Solving Policy Conflicts between Markov Decision Processes

Abstract

Markov decision processes (MDPs) normally execute a single action per state; however, in many domains such as robotics, it is sometimes necessary to execute multiple actions at the same time. We have worked on a novel framework based on functional decomposition that divides a complex MDP into several sub-problems. Each sub-problem is defined as an MDP and solved independently, and their individual policies are combined to obtain a global policy. This combined policy can execute several actions per state but can introduce policy conflicts. We define two kinds of conflicts, resource and behavior conflicts, and propose solutions for both. The first kind of conflict is solved off-line using a two phase process which guarantees a near-optimal global policy. Behavior conflicts are solved on-line based on a set of restrictions specified by the user. If there are no restrictions, all the actions are executed concurrently; otherwise, a constraint satisfaction module selects the action set with higher expected utility. The use of constraints can generate different behaviors, according to a set of user-defined restrictions. Experimental results in a robotics domain show that our approach produces a more efficient solution when several actions can be concurrently executed.

Introduction

The main drawback of MDPs is due to the *curse of dimensionality*. In its basic form, policy and value iteration require an explicit representation of states and actions and need to explore the entire state space during each iteration. In particular, if multiple concurrent actions are allowed, this will imply a further increase in complexity, as all actions combinations need to be considered.

To deal with the complexity problem there are three main approaches: *factorization,* in which the state space is represented in a factored form (e.g. (Boutilier, Dearden, and Goldszmidt 1995; Hoey et al. 1999)); *decomposition*, that divides the global problem into smaller problems that are solved independently and their solutions are combined (e.g. (Meuleau et al. 1998; Laroche, Boniface, and Schott 2001)); and *abstraction*, that creates an abstract model where states with similar features are grouped together(e.g. (Parr and Russell 1997; Dietterich 1998; Jong and Stone 2005; Li, Walsh, and Littman 2006; Dean and Givan 1997)). Next we describe each alternative.

Factored MDPs Factored MDPs address the complexity problem via compactly specifying the model of the MDP in factored form; the state space (**S**) is modeled by a set of variables $X = X_1, .., X_n$, and the actions are described as having an effect on specific variables under certain conditions, implicitly inducing the transition function.For instance, (Boutilier, Dearden, and Goldszmidt 1995) uses a $two - slide$ *temporal Bayesian network* (TBN) (Tawfik and Neufeld 1994) to represent it, algorithm retains the basic steps of *PI* but it exploits the independencies reflected in the TBN. SPUDD (Hoey et al. 1999) uses algebraic decision diagrams (ADDs) (Bahar et al. 1997) to represent the transition and value functions, and based on this representation it uses very efficient techniques for ADDs to implement value iteration. Although factored representations allow to solve quite *large* MDPs, they do not address directly the problem of concurrent actions.

Abstraction Under the abstraction approach, a group of states in the original MDP is mapped to a single abstract state. For this, the group of states must be *equivalent*, or at least share the same local behavior. For instance, (Jong and Stone 2005) is an approach that discovers a set of state variables that are irrelevant for a policy, finds the states where this set of variables is irrelevant, and encapsulates them into one state using temporal abstraction. Hierarchical abstract machines (HAMs) (Parr and Russell 1997) consist of non−deterministic finite state machines whose transition may invoke other lower level machines. These approaches also do not solve the issue of concurrent actions.

Decomposition Decomposition is based on the old principle of *divide and conquer*. In (Meuleau et al. 1998) the problem is divided into smaller tasks. In an off–line phase, the value function and optimal policy for the MDP associated to each subtask is computed. In an on–line phase, these value functions are used within a heuristic search procedure to assign resources to each task, and depending on the resources allocated, the action to execute is selected. In (Laroche, Boniface, and Schott 2001) the state space is divided into physical regions (office, corridor, room, etc.), and

each one is solved as an MDP. The regions have communication with their neighbors through an state (initial or goal) called *intersection*. Then a directed graph is built, where each intersection is a vertex and the regions are the arcs. Each arc is associated to the value function of its respective MDP, and to get a solution the shortest path is found. In (Dietterich 2000), the MDP is divided into several subtasks, identifying a subset of state variables which are *relevant* for each subtask. It then defines a value function and policy using only these relevant variables for each task.

Most decomposition approaches make a *serial* decomposition; that is, the problem is partitioned in subtasks that are executed sequentially. These techniques can not execute concurrent actions. There are few previous work on *parallel decomposition* (e.g. (Meuleau et al. 1998; Sucar 2007)), where the subtasks are executed concurrently, so in principle they can execute several actions simultaneously. Elinas et al. (Elinas et al. 2004) propose an approach for coordinating a service robot based on concurrent MDPs, assuming the subtasks are independent and that there are no conflicts between actions. Parallel MDPs (Sucar 2007) consider the combination of local policies based on Q values, however this combination may result in undesirable behavior in some states, which correspond to conflicts between the local policies. The previous approaches in general assume that the subtasks are independent and they do not consider potential conflicts between the policies of each subtask. In this paper we address these problems. The rest of the paper is structured as follows. We first present the proposed architecture and describe the techniques for solving conflicts. Finally, we describe the experimental evaluations and conclude with directions for future work.

General architecture

Our work is motivated by robotics, specially by service robots, where frequently it is necessary to simultaneously perform multiple tasks to accomplish certain goal. We propose a framework for solving this type of problems based on MDPs. It considers four main phases:

1. Decompose the problem into several subtasks based on functionally.
2. Solve each MDP independently considering the global objective of the subtask.
3. Combine the local policies and solve resource conflicts. Define restrictions for the behavior conflicts.
4. Execute the local policies in parallel, solving behavior conflicts on–line based on the set of restrictions.

Figure 1 shows the main phases of the proposed solution. Next we describe each phase in more detail.

Functional decomposition

Our focus is on functional design –the process of breaking a system into interacting subcomponents based on functionality. Each function contributes to a common objective. All these functions are running at the same time focusing on a particular objective, and at the same time they all contribute to the global goal. In the first phase the global problem is divided in k functions. Each function has its own objective, and it is modeled as an factored MDP, named $subMDP$. The $subMDPs$ are solved independently to obtain their value function (V_j^*) and their optimal policy (π_j^*). In our current implementation we use SPUDD (Hoey et al. 1999) to solve each $subMDPs$. Once the local policies for all the $subMDP$ are obtained, their solutions are combined to obtain a global policy. More details on (ecx).

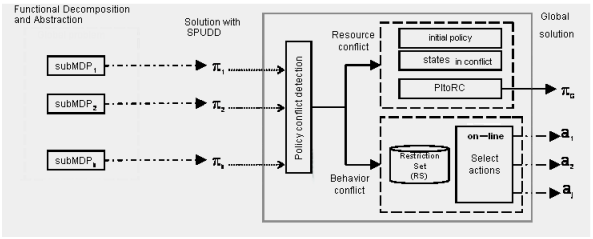

Figure 1: The general architecture is composed by four phases: (a) functional decomposition; (b) definition and solution of the subMDPs; (c) detection of conflicts between local solutions and combination; and (d) execution of the global solution.

Solving policy conflicts

Once the $subMDP$ for each subtask is solved, in principle their policies can be executed simultaneously to solve the global problem. That is, at each time state (s_i), each $subMDP$ selects an action to execute according to its local optimal policy: $\pi_1^*(s_i) = a_1$, $\pi_2^*(s_i) = a_2$, ..., $\pi_k^*(s_i) = a_k$. This set of actions are denoted by $A_{s_i} = \{a_1, a_2, ..., a_k\}$. When these policies are combined conflicts between them can arise. We have identified two types of conflicts when two or more actions are executed at the same time: *Resource conflicts*, and *Behavior conflicts*. .

(i) Resource conflicts This type of conflict arises when two or more actions requires the same resource, so it is impossible to execute them at the same time. For example, suppose that a robot is on track to deliver an urgent message, but the battery is running low. In this state, the $navigation$ function may want to turn left to reach the target point, while the $energy$ function may want to turn right to recharge the battery. This type of conflicts are solved off–line via a two-phase process.

First, an initial global policy is obtained by combining the local policies, such that if there is a conflict between the actions selected by each $subMDP$ for certain state (s_i), the action with maximum value is considered, and the state is marked as a *conflict* state. The previous policy is the initial policy, and considers only the states marked as conflicts to improve the initial policy, using a modified policy iteration algorithm. With these considerations the time complexity is drastically reduced. In this case no simultaneous actions are allowed. See (ecx) for details.

(ii) Behavior conflicts When two or more $subMDPs$ require different resources, to execute their actions, but if the actions are executed at the same time the system results in an undesirable behavior, we say that a *behaviour conflict* is present. To solve this type of conflicts we also consider two steps:(i) *define restrictions* (off–line); and (ii) *execute concurrent actions* (on–line).

Definition of Restrictions

The user establishes a restrictions set (**RS**) in terms of actions, from different $subMDPs$, that should not be executed at the same time. We have defined a syntax for it. We consider that given a functional decomposition of the problem and domain knowledge, it is relatively easy for a person to specify these restrictions. For example, in the robotics domain, imagine that the robot is interacting with a user, while the *navigation* module is trying to evade an obstacle (in this case the same person). Therefore these two actions should not run at the same time, and the system must choose which one to run first, based on the **RS**.

The system has to maintain the Markov property, it means that future behavior depends only on the current state, not on the state(s) in the past. So, on each time state (s_i) the systen only knows the current action of each $subMDPs$ (A_{s_i}), and the actions that are running(E_{s_i}). So, we have defined four kinds of restriction operators:

1. a_i **not_start** a_j. The actions a_i and a_j must not start at the same state.
2. a_i **not_before** a_j. The action a_i must not start before a_j.
3. a_i **not_after** a_j. The action a_i must not start after a_j.
4. a_i **not_during** a_j. If the action a_j is running a_i must not start .

Conjuntion (`AND`), and disjunction (`OR`) of actions are allowed (e.g. interact_whit_user `AND` recognize_user **not_during** robot_turnning).

Concurrent Actions

Once the **RS** is defined, and the optimal policies are obtained for all $subMDPs$, the system runs the k $subMDPs$ simultaneously. At each time state (s_i), the agent consults each optimal policy (π_i^*) of the k $subMDPs$. If there are no restrictions between the actions for all $subMDPs$, all actions are performed concurrently. Otherwise, we solve the problem of selecting the actions via a constraint satisfaction module (*CSM*).

The constraint handling rules (CHR) library of SICStus prolog was used to built the *CSM* that finds the actions set to be executed in this time state (A'_{s_i}), based on the **RS** defined by the user.

The action of each $subMDP$ (A_{s_i}) and the actions that are running (E_{s_i}), and the value functions of both are the input of the *CSM*. We define a data type *actions*, the terms acepted are the actions of all $subMDPs$; the **RS** defined by the user is coded as predicates into a *.pl* file. A set of rules determinates the action(s) set(s) that satisfy the **RS**, $(A_{sc_1}, A_{sc_2}, ..., A_{sc_m}, m >= 1)$. Finally the *CSM* module

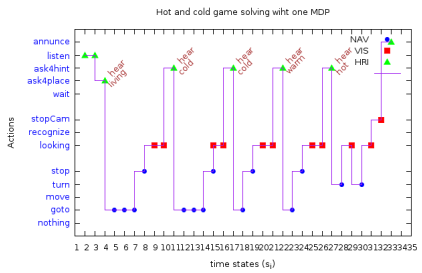

Figure 2: Sequential actions solution needs more time states to get a goal.

calculates the actions set that maximizes the expected reward ($A'_{s_i} = max_A V(A_{sc_j})$) and the actions of A'_{s_i} are sent to their respective $subMDP$ to be executed concurrently. Experimental results in a robotics domain are showed in the next section.

Experimental results

This work is motivated by robotics, in particular service robots. We present two sets of experiments, with **concurrent actions**: (i) hot & cold game, *without conflicts*; and (ii) robot in a message delivery task, with *behavior conflicts*.

Hot & Cold game: concurrent actions without conflicts

To test our approach we consider a hot & cold game. In this test we show that our approach produces a more efficient solution when several actions can be concurrently executed. For this task the robot knows one or more objects. One of these objects is hidden somewhere in the environment, next the robot gets started with a key frase, for example: "Robot, find the <object>". The robot has to navigate (*navigation*) around the escenary to find the object, while searching (*vision*) the user gives hints to the robot via voice (*interaction*), such as: *hot* or *warm*, or *cold* as the robot gets closer or farther from the object until it is identified. The robot may ask for a hint. For this task it is considered that the robot has a 2D map of the environment and laser and sonar sensors, a camera PTZ, headers and microphone. We break hot & cold game into three interacting subcomponents based on functionality. On subtask (*navigation*) the robot navigates around the scenary safely, avoiding obstacles in the trajectory of the robot. A second function (*vision*), the robot moves the camera looking for an object. And finaly the thrid subtask (*interaction*), the robot remains on hold listening for an order or hint, or asks for a hint to the user, if necesary.

On figure we can see that only one action is executed by time state, if we can execute concurrent actions under this focus we need to define each action as a *set of actions*, but this increases significantly the size of the model, and it makes more complex its specification. Our approach allows execute concurrent actions, see figure , and gets the goal an less steps.

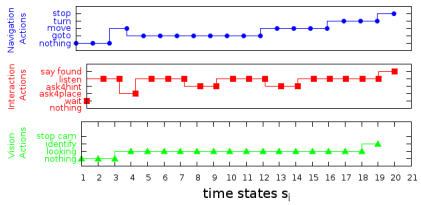

Figure 3: Our approach produces a more efficient solution when concurrent actions can be executed.

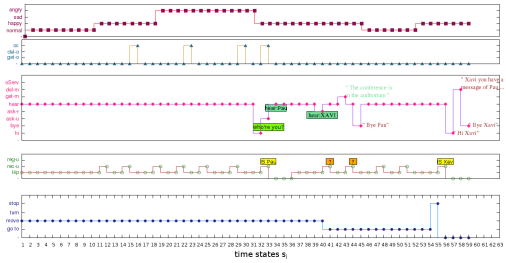

Figure 4: Messenger robot task, without restriction.

Table 1: Restriction set **RS** for messenger robot

action(s)	restriction	action(s)
get message	**not_during**	turn OR advance
ask_user_name	**not_before**	recognize_user
recognize_user	**not_start**	avoid_obstacle
get_object OR deliver_object	**not_during**	directed towards OR turn OR moving

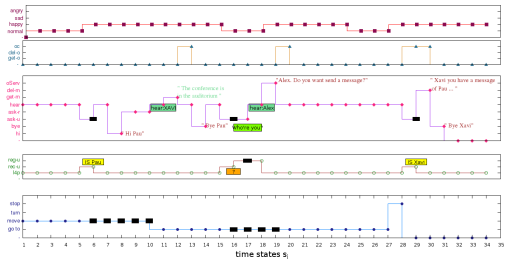

Figure 5: The bahviour

Messenger robot: concurrent actions with behaviour conflicts

The objective of this experiments is to test the approach, in particular evaluate the CSM under the **RS** for solving behaviour conflicts, and execute concurrent actions. The goal in this task is to receive and deliver a message, an object or both, under user request. The interaction again is through natural language. The user gives an order to send a message/object and the robot asks for the name of the sender and the receiver. The robot either records a message or uses its gripper to hold and object,and navigates to the receiver's place and deliver the message/object. The user has the hability to express its emotions according its state. We break the task in five $subMDPs$: (i) navigation, the robot navigates safely on different scenarios; (ii) vision, for looking and recognizing people and objects; (iii) interaction, to listening and talking with a user; (iv) manipulation, to get and deliver an object safely; and (v) expresion, during the task an animated face expresses the robot emotions. We present three cases: (1) without restrictions, and (2) restrictions between all $subMDPs$.

No restrictions On principle, all actions can be executed concurrently, but as shown in figure 4 the robot incurrs in undesirables behaviours. For example *vision* can not get a good image to analyze and recognize the user (see the time states from s_2 to s_{12}), because *navigation* is moving while the robot is trying to avoid the user. So, the user has to be catched the microphone to interact. When the *vision* module detects a person, (s_3, s_7, and s_{11}), the robot does not knowns who is, in the time states s_4, s_8, and s_{12} vision and interaction $subMDPs$ try to identificate who is, if the user is already know, results inappropiate ask for his name again and again.

Restrictions actions between the *k* $subMDPs$ Table 1 lists some restrictions introduced for this test, to avoid the problems mentioned in the previous section. Figure 5 shows that *vision* $subMDP$ has a better performance than when no restrictions. In only two states, s_3 and s_4, the robot is able to detect and recognize the user. So, like the user detected is already know, the action determinated by *interaction* $subMDP$ (*ask_user_name*) on time state s_4 is eliminated by the *CSM* module. In the same behaviour conflict (s_{25}), but when the user is not knows by the robot, the action (*ask_user_name*) is executed after recognition (s_{26}). At the same time when the *vision* or *interaction* $subMDPs$ execute some action the *CSM* module eliminates the *navigation* actions if these afect their performance.

We may also note that more tasks can be performed in less time states.

Conclusions and Future Directions

We have presented a framework for solving complex Markov decision processes based on functional decomposition which partitions the problem in several simpler MDPs. We obtain the optimal policy for each $subMDP$, and then combine the results to obtain a global solution such that the actions of each $subMDP$ are executed concurrently. We have identified two types of conflicts: resource and behavior conflicts. For resource conflicts the local policies are combined to obtain an initial policy which is refined using a modified policy interaction algorithm. For behavior conflicts we have defined a syntax to specify a restrictions set (**RS**) between actions to identified conflicts; and the action set with highest expected value (A'_{s_i}) is selected by the constraint solver module (*CSM*). Experiments show the feasibility of the proposed approach.

We are currently testing our approach in a more complex

experiment. The global problem has a larger state space, including more functions thus increasing the conflicts between local policies. The objective of the messenger robot, is to pickup and deliver more than one object or message between users in an office environment. The system has to plan a delivery schedule. While the robot is delivering, it can also guide a visitor to a particular office. For these new experiments we are adding more actions to: (i) navigation, (ii) vision, and (iii) interaction; we are combining one more $subMDP$: schedule. We are also working on a more formal analysis of the optimality of the proposed approach.

References

Bahar, R. I.; Frohm, E. A.; Gaona, C. M.; Hachtel, G. D.; Macii, E.; Pardo, A.; and Somenzi, F. 1997. Algebraic decision diagrams and their applications. *Formal Methods in System Design* 10(2/3):171–206.

Boutilier, C.; Dearden, R.; and Goldszmidt, M. 1995. Exploiting structure in policy construction. In *IJCAI*, 1104–1113.

Dean, T., and Givan, R. 1997. Model minimization in markov decision processes. In *AAAI/IAAI*, 106–111.

Dietterich, T. G. 1998. The maxq method for hierarchical reinforcement learning. In *ICML*, 118–126.

Dietterich, T. G. 2000. Hierarchical reinforcement learning with the maxq value function decomposition. *J. Artif. Intell. Res. (JAIR)* 13:227–303.

Elinas, P.; Sucar, E.; Reyes, A.; and Hoey, J. 2004. A decision theoretic approach for task coordination in social robots. In *13th IEEE International Workshop on Robot and Human Interactive Communication*.

Hoey, J.; St-Aubin, R.; Hu, A.; and Boutilier, C. 1999. Spudd: Stochastic planning using decision diagrams. In *Canadian Conference on AI*, 279 – 288.

Jong, N. K., and Stone, P. 2005. State abstraction discovery from irrelevant state variables. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, 752–757.

Laroche, P.; Boniface, Y.; and Schott, R. 2001. A new decomposition technique for solving markov decision processes. In *SAC*, 12–16.

Li, L.; Walsh, T. J.; and Littman, M. L. 2006. Towards a unified theory of state abstraction for mdps. In *Ninth International Symposium on Artificial Intelligence and Mathematics*, 21–30.

Meuleau, N.; Hauskrecht, M.; Kim, K.-E.; Peshkin, L.; Kaelbling, L. P.; Dean, T.; and Boutilier, C. 1998. Solving very large weakly coupled markov decision processes. In *AAAI/IAAI*, 165–172.

Parr, R., and Russell, S. J. 1997. Reinforcement learning with hierarchies of machines. In *NIPS*.

Sucar, L. E. 2007. Parallel markov decision processes. *Studies in Fuzziness and Soft Computing, Advances in Probabilistic Graphical Models* 214/2007:295 – 309.

Tawfik, A. Y., and Neufeld, E. 1994. Temporal bayesian networks. In *TIME*, 85–92.