Decomposition Techniques for a Loosely-Coupled Resource Allocation Problem

Pierrick Plamondon & Brahim Chaib-draa
Computer Science & Software Engineering Dept
Laval University
Ste-Foy, Québec, Canada
{*plamon, chaib*}*@damas.ift.ulaval.ca*

Abder Rezak Benaskeur
Decision Support Systems Section
Defence R&D Canada — Valcartier
Val-Bélair, Québec, Canada
Abderrezak.Benaskeur@drdc-rddc.gc.ca

Abstract

We are interested by contributing to stochastic problems of which the main distinction is that some tasks may create other tasks. In particular, we present a first approach which represent the problem by an acyclic graph, and solves each node in a certain order so as to produce an optimal solution. Then, we detail a second algorithm, which solves each task separately, using the first approach, and where an on-line heuristic computes the global actions to execute when the state of a task changes.

1. Introduction

We are interested in perfectly observable stochastic resource allocation problems with hard real-time constraints. The resources are allocated to execute a certain number of tasks, where, in particular, the presence of a task may influence the presence of other tasks. Furthermore, the states of a task are strongly connected since all non-absorbing states are always reachable from each other. However, the tasks are loosely connected.

On the other hand, there is an explosion of the state space in our type of problem, due to Bellman's curse of dimensionality [1]. To alleviate this, many researchers decomposed the state space to diminish the planning time ([5], [2], [8]). In this paper, we present two novel decomposition techniques which solve efficiently the problem when the strongly connected states are grouped in separate components. These techniques are a specialization of the Markov Decision Process (MDP) framework, associated to our problem characteristics. Another originality of our work is to consider effectively the criticality of each task, by defining a weight factor. We now formalize the problem.

2. Problem Formulation

We consider a state as a conjunction of tasks to accomplish by a *planning agent*. A task corresponds to a work to accomplish for the agent. On the other hand, an action corresponds to use a resource to accomplish a task. An action may change the probability to execute a particular task, or all tasks. The tasks may be in many possible states, which are strongly connected. The states are strongly connected because from a non-absorbing state s in a task, we can transit to any other non-absorbing state s' of this task, and go back to s [7]. Furthermore, we have some tasks which may be created by another task. We consider the creation of a task as stochastic. A task which may not create another task is considered as *critical*. The parent of a critical task, considered as *non-critical*, have to be accomplish, because they may create a critical task. Also, the planning agent is under hard real-time constraints to produce an effective policy. The very high number of states in this type of problem coupled with the time constraint makes it very complex.

We should note that a resource allocation problem is known as being NP-complete [9]. The state space consists of the cross product of each task individual state spaces and the available resources; the action space is the set of resource assignment. Meuleau et al. [5] proposed an approach where each task is solved optimally in an independent manner. The solutions are merged afterwards with a greedy strategy, thus producing an approximate solution. On the other hand, Dolgov et al. [2] extended Meuleau's works by assigning resources to each task and then producing the global optimal policy. Another promising approach is to plan for the resources separately [8]. In particular, the approaches of all cited authors do not consider the fact that a task may create other tasks, which is the base of our algorithms. We now introduce Markov Decision Processes (MDPs), as a modelling tool.

2.1. Markov Decision Processes MDPs

To model our problem, as it is stochastic, we use Markov Decision Processes (MDPs) [1]. Indeed, MDPs provide a well-studied and simple, yet a very expressive model of the world. An MDP for our problem is given by:

- A state space S. A state $s \in S$, represents a conjunction of the particular state s_{t_i} of each task t_i in the environment. Also, S contains a non empty set $G \subseteq S$ of goal states.

- Actions $A(s) \subseteq A$ applicable in each state $s \in S$. The possible actions are limited by L_{res} and G_{res}, which are the local and global resource constraints on the amount of resource r that may be used on a single step (local), or in total (global).

- Transition probabilities $P_a(s'|s)$.
- State rewards $R(s)$ related to the problem. The reward of a state s is the sum of the product of the state reward $r(s_{t_i})$ by the relative weight $W(t_i)$ of each task.

A solution of an MDP is a function π mapping states 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. Bellman's *principle of optimality* [1] forms the basis of the stochastic dynamic programming algorithms used to solve an MDP. In particular, the optimal value of a state is the immediate reward for that state add to the expected value of the next state transition probability, assuming that the agent chooses the optimal action. That is, the value of a state $V(s)$ is given by:

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

subject to the local and global resource constraints for each state task s_{t_i} of a state s

$$\sum_{i=1}^{nbTasks} res(\pi_{t_i}(s_{t_i})) \leq L_{res} \text{ for all } s \in S \quad (2)$$

$$\sum_{j=1}^{S_{tra}} \sum_{i=1}^{nbTasks} res(\pi_{t_i}(s_{t_i})) \leq G_{res} \quad (3)$$

for all system trajectories tra

Using Bellman's principle of optimality avoid enumerating all possible solutions, and is sometimes called *pruning by dominance*. The standard dynamic programming algorithms to solve an MDP are *value iteration* [1] and *policy iteration* [4]. However, these classical algorithms suffer from the so-called *curse of dimensionality* [1] : the number of states grows exponentially with the number of variables that characterize the planning domain. We now describe how we weight the specific importance to achieve each task in the environment.

2.2. Task Weighting

We have developed an heuristic to weight the importance of each task. based on the fact that some tasks may create other tasks. We can represent these dependencies between tasks with a graph. An important characteristic for the problem we are interested in, is that the representation of task dependencies in a graph produces a certain number of acyclic graphs. Note that an acyclic graph is one that all state transitions always results in a state that were never previously visited, thus implying a partial order on the set of states. Figure 1 represents the acyclic graphs where two tasks, t_1 and t_2, are in the environment. t_1 may create task t_3, and t_2 may create task t_4. So, task t_3 and t_4, which are leafs, may produce dire consequence for the robot agent. We use Algorithm 1 to define the weight factor $W(t_i)$ representing the relative importance to accomplish each task t_i.

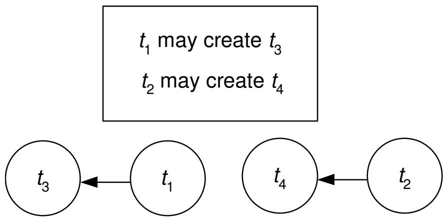

Figure 1. Acyclic graphs of task dependencies.

Algorithm 1 Task weighting.

```
Function TASK-WEIGHTING(graph $DepG$)
while $DepG \neq null$ do
  Remove from $DepG$ a task $t_i$, such that no descendent of $t_i$ is in $DepG$
  if $t_i$ is a leaf task then
    $W(t_i) \leftarrow P(Failt_i|s_{t_i}) \times$ CONS$(t_i)$
  else
    $W(t_i) \leftarrow \sum_{j=1}^{nbTasks} P(\mathrm{C}(t_j)|s_{t_i}) \times$
        $E(t_j|(\mathrm{C}(t_j)|s_{t_i})) \times W(t_j)$
```

The $DepG$ graph contains many acyclic graphs, which nodes (tasks) are affected a weight factor. The tasks are traversed using a backward approach just like in the AO* algorithm [6] for an acyclic graph. For a critical task t_i, we define the weight factor $W(t_i)$ as the product of the probability that the task is failed of being accomplished by the planning agent, considering its current state ($P(Fail_{t_i}|s_{t_i})$), by the consequence it may inflict to the planning agent (CONS(t_i)). For a non-critical (non-leaf) task, the weight is specified according to the sum of each task t_j, as the product of the probability of creating j ($P(\mathrm{C}(t_j)|s_{t_i})$), by the expected number of created tasks j ($E(t_j|(\mathrm{C}(t_j)|s_{t_i}))$), and by the weight factor of j ($W(t_j)$). The probabilities $P(\mathrm{C}(t_j)|s_{t_i})$ and $P(Fail_{t_i}|s_{t_i})$ are computed a priori considering that no action is made using Bayesian inference. Consequently, the weight function overestimates the weight of all tasks, since the probability of creating other tasks is higher when no action is made, than with an optimal policy. However, the weight of a task should not change as the computing of a policy is made, because some state value may be overestimated. Considering this, we have used this approach to define the probability, which is an approximation. The next section describes two decomposition strategies to reduce the planning complexity.

3. Decomposition Techniques

3.1. Acyclic Decomposition Algorithm

An efficient decomposition technique may regroup together strongly connected states [5]. In the same sense, it is known that a planning problem which may be represented with an acyclic graph, instead of a cyclic one, is generally easier to solve [3].

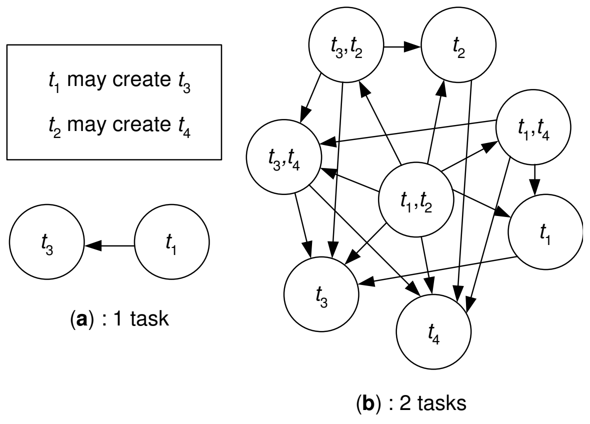

Figure 2. The acyclic graph of cyclic components.

We recall that an acyclic graph is one that all state transitions always results in a states that were never previously visited, thus implying a partial order on the set of states. On the other hand, a cyclic graph may visit certain states more than one time. The idea we have here is to transform our problem in an abstract acyclic one, which contains many cyclic components. A component corresponds to a group of task, and the graph contains a component for each possible task combination. Since each task is strongly connected, the components are cyclic. On the other hand, the acyclic graph represents the possible task transitions. One may use [7] linear algorithm for detecting the *strongly-connected components* of a directed graph to create the acyclic graph. Figure 2 shows the acyclic graph when task t_1 (a) or task t_1 and t_2 (b) of Figure 1 are in the environment. All nodes represent a cyclic component. This graph suppose that a task may create one other task in maximum. The signification of a link in Figure 2, means simply a change the tasks the planning agent has to achieve. Thus, it has a task transition meaning, as opposed to a task creation in Figure 1. Once the abstract acyclic graph is formed, we can solve it using a backward approach just like in the AO* algorithm [6]. Algorithm 2 describes how we calculate the value V_c of each strongly connected component c of an acyclic graph AcG.

Algorithm 2 Acyclic decomposition.

```
Function ACYCLIC-DEC(graph $AcG$)
while $AcG \neq null$ do
    Remove from $AcG$ a component $c$, such that no descendent of $c$ is in $AcG$
    $V_c \leftarrow$MDP-ALGO($c$)
```

This algorithm solves each component, using an MDP algorithm ("MDP-ALGO(c)"), from the leaf to the root of the graph. This way, each component may only transit to a solved component, thus each component have to be solved one time. "MDP-ALGO(c)" may be any algorithm to solve an MDP, such as standard approaches like value iteration or policy iteration. For example, in Figure 2 (b), we can choose to remove whichever of t_3 or t_4, and solve it using MDP-ALGO(c). Then, if we removed t_3 in the first iteration, we may now remove t_4, or vice versa. When both t_3 and t_4 are removed, t_1 and t_2 may be removed. We remove components in this order, until the component t_1, t_2 is removed and solved.

Theorem 3.1 *The acyclic decomposition algorithm produces an optimal policy.*

Proof: It is known that the AO* algorithm generates an optimal policy of a problem represented by an acyclic graph [6]. However, the nodes in our graph regroups many states (component), while in AO*, it is an atomic state. Thus, to prove the optimality of the acyclic decomposition algorithm, we have to determine whether a component exhibits the same characteristics as a state. Since the action of all states in the component is optimal, we could say that the component, which is an abstract state, has an abstract optimal action. Thus, the computation of the optimal action for a state in AO*, may be viewed as the computation of the optimal abstract action for a component in our algorithm. It follows that the acyclic decomposition algorithm produces an optimal policy. ■

The planning time is reduced compared to a standard approach, however, as we can observe on Figure 3 of Section 4, it still takes about 20 seconds to plan for a problem with three initial tasks. For our hard real-time problem, we should further reduce the complexity of the planner. To this event, the next section introduces an on-line decomposition strategy to further reduce the planning time; however, at the compromise of producing an approximate solution.

3.2. On-line Decomposition Algorithm

A standard dynamic programming, or the previous acyclic decomposition approach to solve the problem is done off-line. Indeed, all the computation is made before executing any action. Since we need a decision in real-time, we introduce an on-line approach, in which planning and the execution of actions are carried out concurrently. This approach is elaborated based on the acyclic decomposition approach. Indeed, the approach plans a separate policy for each task in the environment, using the ACYCLIC-DEC(graph AcG) function. A solution for one task is produced very quickly using the acyclic algorithm. We adopt this approach to plan for each task, because each task are strongly connected, but loosely connected with the other tasks. Algorithm 3 overviews our decomposition approach.

The first part of the algorithm computes a value function V_{t_i} for the state space of each task S_{t_i} in the environment using a standard dynamic programming approach ("ACYCLIC-DEC(S_{t_i})"). For example, if one task is in the environment, we compute the value function for this task only, and all of its potential descendant. When the component value function of each task is made, we proceed to the on-line phase. In particular, we determine the global sub-optimal action to execute each time a task changes. The local and global resource constraints (L_{res}) had been verified in the off-line planning of each task. When

Algorithm 3 On-line decomposition.

```
Function ON-LINE-DEC(State space $S$)
for each task $t_i$ do
  $V_{t_i} \leftarrow$ ACYCLIC-DEC($S_{t_i}$)
repeat
  if the state of a task has changed then
    $a(s) \leftarrow \arg\max_{a \in A(s)} \sum_{i=1}^{nbTasks} \sum_{s'_{t_i}} P_a(s'_{t_i}|s_{t_i}) V_{t_i}(s'_{t_i})$
until the scenario is over
```

computing the global action, we verify the local and global constraint for this state only, thus producing an approximative solution. We now detail the results we have obtained using three algorithm; standard dynamic programming, the acyclic decomposition as presented by Algorithm 2 of Section 3.1, and the on-line decomposition as presented by Algorithm 3 of Section 3.2.

4. Empirical results

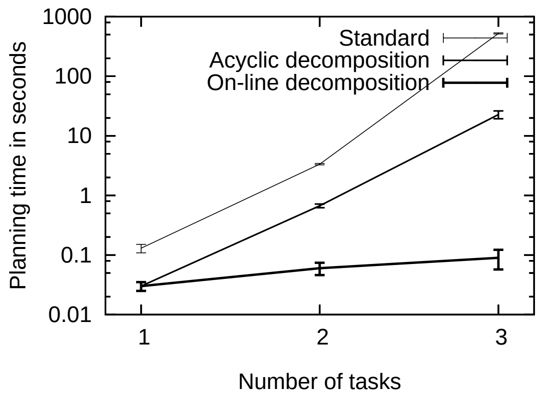

Figure 3. Planning time for our problem.

To test the algorithms, we computed a policy with a certain number of initial tasks which may creates other tasks. From the results, we can conclude that planning an optimal policy, using a standard dynamic programming approach (value iteration), for this problem is very complex, as shown on Figure 3. The acyclic decomposition algorithm reduces the planning time, but it is still too high for our real-time application. So, this approach is promising, but we should improve it in the future. On the other hand, the on-line decomposition algorithm produces an approximative policy in a very short time.

Table 1 details the percentage of the optimal obtained with our on-line decomposition planner. We can see that the value of the policy decreases when the number of tasks augment. As a matter of fact, when one threat is present, the policy is optimal, because it uses the optimal acyclic decomposition algorithm for the task.

Table 1. The percentage of the optimal obtained with our on-line decomposition planner.

# of tasks	Average cases	Worst cases
1	100 $\pm 0\%$	100
2	97,64 $\pm 0,45\%$	96,77
3	95,4 $\pm 0,71\%$	94,18

5. Conclusion

We are working on a multi-tasks problem, where the presence of some tasks is dependent on the presence of others. Such problem produces an enormous state space. To compute a policy in a timely manner for this sort of problem, we have decomposed the state space. In particular, a task is strongly coupled within its own possible states, but weakly coupled with the other tasks. In our approach, we estimated the relative importance of achieving each task using a weight function. We have solved our resource allocation problem in an effective manner using our optimal acyclic or approximate on-line decomposition approaches.

We would like to improve our decomposition algorithms. For example, computing the solution using the acyclic decomposition approach where all components are solved using the on-line decomposition approach, instead of a standard one seems promising.

References

[1] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, New Jersey, 1957.

[2] D. A. Dolgov and E. H. Durfee. Optimal resource allocation and policy formulation in loosely-coupled markov decision processes. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, pages 315–324, June 2004.

[3] E. A. Hansen and S. Zilberstein. LAO * : A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1-2):35–62, 2001.

[4] R. A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, Massachusetts, 1960.

[5] N. Meuleau, M. Hauskrecht, K. Kim, L. Peshkin, L. P. Kaebling, T. Dean, and C. Boutilier. Solving very large weakly coupled markov decision processes. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 165–172. AAAI Press, 1998.

[6] N. J. Nilsson. *Principles or Artificial Intelligence*. Tioga Publishing, Palo Alto, Ca, 1980.

[7] R. E. Tarjan. Depth first search and linear graph algorithm. *SIAM Journal on Computing*, 1(2):146–172, 1972.

[8] C. C. Wu and D. A. Castanon. Decomposition techniques for temporal resource allocation. Technical report: Afrl-va-wp-tp-2004-311, Air Force Research Laboratory, Air force base, OH, 2004.

[9] W. Zhang. Modeling and solving a resource allocation problem with constraint techniques. Technical report: Wucs-2002-13, Washington University, Saint-Louis, Missouri, 2002.