	8243	–	–	–	–	–
DISTQ-LRTDP	1.24	3.78	10.65	38.56	122.62	457	1646	7654.45
R-FRTDP	8.3	56	412	3876	–	–	–	–
DISTQ-FRTDP	0.85	2.28	5.65	11.45	25.63	74.23	258	1192
R-BRTDP	9.2	62	445	4287	–	–	–	–
DISTQ-BRTDP	0.87	2.45	6.64	11.97	29.73	76.38	285	1359

To implement distributed Q-values, we divided the set of tasks in two equal parts. The set of tasks Ta_i, managed by agent i, can be accomplished with the set of resources Res_i, while the second set of task $Ta_{i'}$, managed by agent $Ag_{i'}$, can be accomplished with the set of resources $Res_{i'}$. Res_i had one consumable resource type and one non-consumable resource type, while $Res_{i'}$ had two consumable resource types and one non-consumable resource type. When the number of tasks is odd, one more task was assigned to $Ta_{i'}$. There are constraints between the group of resources Res_i and $Res_{i'}$ such that some assignments are not possible. These constraints are managed by the arbitrator as described in Section 4.1.

Distributed Q-values permits to diminish the planning time significantly in our problem settings, as seen in Table 2. Thus, distributed Q-values seems a very efficient approach when a group of agents have to allocate resources which are only available to themselves, but the actions made by an agent may influence the reward obtained by at least another agent.

5.2. *Experiments for Case 3 of Figure 1*

We also compared the performance of LRTDP, FRTDP and BRTDP on our resource allocation problem when the resources are available to all agents. The following approaches are implemented with the resources available to all agents:

- LRTDP: The upper bound of MAXU has been used for LRTDP.
- S-BRTDP: The SINGHL and SINGHU bounds have been used for BRTDP.
- S-FRTDP: The SINGHL and SINGHU bounds have been used for FRTDP.
- R-BRTDP: The RBL and MAXU bounds have been used for BRTDP.
- R-FRTDP: The RBL and MAXU bounds have been used for FRTDP.
- L-FRTDP: The RBL and SINGHU bounds have been used for FRTDP.
- U-FRTDP: The SINGHL and MAXU bounds have been used for FRTDP.
- NPR-FRTDP: The same than R-FRTDP, but without action pruning.

As one can notice in Table 3, and in Figures 4 and 5, the efficient trajectories of the two bounded approaches BRTDP and FRTDP coupled with our tight bounds reduce the planning time significantly. Indeed, the LRTDP approach for resource allocation, which does not prune the action space, is much more complex. For in-

Table 3. Planning time in seconds of FRTDP and BRTDP for our resource allocation problem.

$\|Ta\|$	3	4	5	6	7	8
S-BRTDP	0.34	2.4	29	287	2373	–
S-FRTDP	0.3	2.1	23	202	1745	–
R-BRTDP	0.23	1.5	12.4	76	482	3987
R-FRTDP	0.21	1.4	11.2	69	450	3550
L-FRTDP	0.23	1.5	13.2	79	510	4150
U-FRTDP	0.27	1.8	18.2	180	1467	–

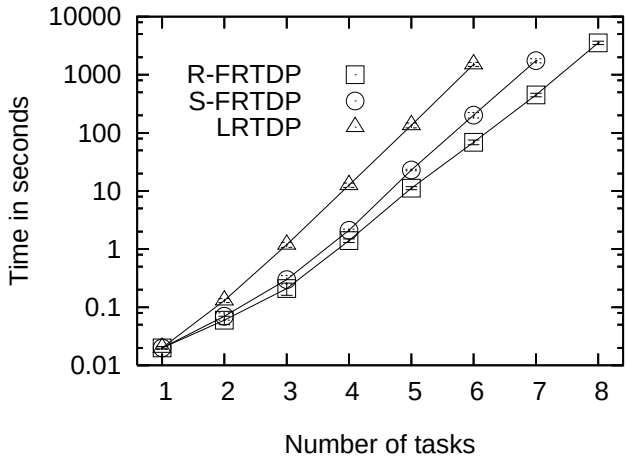


Fig. 4. Computational efficiency of S-FRTDP, R-FRTDP and LRTDP.

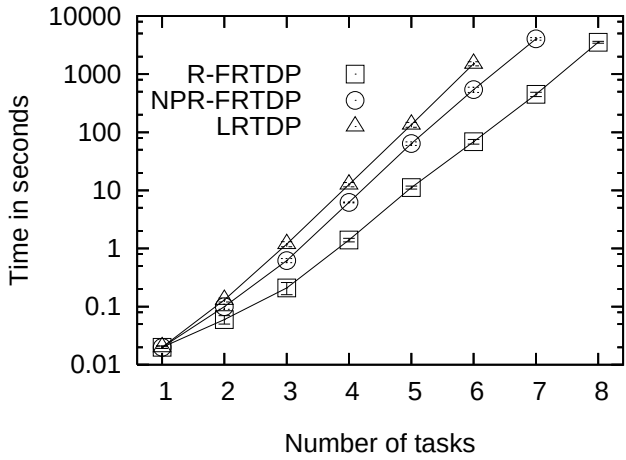


Fig. 5. Computational efficiency of R-FRTDP, NPR-FRTDP and LRTDP.

Table 4. Average number of backups and average number of actions performed for each backup made at s_0, ($|A(s_0)|$), for our resource allocation problem.

$\|Ta\|$	4	5	6
LRTDP	58 (625)	316 (3125)	1038 (15625)
S-FRTDP	34 (214)	173 (956)	533 (4276)
R-FRTDP	23 (152)	122 (644)	345 (2604)

stance, it took an average of 1512 seconds to plan for the LRTDP approach with six tasks. The S-FRTDP approach diminished the planning time by using a lower and upper bound to prune the action space and with the efficient trajectories. R-FRTDP further reduce the planning time by providing very tight initial bounds. In particular, S-FRTDP needed 202 seconds in average to solve problem with six tasks and R-FRTDP required 69 seconds. Indeed, the time reduction is quite significant compared to LRTDP, which demonstrates the efficiency of using bounds to prune the action space and produce efficient trajectories. Finally, in Table 3 the gain in speed of R-FRTDP compared to S-FRTDP is mainly due to the lower bound as the planning time is less for L-FRTDP than for U-FRTDP.

Table 4 details the average number of Bellman backups made for LRTDP, S-FRTDP, and R-FRTDP for problems of 4 to 6 tasks. The average number of actions performed for each backup made at s_0 are in parenthesis. The results demonstrate that tights initial bounds helps to reduce the number of backup to perform. Indeed the number of backups required for convergence is significantly smaller for R-FRTDP and S-FRTDP than for LRTDP. Indeed, the tighter the bounds are, the faster these bounds converge to the optimal value. Furthermore, since the action space is pruned for the bounded versions, the backups becomes less complex, even if two values are computed.

6. Conclusion

This paper has contributed to solve complex stochastic resource allocation problems. First, a distributed Q-values approach has been proposed when the resources are distributed among agents a priori, but the actions made by an agent may influence the reward obtained by at least another agent. In this case, the experiments have shown that the proposed approach is very efficient.

Then, when the available resource are available to all agents and no distributed Q-values is possible, we proposed tight bounds with heuristic search based on real-time dynamic programming. In this context, we have implemented BRTDP and FRTDP two existing real-time dynamic programming with two new tight bounds. The experiments have shown a faster convergence of these two algorithms with the proposed tight bounds.

The only condition for the use of our proposed bounds is that each task possesses consumable and/or nonconsumable limited resources, which we feel is a very frequent problem in resource allocation domains.

As future work, we plan to complete our experiments with a rollout algorithm,[25] which is an approximation of dynamic programming where the value of the states are initialized using a lower bound heuristic and a backup is performed on reachable states until a certain depth is reached. In this context, the value of the further states can be approximated using monte-carlo simulation and our marginal revenue lower bound can be used as its initial heuristic.

Acknowledgments

This research was supported by the Natural Sciences and Engineering Council of Canada and the Fonds Québécois de la Recherche sur la Nature et les Technologies. We would also like to thank the anonymous reviewers for their helpful comments and suggestions.

References

1. W. B. Powell. *Approximate Dynamic Programming: Solving the Curses of Dimensionality*, volume 703. Wiley-Blackwell, 2007.
2. R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*, volume 116. Cambridge Univ Press, 1998.
3. C. Szepesvári. Algorithms for reinforcement learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 4(1):1–103, 2010.
4. A. Barto, S. Bradtke, and S. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1):81–138, 1995.
5. B. Bonet and H. Geffner. Labeled LRTDP approach: Improving the convergence of real-time dynamic programming. In *Proceedings of the Thirteenth International Conference on Automated Planning & Scheduling* (*ICAPS-03*), pages 12–21, Trento, Italy, 2003a.
6. H. B. McMahan, M. L., and G. J. Gordon. Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In *ICML '05: Proceedings of the Twenty-Second International Conference on Machine learning*, pages 569–576, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-180-5. http://doi.acm.org/10.1145/1102351.1102423.
7. T. Smith and R. Simmons. Focused real-time dynamic programming for MDPs: Squeezing more out of a heuristic. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI)*, Boston, USA, 2006.
8. B. Bonet and H. Geffner. Faster heuristic search algorithms for planning with uncertainty and full feedback. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*, August 2003b.
9. E. A. Hansen and S. Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1-2):35–62, 2001.
10. S. Singh and D. Cohn. How to dynamically merge markov decision processes. In *Advances in Neural Information Processing Systems*, volume 10, pages 1057–1063, Cambridge, MA, USA, 1998. MIT Press. ISBN 0-262-10076-2.
11. R. S. Pindyck and D. L. Rubinfeld. *Microeconomics*. Prentice Hall, 2000.

12. R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, New Jersey, 1957.
13. R. A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, Massachusetts, 1960.
14. M. L. Puterman. *Markov decision processes: Discrete stochastic dynamic programming*. John Wiley & Sons, Inc. New York, NY, USA, 1994.
15. D. P. Bertsekas and J. N. Tsitsiklis. Neuro-dynamic programming: An overview. In *Decision and Control, 1995. Proceedings of the 34th IEEE Conference on*, volume 1, pages 560–564. IEEE, 1996.
16. C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.
17. S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, 3rd edition. Prentice-Hall, Englewood Cliffs, 2009.
18. B. Raphael P. Hart, and N. Nilsson. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Science and Cybernetics*, 4(2):100–107, 1968.
19. Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM*, 32(3):505–536, 1985. ISSN 0004-5411. http://doi.acm.org/10.1145/3828.3830.
20. R. E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(3):189–211, 1990.
21. S. J. Russell and A. Zimdars. Q-decomposition for reinforcement learning agents. In *International Conference on Machine Learning* (*ICML*), pages 656–663, 2003.
22. G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical report CUED/FINFENG/TR 166, Cambridge University Engineering Department, 1994.
23. P. Plamondon, B. Chaib-draa, and A. Benaskeur. A q-decomposition and bounded RTDP approach to resource allocation. In *Proceedings of the Sixth International Conference on Autonomous Agents and Multiagent Systems* (*AAMAS-2007*), May 2007.
24. A. Beynier and A. I. Mouaddib. An iterative algorithm for solving constrained decentralized markov decision processes. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence* (*AAAI-06*), 2006.
25. D. Bertsekas. Rollout algorithms for constrained dynamic programming. Technical report 2646, Lab. for Information and Decision Systems, MIT, Mass., USA, 2005.