ASD: A Framework for Generation of Task Hierarchies for Transfer in Reinforcement Learning

Jatin Goyal[1(✉)], Abhijith Madan[1], Akshay Narayan[2], and Shrisha Rao[1]

[1] International Institute of Information Technology, Bangalore, India
{jatin.goyal056,abhijith.m}@iiitb.org, shrao@ieee.org
[2] National University of Singapore, Singapore, Singapore
anarayan@comp.nus.edu.sg

Abstract. We present ASD (Action, Sequence, and Divide), a new framework for Hierarchical Reinforcement Learning (HRL). Present HRL methods construct the task hierarchies but fail to avoid exploration when tasks are to be performed in a particular sequence, resulting in the agent needlessly exploring all permutations of the tasks. When the task hierarchies are used as an ASD framework, the RL agent encounters better constraints, preventing it from pursuing policies that are not valid, thus enabling the agent to achieve the optimal policy faster. The hierarchies created using the methods explained in this paper can be used to solve new episodes of the same environment, as well as similar instances of the problem. The hierarchies generated with an ASD framework can be used to establish an ordering of tasks. The objective is to not only to complete the tasks but also give the agent insights into the sequence of tasks that need to be performed in order to correctly solve a problem. We present an algorithm to generate the hierarchies as an ASD framework. The algorithm has been evaluated on some of the standard RL domains, namely, Taxi and Wargus, and is found to give correct results.

1 Introduction

A standard Reinforcement Learning (RL) [14] problem assumes that an agent starts afresh with zero knowledge about the environment. The agent [1] starts exploring and accumulating rewards for the various actions it takes and devises a policy to solve the problem by either maximizing or minimizing the rewards it gets, depending on the problem. In large environments, a substantial amount of time is spent by the agent to discover the optimal solution.

In HRL [4,15], the agent exploits the environment structure to generate the task hierarchies [10]. These hierarchies can be used to transfer the knowledge from one domain to another which speeds up the overall learning phase, but the agent has no knowledge as to what is the sequence of tasks that needs to be performed, resulting in the agent needlessly evaluating every permutation of the tasks.

L. Cheng et al. (Eds.): ICONIP 2018, LNCS 11303, pp. 313–325, 2018.
https://doi.org/10.1007/978-3-030-04182-3_28

We propose the ASD framework to address the problems. ASD is an approach to HRL which divides actions into three types of nodes: Action, Sequence, and Divide. Scaling down a version of the same problem by generating a task hierarchy, it helps solve problems in large domains. When the hierarchy generated in smaller problems is fed as input to similar larger problems, it brings about a significant improvement over the training cost of the new problem.

The generation of hierarchies as an ASD framework has two steps. First, we access the solutions of the smaller problem which are stored in the form of a Directed Acyclic Graph (DAG) [16] to hold the sequence of operations it needs to perform. By using this structure, we generate a hierarchical sequence of tasks (the Component Hierarchy) for that episode. The DAGs can be generated by simulating a much smaller and simpler instance of the environment and letting the agent perform some episodes by interacting with it (optimal policies of these episodes can be stored in the form of a DAG). Second, we amalgamate results from different episodes of the problem. This is done by merging component hierarchies from different episodes into a single hierarchy.

Algorithm 1 constructs a component hierarchy when a solution to the RL problem is given in the form of a CAT (causally annotated trajectory) [11]. We parse the CAT by identifying the task hierarchies based on which subtasks should be accomplished in what order and construct a hierarchical structure. The inner nodes of the hierarchy consist of high-level tasks. The leaf nodes of the hierarchy are the primitive actions that need to be taken in order to complete the high-level tasks. Some sample component hierarchies are shown in Figs. 3 and 4.

Algorithm 2 merges two component hierarchies into a single hierarchy. Here, the three types of nodes: Action, Sequence and Divide, are used to merge the hierarchies. We start from the root nodes of the two trees and check which type of node is present in it. Based on the type of node we merge the children such that the sequence of execution of the subtasks is maintained in the merged hierarchy.

We tested our algorithms with the Taxi problem [8] and the Wargus domain [11], and accurately generated the task hierarchies. For the case where the environment is stochastic in nature, the problem is solved as a Markov Decision Problem [2]. However, the algorithms mentioned in this paper do not depend on the nature of the environment, i.e., it can either be deterministic or probabilistic [6]. The algorithms can also handle conjunctive goals (both goals can be performed with or without any order) as well as disjunctive goals (the overall task being considered complete if either one of the subtask is completed), which is a significant improvement over the HI-MAT Algorithm [11] which worked on a single DAG.

The key difference between HI-MAT and ASD is that HI-MAT gives task hierarchies using only one CAT, which may not explore the complete dynamics of the environment, whereas ASD uses multiple CATs, makes individual hierarchies, and then merges all the results to get one complete task hierarchy.

2 Background and Related Work

HRL allows the agent to exploit the domain structure by splitting a task into subtasks and solving them, which speeds the overall learning phase [1,13].

To generate a component hierarchy, we require optimal policies for numerous instances. The policies are defined in the form of a CAT [11]. A CAT is a directed acyclic graph that gives information about how the variables of an environment are changed with the actions that are being taken. Consider the Taxi problem in Fig. 1, whose objective is to pickup a passenger from a specific location and then drop off the passenger at a particular destination. The pickup and drop off locations are given as inputs to the agent, the variable *pass.loc* changes at two locations, once during pickup, i.e., when moving from start to pickup and then, at drop off, i.e. when moving from pickup to drop off. Similarly, *pass.dest* is changed only after the agent has picked the passenger and then dropped off at the required destination. These CATs reflect how some of the tasks contains smaller subtasks, as seen in Fig. 1.

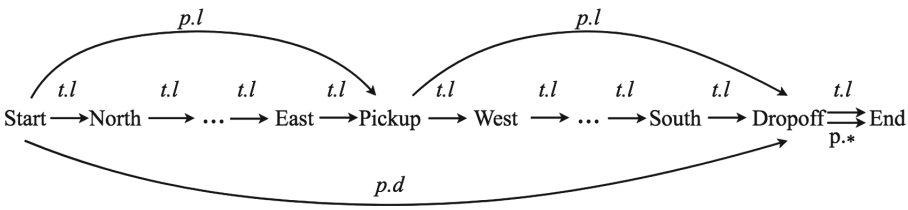

Fig. 1. CAT for taxi problem, where p.l = pass.loc, t.l = taxi.loc and p.d = pass.dest [11]

HEXQ [7], VISA [9], and HI-MAT [11] are some existing approaches to get the required task hierarchies. HEXQ generates the hierarchies based on the frequency of change in values of variables. VISA uses Dynamic Bayesian Networks [3,12] (DBNs) to get the required hierarchies. HI-MAT uses CATs to get the required Hierarchies for MAXQ [4] decomposition. In HI-MAT, the CAT and DBN models are applied to previously solved RL tasks to induce the MAXQ task hierarchies.

3 Component Hierarchies

ASD is an abbreviation of Action, Sequence and Divide, these being the three types of nodes. These nodes are used in the component hierarchy by identifying the sequence of subtasks which need to be completed in order to accomplish a task. Each node has information about the variable that is changing when an action is performed by the agent with the help of a label which is either *Sequence*, *Divide* or *Action*. It also contains a task equation [11] which signifies the task that the node needs to accomplish, the task equation also behaves as an identifier to differentiate it from other nodes.

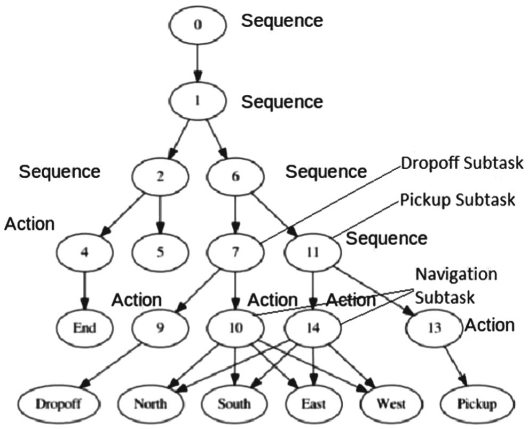

Fig. 2. Component hierarchy for taxi problem [11]

Action

Action nodes signify that there are no more subtasks left, i.e., it is a primitive action. To complete the task signified by this Action node, the agent interacts with the environment using the primitive actions available to it (in the hierarchies). The end state and the start state are provided in the parent node of this subtask. The agent gives the optimal policy and total optimal reward for reaching the end state. For example, in the component hierarchy for the Taxi problem shown in Fig. 2, the navigation task has only four primitive actions (North, South, East, West) which can be used to complete the task.

By marking the Action nodes in the ASD framework, the agent uses only those actions which are available as children of that Action node. This prevents performing some unnecessary computations (like computing costs for `pickup` and `dropoff` for a navigation task), which speeds up the overall process for getting the optimal policy. For example, in Taxi problem, the agent tries not to `pickup` or `dropoff` while on route to the destination.

Sequence

Sequence nodes signify that the subtasks need to be performed sequentially. For example in Fig. 2, the entire problem is marked as a Sequence task, with two subtasks, `pickup` (task 11 in Fig. 2) and `dropoff` (task 7 in Fig. 2), which needs to be performed in that order alone. Like Actions, Sequence nodes also prevent the performing of some unnecessary computations to improve the performance of the system. An agent on coming across a *Sequence* task knows what needs to be performed first, and does not complete the second task before the first task, as in the case of traditional Q-learning agents.

Divide

Divide nodes signify that although the entire task of this node has been split into subtasks, there is no required order for them to be executed, hence the optimal policy can be any permutation of the subtasks in the hierarchy, with the present node as root. For example, in the Wargus domain problem, the agent can mine gold first or chop wood, both being valid ways to do it, but the agent needs to identify the order of the subtasks, so as to get the optimal rewards. In the Taxi problem, there are no Divide tasks since tasks are always in a Sequence and no permutations are allowed.

4 Generating Component Hierarchies

We first automate the discovery of the task hierarchies and generate multiple component hierarchies (smaller task hierarchies) for different instances of the task. Next, we amalgamate these component hierarchies into a single merged hierarchy. The hierarchies once generated need not be computed again; they can be stored and reused repeatedly unless the dynamics of the environment is changed; for instance, by adding a new variable. One such change would be, in the Taxi problem, if we add a new action like Refuel which should be performed before the fuel of the taxi runs out. This would affect the sequence of operations that should be performed and would require retraining, but no retraining will be required if the change is to rearrange the blocked grids in the environment or similar, which does not affect the dynamics of the environment.

Automatic Discovery of Task Hierarchies

The algorithm needs the optimal policies of the source task, stored in the form of a causally annotated trajectory [11] (CAT) to create task hierarchies. Since a single CAT may not use all the actions available to it, we need multiple such CAT which cover the entire dynamics of the environment by using all possible actions in a valid sequence.

An example of a CAT is shown in Fig. 1 for the Taxi problem. These trajectories contain information about the variables that are being changed on every action, and how these change in values of variables tends to completion of the given task.

A CAT is a directed acyclic graph. The CATs are made in such a way that they all have a *Start* and an *End* node. First, the shortest path is calculated from the *Start* node to the *End* node using Depth First Search (DFS) [5]. Also, the in-degree of the *End* node is calculated and stored in the variable *indegree* (line 4). If *indegree* is equal to 2 (line 4), then we mark the entire task as a *Sequence* task since there can be only one way to complete this subtask. If it is more than 2 (line 17), then we mark the entire subtask as a *Divide* subtask since there can be more than one permutation of subtasks that are valid, and if it is equal to 1 (line 11), we mark the subtask as an *Action* subtask.

Algorithm 1. generateASDHierarchy(Ω, ω, S, V) is used to generate a component hierarchy from a CAT. Where Ω is the CAT, ω is the root node of the component hierarchy, S is the start node of the CAT and V is the end node of the CAT.

Input: CAT Ω, ASDNode ω, Start S and End V nodes of the CAT
Output: ASD Hierarchy

```
minCostPath = δ(S, V) = min(W(p) S→ V)
                where W(p) is the Cost of path = Σ_{i=1}^{k} w(v_{i−1}, v_i)
indegree = inDegree(V) //finds the number of incoming edges of a node
if indegree.equalsTo(2) then
    remEdge = minCostPath.remove(0) //remove the first edge from the list of
    edges
    S = remEdge[0]
    V = remEdge[1]
    Create a new node of type "Sequence" and add it to the root node
    generateASDHierarchy(Ω, ω, S, V)
else if indegree.equalsTo(1) then
    Create a new node of type "Action" and add it to the root node
    foreach Action a_i ∈ Ω do
        add a_i to node as a child
    ω.add(node)
else
    Create a new node of type "Divide" and add it to the root node
    V = V.parent
    generateASDHierarchy(Ω, ω, S, V)
```

Algorithm 2. MergeHierarchies(ω_1, ω_2) is used to merge two component hierarchies generated by Algorithm 1. Here ω_1 and ω_2 are the two component hierarchies.

Input: ASDNode ω_1, ASDNode ω_2
Output: Merged ASD hierarchy

```
if (ω1.type.equalsTo("Actions") AND ω2.type.equalsTo("Actions")) then
    ω2.children = ω1.children ∪ ω2.children
if (ω1.type.equalsTo("Divide") OR ω2.type.equalsTo("Divide")) then
    ω1.type = "Divide"
    ω2.type = "Divide"
    foreach child c_i ∈ ω1 do
        if (¬ω2.contains(c_i)) then
            ω2.add(c_i)
if (ω1.type.equalsTo("Sequence") AND ω2.type.equalsTo("Sequence")) then
    foreach child c_i ∈ ω1 do
        if (¬ω2.contains(c_i)) then
            ω2.add(c_i)
        else
            MergeHierarchies(ω1.child[i], ω2.child[i])
return ω2
```

Case 1: If the type of task is *Sequence*, we first remove the minimum path present in the CAT and update the *Start* and *End* nodes with the beginning and ending edges of the removed edge. We create a new node of type *Sequence* and add it to the hierarchy. The function $generateASDHierarchy(Start, End)$ is recursively called for the updated *Start* and *End* nodes. Note that these subtasks are solved in the order of the edges appearing in the shortest path, and when the agent sees the node as a *Sequence* task, it executes these subtasks in that particular order alone.

Case 2: If the type of task is *Action*, then we simply read all the nodes appearing in the path from *Start* to *End*, and add each one of them as a child of this task. The children are a set of primitive actions that need to be performed to accomplish the overall task of this node.

Case 3: If the type of task is *Divide*, then the function $generate\ ASDHierarchy(Start, End)$ is recursively called $indegree - 1$ times. For each time the function is called, the *End* node is updated to the starting node of the edge connecting to the *End* node. This is also explained in Sect. 5. When the agent sees the node as a *Divide* node, it tries all permutations of the subtasks and chooses the optimal policy.

This algorithm ends when it has reached the action path, as there are no more recursive calls from there. The Action path is the sequence of primitive actions that are taken by the agent to generate the optimal policy, and since it is the longest path in the trajectory, it is always the last one selected for building the hierarchies.

Incomplete Hierarchies

Algorithm 1 relies on optimal policies stored for making the hierarchies, but an optimal policy stored may not contain all the primitive actions in the environment. It may not even contain an entire subtask because that was not needed for that particular optimal policy though it might be needed for other episodes of the same environment. Therefore the hierarchies should cover the entire dynamics of the system. For example, in Fig. 3, the Hierarchy is of a Taxi problem which does not contain the complete dynamics of the Taxi environment, as it does not contain "West" as a primitive action, also for subtask `pickup`, only "North" is available for navigation, which shows incomplete information about the environment. To overcome this problem, we give Algorithm 2 for merging multiple hierarchies of the same environment.

The function $MergeHierarchies()$ in Algorithm 2 takes two hierarchies from different instances of the same environment as inputs. The function recursively compares the nodes and their children. First, the function compares the type (Action, Sequence or Divide) of both the root nodes.

Case 1: If both nodes are of type *Action*, the function $MergeHierarchies$ merges the primitive actions list of both the root nodes and store in the primitive actions list of the root node in *Hierarchy2*.

Case 2: If either of the nodes is a *Divide* node, mark the node in *Hierarchy2* as *Divide*. If a subtask of the root node in *Hierarchy1* is found in *Hierarchy2*, call *MergeHierarchies(Hierarchy1, Hierarchy2)* using the similar nodes as root. For every child in the list of subtasks of the root node in *Hierarchy1*, not present in children list in *Hierarchy2*, append those tasks to the children list of the root node of *Hierarchy2*.

Case 3: If both the nodes are of type *Sequence*, then children of both the nodes are compared. Since these tasks have to be executed in a sequence, the comparison done in the reverse order of execution. If the list from *Hierarchy1* contains some subtask which is not present in *Hierarchy2*, these subtasks are appended to the start of the list in *Hierarchy2* (because the function gives *Hierarchy2* as output). Also for the subtask found identical in both the lists of subtasks, call *MergeHierarchies(Hierarchy1, Hierarchy2)* using the identical node as the root nodes.

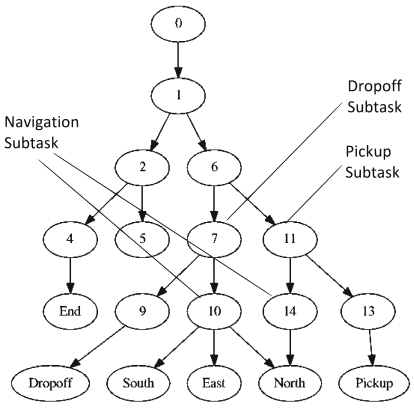

Fig. 3. Hierarchy 1

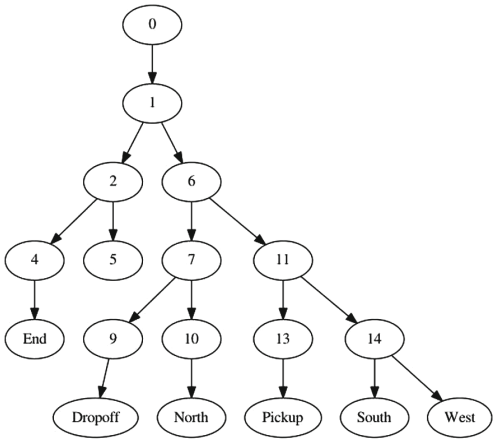

Fig. 4. Hierarchy 2

5 Evaluation of ASD in Standard RL Domains

Taxi Problem

Consider the 5×5 single agent Taxi Cab domain [11]; the CAT for a solved episode is given in Fig. 1. As given in Algorithm 1, for the *Start* and *End* node, the shortest path is calculated and it is stored; in this case, it is *Start* $\rightarrow$ *dropoff* $\rightarrow$ *End*. Consecutively, the number of edges incoming to the end node is also calculated, i.e., the in-degree of the end node which is 2 in this case. According to Algorithm 1, we mark this task as *Sequence* node and divide the entire problem into two subtasks, 1. *Start* $\rightarrow$ *dropoff*, 2. *dropoff* $\rightarrow$ *End*

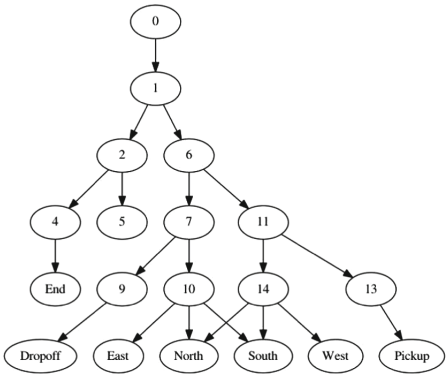

Fig. 5. Merged result for Hierarchy 1 in Fig. 3 and Hierarchy 2 in Fig. 4

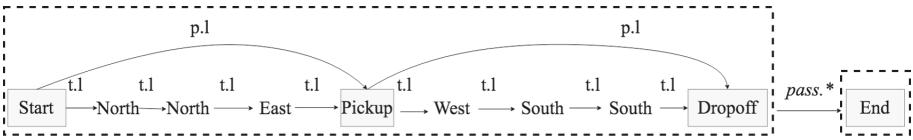

Fig. 6. Dividing tasks in a CAT according to Algorithm 1

as shown in Fig. 6. According to Algorithm 1, the edge *Start* → *dropoff* is removed and *generateASDHierarchy*(*Start*, *dropoff*) is called. Algorithm 1 is followed, finding shortest path (with *dropoff* as the end node), which is *Start* → *pickup* → *dropoff*, and the in-degree of the end node is calculated which is 2 in this case. We then mark this subtask as *Sequence* and further divide it into subtasks as shown in Fig. 7.

The edge *Start* → *pickup* is removed and *generateASDHierarchy* (*Start*, *pickup*) is called recursively. Now the in-degree of the end node is 1 and as a result, this subtask is marked as *Action* node, and all the actions occurring on the path from the *Start* node to *End* node are added to this node as its children. After *Actions* there are no more recursive calls, hence *generateASDHierarchy*(`pickup`, `dropoff`) is called and the same steps are followed, and consecutively, component hierarchies are formed. Similar steps are followed and the entire task of picking up and dropping off the passenger is divided into subtasks, which are executed according to the node type, as is mentioned in Sect. 3.

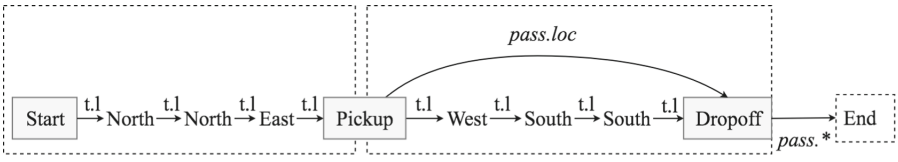

Fig. 7. Dividing tasks of Fig. 6

Similarly, two task hierarchies are generated, as shown in Figs. 3 and 4, from two different CATs. It can be observed that neither of them contains all the primitive actions available to the agent. Algorithm 2 gives the protocol to merge the knowledge of both the hierarchies. Starting from the root node (0) in both the trees, nodes at each level are compared in both hierarchies. If both the nodes are *Sequence* then we move to the next level, and if one of them is *Divide* then the resulting node is also *Divide*. In the case of Figs. 3 and 4, both are *Sequence*, so we move to node 1 in both the hierarchies; both are *Sequence*, so we move to the next level. This level contains multiple nodes in both the hierarchies; first the union of both the nodes is done and stored as the result. If two nodes contains the same task equation, then $MergeHierarchies(node1, node2)$ is implemented, which follows the Algorithm 2 again. When the iteration reaches the Action nodes, the union of the primitive actions in both the nodes is done resulting in a node which contains actions from both the nodes and is stored in the resultant hierarchy. As it is seen in Fig. 5, all the primitive actions in *Hierarchy1* and *Hierarchy2* in Figs. 3 and 4 are merged to form action nodes in Fig. 5.

Wargus Domain Problem

Consider a 5×5 grid for Wargus resource gathering-problem [11] where the agent is the peasant and its objective is to chop wood, mine gold and deposit it at the City center. The optimal policy is the steps taken to complete the task with maximum reward.

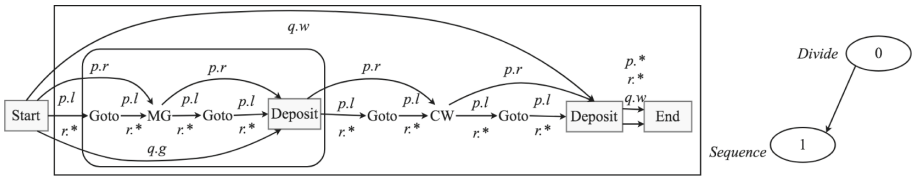

Fig. 8. Solved RL CAT for Wargus domain

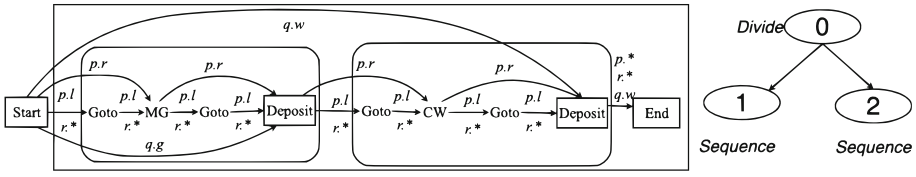

Fig. 9. Dividing tasks in Fig. 8 and making hierarchies according to Algorithm 1

For generating the component hierarchies for this problem, access to solved instances of Wargus domain are required, an example is shown in Fig. 8, here $p.l$ stands for the peasant's location, $p.r$ is the peasant's resource, $r.w$ and

$r.g$ stands for region to chop the wood and region to mine for gold. $q.w$ and $q.g$ stands for quota for wood and gold respectively. Similar to the previous example, first the shortest path between *Start* and *End* is calculated, as also the number of edges connecting to the *End*, which is 3 in this case. As a result this task is marked as *divide*, the edge $Start \rightarrow Deposit$ is removed and the procedure $generateASDHierarchy(Start, Deposit)$ is recursively called. The task hierarchies are shown in Figs. 8, 9 and 10. Note when calling $generateASDHierarchy(Start, Deposit)$ for the second *Deposit* node, the *Start* node is changed to *Goto* in $Deposit \rightarrow Goto$. This is because while dividing the task into subtasks, first task was completed at the first *Deposit* node, and the second task started from there, but unlike the Taxi problem, there is no edge from first *Deposit* node to the second one, because both the goals can be achieved in either order. So when the agent uses the merged hierarchy for solving problems, and reaches the *Divide* node, it evaluates rewards for both the actions and then decides which one to choose based on the policy that gives the higher reward. Before $generateASDHierarchy(Start, Deposit)$ is called, the edge $Start \rightarrow Deposit$ is removed, after which the shortest path is calculated, and number of edges connecting to the *End* node, which is 2 in this case, as a result of which, this task is marked as *Sequence*, as shown in Figs. 8 and 9, similar to the Taxi Cab domain problem, and the final task hierarchies are generated as shown in Fig. 10. Also when a task is marked as *Divide*, the subtask associated with the last edge of the shortest path, is not used for any input in task hierarchies, in this case, the tasks associated with both the $Deposit \rightarrow End$ edges are never used to make the hierarchies, because these edges signify the completion of that task.

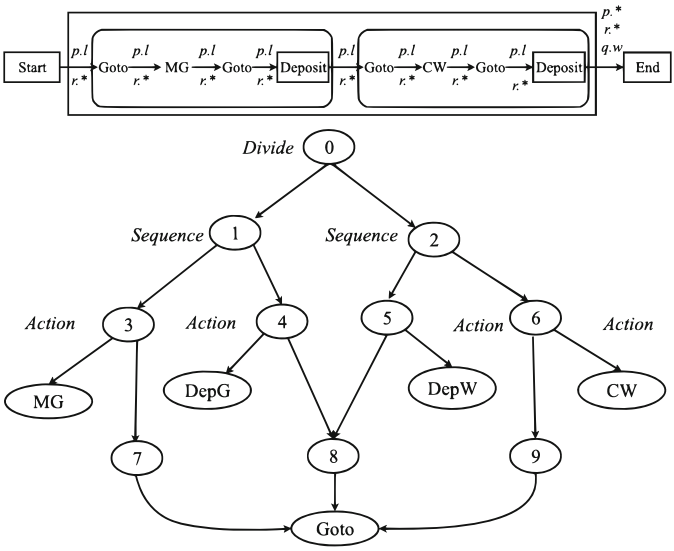

Fig. 10. Dividing tasks and generating task hierarchies

In certain instances, it is possible that the CAT may contain only one of the subtasks (chop wood or mine gold). In such cases, the component hierarchies would be incomplete. Also Algorithm 1 would mark this subtask as a *Sequence* node. But when this component hierarchy is merged with a component hierarchy generated for a CAT shown in Fig. 8, then according to Algorithm 2, it is clear that when either of the two tasks at the same level are *Divide* while merging, then the resulting task will also be a *Divide* task. The main function of Algorithm 2 is to learn about the tasks and the actions available from multiple task hierarchies.

6 Observation and Conclusion

Both Algorithms 1 and 2 were tried on the Taxi domain problem and the Wargus problem, and the result is that the hierarchies obtained contained the subtasks in the right order. The CATs required to generate the Hierarchies may come from a smaller version of the environment. For example, in the Taxi domain problem, the required simulations were to be done on a 50×50 grid, and we know the optimal policies need to come from the same environment. However, we noticed, that if we input CAT from a grid of a 5×5, the hierarchies generated are in accordance to the 50×50 grid. This is because irrespective of the size of the domain, the sequence of actions performed to achieve a task would remain the same.

We presented an approach to automatically induce subtasks, which cover the entire dynamics of the system, from solved RL problems. We presented the algorithm to generate a component hierarchy from the CATs of all the solved instances of the problem. This will speed up the process of finding solutions in other instances of same domain. Also, these hierarchies are reusable.

References

1. Andre, D., Russell, S.: State abstraction for programmable Reinforcement Learning agents. In: Association for the Advancement of Artificial Intelligence, vol. 112, pp. 119–125 (2002)
2. Bai, A., Srivastava, S., Russell, S.: Markovian state and action abstractions for MDPS via hierarchical MCTS. In: Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, pp. 3029–3037. AAAI Press (2016)
3. Cao, F., Ray, S.: Bayesian Hierarchical Reinforcement Learning. In: Pereira, F., Burges, C.J.C., Bottou, L., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems 25, pp. 73–81. Curran Associates Inc, New York (2012)
4. Dietterich, T.G.: Hierarchical Reinforcement Learning with the MAXQ value function decomposition. J. Artif. Intell. Res. **13**, 227–303 (2000)
5. Franciosa, P.G., Gambosi, G., Nanni, U.: The incremental maintenance of a depth-first-search tree in directed acyclic graphs. Inf. Process. Lett. **61**(2), 113–120 (1997)
6. Getoor, L., et al.: Introduction to Statistical Relational Learning. MIT press, Cambridge (2007)

7. Hengst, B.: Discovering hierarchy in Reinforcement Learning with HEXQ. In: Proceedings of the Nineteenth International Conference on Machine Learning, ICML 2002, pp. 243–250. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2002)
8. Jardim, D., Nunes, L., Oliveira, S.: Hierarchical Reinforcement Learning: Learning sub-goals and state-abstraction. IEEE, July 2011
9. Jonsson, A., Barto, A.: Causal graph based decomposition of factored MDPS. J. Mach. Learn. Res. **7**, 2259–2301 (2006)
10. Knoblock, C.A.: Hierarchical problem solving. In: Knoblock, C.A. (ed.) Generating Abstraction Hierarchies. The Springer International Series in Engineering and Computer Science (Knowledge Representation, Learning and Expert Systems), vol. 214. Springer, Boston (1993). https://doi.org/10.1007/978-1-4615-3152-4_3
11. Mehta, N., Ray, S., Tadepalli, P., Dietterich, T.: Automatic discovery and transfer of MAXQ hierarchies. In: Proceedings of the 25th International Conference on Machine Learning, pp. 648–655. ACM (2008)
12. Ngo, V.A., Ngo, H., Wolfgang, E.: Monte carlo bayesian Hierarchical Reinforcement Learning. In: Proceedings of the 2014 International Conference on Autonomous Agents and Multi-agent Systems, AAMAS 2014, pp. 1551–1552. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2014)
13. Sutton, R., Precup, D., Singh, S.: Between mdps and semi-MDPS: a framework for temporal abstraction in reinforcement learning. Artif. Intell. **112**, 181–211 (1999)
14. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An introduction, vol. 1. MIT press, Cambridge (1998)
15. Tadepalli, P., Dietterich, T.G.: Hierarchical explanation-based Reinforcement Learning. In: In Proceedings of the Fourteenth International Conference on Machine Learning, pp. 358–366. Morgan Kaufmann (1997)
16. VanderWeele, T.J., Robins, J.M.: Directed acyclic graphs, sufficient causes, and the properties of conditioning on a common effect. Am. J. Epidemiol. **166**(9), 1096–1104 (2007)