Proceedings of the Twentieth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2024)

Using Natural Language to Improve Hierarchical Reinforcement Learning in Games

Dave Mobley, Adrienne Corwin, Brent Harrison
Department of Computer Science
University of Kentucky
Lexington, KY 40508
dave.mobley@uky.edu, adriennecorwin@gmail.com, harrison@cs.uky.edu

Abstract

This work investigates how natural language task descriptions can accelerate reinforcement learning in games. Recognizing that human descriptions often imply a hierarchical task structure, we propose a method to extract this hierarchy and convert it into options – policies for solving sub-tasks. These options are generated by grounding natural language descriptions into environment states, which are then used as task boundaries to learn option policies either by leveraging prior successful traces or from human created walk-throughs. As part of our work, we discuss a method to generate natural language from prior knowledge as a precursor step to use when natural language descriptions are unavailable. We evaluate our approach in both a simpler grid-world environment and the more complex text-based game Zork, comparing option-based agents against standard Q-learning and random agents. Our results demonstrate the effectiveness of incorporating natural language task knowledge for faster and more efficient reinforcement learning across different environments and Q-learning algorithms, including tabular Q-learning and Deep Q-Networks (DQNs).

Introduction

An effective way for a Reinforcement Learning (RL) agent to solve a game is for it to break it down into sub-tasks. This approach, known as Hierarchical Reinforcement Learning (HRL), involves creating a hierarchy of sub-tasks and allowing the agent to learn which sub-tasks to choose. A key challenge in HRL is the automatic identification of effective sub-tasks for the agent. Traditionally, human authors have manually created hierarchies and policies for RL algorithms to apply to specific problems such as games. We have explored automated methods for creating hierarchies, such as using random options generated from states (McGovern and Barto 2001), choke-points (Şimşek and Barto 2004a), required pathways (Şimşek and Barto 2004a), reachability (Dai, Strehl, and Goldsmith 2008) and a statistical model using prior successful traces (Mobley, Goldsmith, and Harrison 2020). A significant challenge with some of the prior techniques are that they either do not generate options tailored to the game or they face scalability issues in very large domains, especially where choke-points span many states.

Copyright © 2024, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Our goal was to develop an approach that uses natural language descriptions to provide instructions on navigating an environment and to map that natural language to starting and ending states, along with developing policies for navigating between those states and ideally these options should scale in larger environments. We call this collection of starting and terminal states with a variable length partial policy an *Option*. Options easily define a ready-to-use partial policy for an agent to use during an RL problem, potentially reducing the learning effort to achieve an efficient solution. Options also tend to be easy to use with various common RL algorithms because they can work essentially in place of simple actions(Sutton, Precup, and Singh 1999).

In this paper we propose using natural language to describe starting and ending states in a task and then creating partial policies from prior traces or natural language walk-throughs. We have chosen to focus on problems where a significant portion of the agent's actions involve navigation, as this concentrates on a small subset of all possible actions, which has a large impact on the agent's overall success and makes the grounding exercise easier.

Natural language possesses an inherent capacity to abstractly categorize states and actions based on similarities. This makes it an effective tool for identifying how to start a task and which actions to take to execute that task. The sequential nature of instructions in natural language also creates boundaries for when one task ends and another should start. Imagine a simple task like going to the bank to withdraw money. Although it seems straightforward, the task involves several steps: leaving your house, getting in your car, driving to the bank, using the ATM, and putting the money in your wallet. Each step can be considered a sub-task, with its own starting and ending states. For example, the first sub-task of going to your car might start when you are at home and have not yet retrieved the money. Once you get up, walk to the garage, and enter your car, the first sub-task ends, and a new sub-task, driving to the bank, begins. Each sub-task terminates when the next one can start.

People use natural language as an effective tool for describing just enough information to include states in a set without including unnecessary details. Our state set representation doesn't need to consider factors like the time of day, the amount of fuel in the car, or the specific home or bank you are going to. Instead, it represents the sub-task in

an abstract way without defining the concrete states they represent. Our goal is to use this ability to describe abstract collections of states to construct options that can be utilized as part of an RL algorithm. To create these options, we can either consider an existing natural language description on how to complete the overall task, or look at prior cases where the whole problem has been successfully completed but has no natural language description. We can use this prior knowledge to describe the states of the problem in natural language, and construct an ordering of options from that. This provides two key pieces of information. The collection of abstract sets of states that will compose our options and, because these are successful completions, we can use the partial policies from these prior traces as representative policies for our newly constructed options, saving some of the effort required to develop a policy for each new option.

Background and Related Work

In this section, we will discuss related topics in natural language guided reinforcement learning and hierarchical reinforcement learning. We will also cover relevant background on Markov decision processes and semi-Markov decision processes.

Background on Markov Decision Processes and Semi-Markov Decision Processes

For our approach, we model our game environments as *Markov Decision Processes* (MDPs). An MDP is an environment where the state S of the environment is represented as a tuple (S, A_s, R_a, T_a). The action space A_s is a set of all actions available in a given state $s \in S$. The reward R_a is a value received immediately when transitioning to a next state, s' because of action $a \in A_s$. For each *episode*, the agent traverses different states over discrete time steps reaching either a terminal state or time condition. We denote a state s at time step t as s_t in an episode. Because MDPs are stochastic in nature, there is a transition probability T_a such that at time t given state s_t and action a_t there is a probability distribution over the next states. This can be written as $T_a(s_t, s_{t+1}) = Probability(s_{t+1} = s' | s_t = s, a_t = a)$.

Given an MDP, a popular technique used for hierarchical reinforcement learning is called *options* (Sutton, Precup, and Singh 1999). There are three parts to an *option*: (i) a set of starting states, I, (ii) a policy of actions π, (iii) a termination probability β. An agent, once entering an option, it executes the option following the option policy until it terminates as per the conditions of β. Options can require a variable length of time to complete. To be used with an MDP framework, so that one option may call another, the problem must be adjusted so that it is now a Semi-Markov Decision Process (SMDP), such that rewards over a time interval can be applied. Doing this, allows options to have variable time length and accrue rewards over those options. All MDPs can be viewed as options-based SMDPs because every state and its accompanying actions meet the criteria for being options. The state itself is the entry state where $\{s\} = I$. The policy for each state is that which is learned by taking one of the possible actions $a_s \in A$ for that particular state, and the terminating probability $\beta = 1$, meaning it terminates after taking exactly 1 time step. These types of options are called *primitive* options. Because this framework allows options to call subsequent options and provide rewards for the option, we now have the ability to structure hierarchies of sub-tasks based on options.

Related Work on Natural Language Guidance in Reinforcement Learning

As part of our goal, we seek to utilize procedural knowledge embedded in natural language to extract hierarchical task information. Natural language has shown a great deal of promise in other areas of reinforcement learning. The area most closely related to our own is *natural language guidance*, where natural language provided by humans is used to instruct agents on how to complete a complex task (Goyal, Niekum, and Mooney 2021; Tasrin et al. 2021). In some instances, natural language can be used to identify dangerous states that the agent should avoid (Yang et al. 2021). The difference between this work and our own is that our work seeks to use procedural knowledge encoded in natural language to extract hierarchical information. In this past work, natural language is often mapped to single actions or states, rather than sequences of actions and states.

Related Work in Hierarchical Reinforcement Learning

Hierarchical Reinforcement learning techniques have been used to improve the performance of agents in RL problems by providing levels of abstraction to organize subtasks There are a few common algorithms such as using abstract machines (Parr and Russell 1998), MAXQ, and options (Sutton, Precup, and Singh 1999). HRL has been shown in general to out-perform Q-learning for the problems at hand, but still requires the creation of the abstract hierarchy to work effectively. Typically, these hierarchies are hand specified, but this method does not scale to more complex environments. To that end, there has been an increased interest in automatically learning hierarchical information for use in reinforcement learning. Prior work has explored using randomly generated options (Stolle and Precup 2002), using prior knowledge to find candidate states for starting and terminal states of options (McGovern and Barto 2001), and by using choke-points, required pathways, or reachability (Şimşek and Barto 2004b; Şimşek, Wolfe, and Barto 2005; Dai, Strehl, and Goldsmith 2008). One statistical approach finds commonality in traces with Association Rule Learning (ARL) (Mobley, Goldsmith, and Harrison 2020). This approach, though promising, has the challenge that it may be computationally infeasible to calculate ARL support for the myriad combinations of states, especially if chokepoints span a large number of states. Because of this, we look at a new method utilizing natural language to create a hierarchy of options and utilizing prior successful traces as the policy for those options.

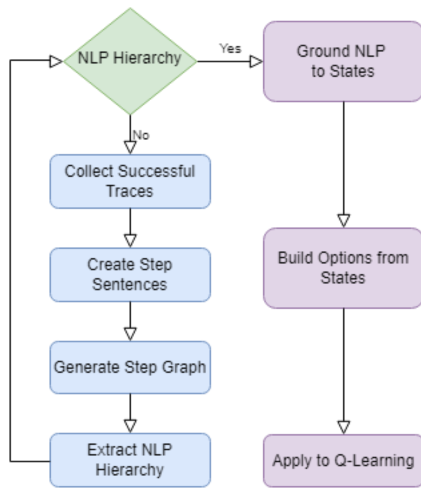

Figure 1: Steps of the algorithm.

Methodology

Our approach to building a hierarchy for a problem using natural language begins by gathering a natural language description of how to successfully complete the overall task. Depending on the task, we may already have a description, often provided beforehand by a human as a set of steps or a walk-through. If no description is available, we can create one by analyzing a collection of prior successful traces and describing similar states attained on successfully completing the game. In practice, the similarity of state will be based on the location of the agent in the game.

Once we have the natural language description, the next step is to ground these descriptions into complex options, converting the natural language into a form that the agent can recognize and act on. A complex option has a starting set of states, a terminating set of states, and can take a variable length of time to complete, unlike a primitive option, which has a starting state and is a simple action taking only one time step. In our examples, we use descriptive names for locations, such as "gem 1" or "library", and map these words onto abstract states representing all states in that location. These states will represent the start of the option. An option is considered terminated when the agent transitions to a state that marks the beginning of another complex option or the end of the task. We can use prior trace moves that transition between the start and ending states of an option to define a policy for the option.

After constructing the options, they can be integrated into different variations of Q-Learning, such as tabular Q-Learning or Deep Q-Networks (DQNs). This hierarchical structure allows an agent to perform complex tasks using natural language descriptions effectively while still exploring for newer and possibly better policies. A graphical representation of the algorithm's flow can be seen in Figure 1

Gathering Natural Language Descriptions

Sometimes we already have a natural language description of the steps needed to complete a task, but this may not always be the case. If we do not have a natural language description we can manufacture one using successful traces from the game. We want our natural language to be similar to the way people would describe taking actions with their reasoning in the game. When explaining the reasoning behind taking certain game actions, people often use goal-oriented language. For example, in one of our test environments an agent needs to collect a key in order to open a door. If an agent moves closer to a key, a possible explanation for that may be "I want to pick up the key." Another possible explanation could be "I want to eventually open the door." Our task is to construct natural language that follows this pattern.

We first assume there is a corpora of successful traces. If there are not, an agent can play the game using an algorithm of choice, or can be assisted by people to successfully complete the game. Traces are recorded as transition steps called *moves*, of the format <*starting state, action, next state*>. Note that these traces do not necessarily contain optimal paths through a game level, but as noted in Stolle et al. (2002) where random options were used, the agent will learn to not use a poor option when a better alternative is recognized.

Using our successful traces, we create a grammar that will generate a simple sentence for each transition of the trace. For our games, the grammar rules were fairly straightforward, creating sentences that describe an immediate outcome or borrow a description from a future outcome such as "I am going to the trapdoor", "I picked up the lamp", or "I will get the sword." We refer to this grammar as our *game grammar*. By creating sentences this way, we mimic the way people describe future goals while in a given state without having to develop a full natural language walkthrough for the game.

We would like to use that knowledge to start constructing hierarchy. Our first step is to build a sequence-to-sequence network to recognize a state and be able to output expected (next) or future goals. We teach a sequence-to-sequence network (Sutskever, Vinyals, and Le 2014) to learn relationships between individual states/actions and the natural language used to describe them using the transitions and game grammar. Specifically, the input to the sequence-to-sequence network is a representation of state and action information, and the desired output of the network is a set of possible natural language utterances describing them. By learning these relationships, the sequence-to-sequence network learns how individual actions taken potentially align with future actions that can be taken. By training a sequence-to-sequence network on such data, the network would learn that a given state and action might mean both picking up the key and opening the door. We limit the trained sequence-to-sequence network to generating 5 utterances for each state and action taken in a trace.

Forming Natural Language Relationship Graph

Using the sequence-to-sequence network described above, we can produce candidate natural language goal information for a series of states and an actions in a game environment. The issue is that we do not necessarily know if an ordering exists between this goal information. Recall the

example used above about the key and the door. Even if the sequence-to-sequence network correctly identifies that moving towards a key has potential goals of picking up the key and unlocking the door, it does not know that picking up the key is required to eventually open the door.

To better determine an ordering of goals, we choose to represent the relationship between goals in a game world as a directed graph, where nodes are goals and edges are dependence relationships. The first step in constructing this graph is to ground the natural language generated by our sequence-to-sequence network into a node in the graph. We do this by clustering semantically similar phrases into the same nodes based on keywords and patterns.

On larger problems, alternative familiar NLP clustering algorithms could be used to allow for scaling to bigger sets of nodes.

Working trace-by-trace and phrase-by-phrase we build a directed graph of the nodes that shows the precedence for how one phrase comes before another in a trace, represented by an edge from the preceding node to the next node. Working backwards, this graph can be considered a dependency graph because later nodes could be considered to depend on prior nodes having occurred. To build the graph, using prior traces, we traverse a trace, matching a state with a phrase. If the phrase is not on the graph as a node, we include it. This phrase is considered the start phrase. We continue along the trace until we find another state representing a different phrase. If that phrase is not on the graph, we include it as well. This phrase is considered the terminal phrase. A directed edge is added from the start phrase node directed at the end phrase node. The end phrase node is now considered the next start phrase node and the process continues until we reach the end of the trace and there are no future nodes.

An example graph is shown in Figure 2 with the directed mappings from phrase to subsequent phrase.

Once we have this graph, we remove edges that occur infrequently and perform a transitive reduction to accurately represent dependence relationships in the graph. A pruned graph is shown in Figure 3.

Creating a Hierarchy from a Directed Graph

In this graph, the existence of an edge between nodes indicates that an option can be constructed that describes how to transition from one goal to the next. The issue is that each goal node grounds to potentially many states. This is because each goal was extracted based on natural language descriptions which potentially abstract many states into one idea. Thus, we first need to ground each goal node back into concrete states. To do this, we construct rules based on the states in the training set associated with each goal. For example, consider a possible goal *has egg*. We can identify the set of states in the training set that are associated with this goal. In our work, we represent a state symbolically using a vector containing various pieces of information about the state. This enables us to use strategies such as frequent itemset mining (Borgelt 2012) to extract elements common to each state in this set. In this example, each state in the training set associated with this state should have a field associated with having an egg object set to 1. Thus, this could become a general rule that needs to be true for a state to be associated with this goal. This enables us to map abstract goals onto concrete states. We do this for both the head of an edge (the eventual starting states of an option) and the tail of an edge (the eventual terminating states of an option). With the mapping done and with a list of starting node sets and terminating node sets, we can now consider the issue of converting starting and terminating sets into options for HRL.

Applying a Hierarchy to an MDP

As stated before, an *option* is composed of three parts: the first is a set of starting states, S, the second is a set of terminating conditions, which in our case will be ending states T, and some policy that ideally transitions from start to termination. To complete some sub-task of work, once an agent enters a starting state, it must execute until some time threshold has elapsed or the agent reaches one of the terminal states $t \in T$ for that option. At its simplest, every MDP can be viewed as an options-based SMDP because every state can be described as an option that takes exactly one time step t. The state itself is the entry state where $\{s\} = S$. The terminal states T are those that the agent can transition to directly from state s. Because any state transition can be mapped to an option in this way, these can be called *primitive* options, conversely, options that can take many time steps are called *complex* options. The creation of options thus builds a hierarchy in an MDP where an option is a subtask where once an option has started, it runs until it reaches a terminal state, or the amount of time steps it has been allotted is exceeded.

When we apply the options technique to a problem, all primitive options are available, but new options can be introduced so an agent has the opportunity to learn alternate, and possibly better, routes to the goal. The purpose of introducing a hierarchy is to provide guidance to the agent so that it focuses on sub-tasks that will consistently achieve a goal quicker than from simple Q-learning.

Now that we have our abstract states, we can create options by identifying relationships between them. Specifically, if one abstract state leads to another in the graph, we can create a rule to represent the starting state or generate a list of all possible concrete representations. We can do the same for potential terminal states. By combining the starting and terminal states, we can begin to build an option. In practice, we create an option and either assign a list of starting states or define a function to determine if a state is a starting state. We follow a similar process for terminal states. To create an initial policy, we can analyze our prior traces to find examples of starting and terminal states, and use the moves from those traces as a partial policy. Once an option has starting and terminal states and a policy, we add it to our collection of options to apply to the problem. After creating all new options, we can apply one of our reinforcement learning algorithms that can utilize options, allowing us to leverage our extracted prior knowledge.

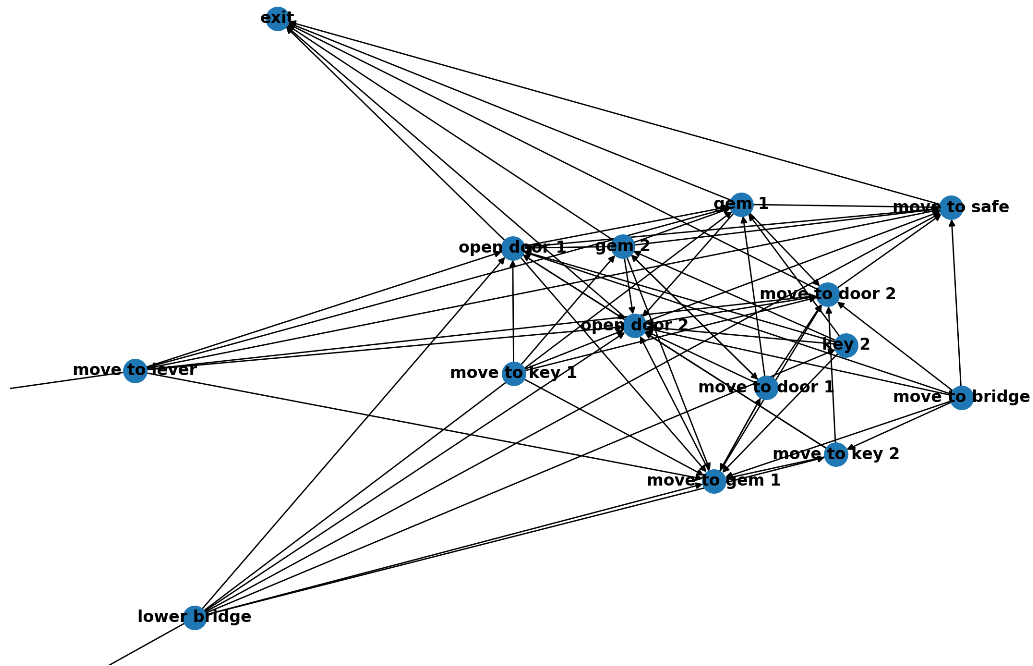

Figure 2: An example of a raw directed graph based on natural nodes.

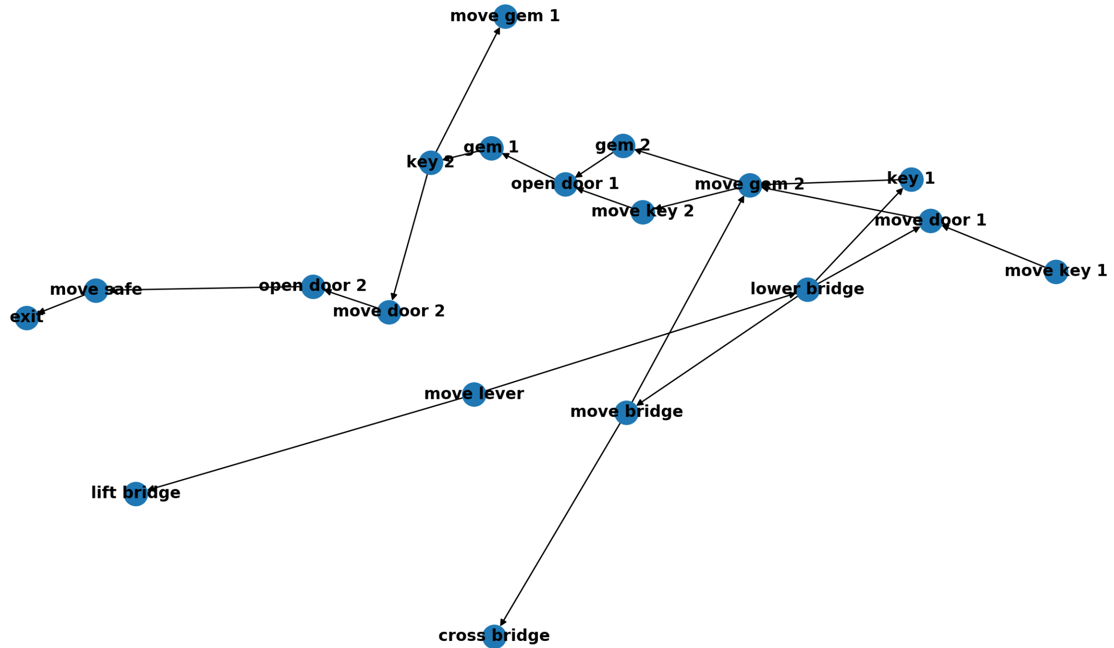

Figure 3: An example of a directed graph based on natural nodes where ancestor dependency and low-count outlier dependencies have been removed.

Experiments

We designed two experiments to test our approach and demonstrate how a hierarchy can be built and applied in game environments.

The first experiment is a modified grid-world game called Gemworld. In Gemworld, an agent must collect two unique keys to open two doors, find two gems, and reach a safe location with the gems. This game is designed to show that our algorithm can learn a hierarchy based on natural language descriptions of successful paths through the game. Additionally, the gems can be collected in any order, introducing the element of task flexibility.

For the second experiment, we chose a text-based game called Dungeon or Zork (Infocom 1980). Zork accepts text commands from the user, allowing them to direct their avatar to perform various tasks. Points are earned by exploring new locations, finding objects, and storing trophies. The game has a maximum score of 585 points, not including bonus points, but it requires a long and specific sequence of actions to achieve this score. The game ends if the agent dies or takes too many steps.

In both experiments, we compare our approach against random agents and those using standard Q-Learning algorithms. In Gemworld, we also include an agent that utilizes options derived via ARL. For Zork, we evaluate two types of options: one focused solely on navigation and another that navigates and receives a reward upon completion. We wanted to test both types of options because larger problems using DQNs often train faster when an option has an associated reward.

These experiments will showcase how our approach can handle different game environments and validate the effectiveness of our hierarchical method.

Gemworld Environment

The Gemworld environment is a 10x10 grid, shown in Figure 4, where an agent represented as the blue circle, can start at any location and must collect both gems and reach the exit, which is represented by a green square. There are several objects in the environment. The environment includes gems (represented by diamonds in the figure), doors (brown squares), keys (the letter k), and levers (yellow squares). Keys are required to open doors, and the levers raise and lower a bridge that enable crossing the river (blue squares). Black circles are holes that result in death if entered. Grey squares represent impassable walls. The agent can move in any cardinal direction. By moving into spaces where items are, the agent will interact with that item. For simplicity, we still represent this as movement, meaning that the agent only has four actions in this environment.

Each step in the game has a cost of 1 point. Gaining gems, keys, and exiting the game grants points for each successful goal, where gems and keys provide 20 points, and winning the game gives 100 points. The Gemworld game is a deterministic game, so no randomness occurs when an action is taken. For an episode, we set a move limit to 200 maximum steps and each experiment plays the game for 1000 episodes. A state in the Gemworld environment is represented as a vector: $(X, Y, G1, K1, K2, D1, D2, G2, B)$, representing the (x, y) coordinates of the agent and flags representing whether the agent has collected gem 1, key 1, key2, gem 2, opened door1 or door2, and if the bridge is lowered.

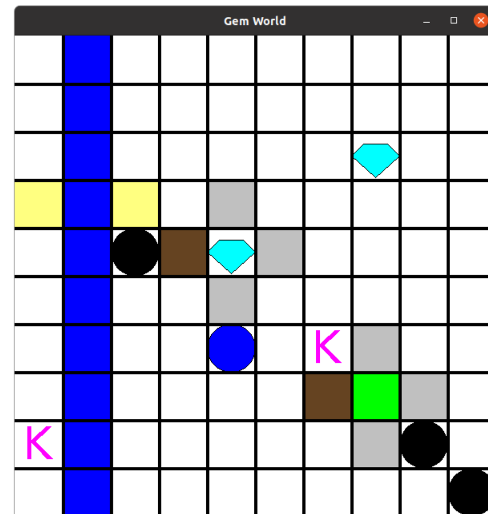

Figure 4: Gemworld Game

Zork I Game Environment

Zork I is a text-based game played in steps. Each step allows players to choose an action, type it in, and observe the environment's response based on that action. The game offers hundreds of possible commands, involving verbs possibly followed by direct or indirect objects. To manage training on single-system GPUs more effectively, we've limited the action space to phrases used in successful game traces, significantly increasing the number of viable commands from the previous environment's four actions. We've also reduced episode length to enable more episodes.

Initially, we used the Jericho project's game interface (Hausknecht et al. 2019). For increased speed, we compiled the original code into a dynamically linked library and interfaced directly with it (Dietrich et al. 2024).

In our setup, the agent is granted access to all necessary actions for the first 150 game steps. This allows the agent to learn early sub-tasks such as collecting the egg (worth 5 points), entering a house (worth 10 points), and navigating to the cellar (worth 25 points). Successfully completing these tasks requires extra actions like opening a window to enter the house or lighting a lantern before entering the cellar. Without performing certain actions, like entering the cellar without grabbing the lamp or lighting it, the agent could effectively become stuck where the only option is to move and die, costing the agent 5 points.

Reaching the cellar successfully grants the agent a total of 40 points. Additional tasks include stealing a painting, robbing a bank, and finding coins in a maze. A human player using a walk-through could potentially achieve up to 106 points within the first 100 steps, assuming they avoid stochastic events like dying from the troll or thief.

Given Zork's text-based nature, its state isn't easily evaluated or map-able. To address this, we extract relevant game-state data from the save-game provided by the source code, converting it into a numeric tensor. This data includes the locations of all items, the player and NPC positions, and flags indicating the state of items (e.g., whether a lamp is on or a door is open). Currently, the agent has full visibility of the game state, making the game a stochastic MDP.

Experimental Set Up

Recall that our tests cover four agents in each environment. For both environments we use a random agent that chooses one of the possible moves for any given state randomly. We also use a Q-Learning algorithm to learn how to successfully navigate the environment as a baseline agent.

For Gemworld, we use the ARL-Options algorithm which calculates a set of options based on prior traces and also our NLP-Options algorithm which creates options based on the algorithm described in the Methodology section. The way it is designed, the ARL agent will only construct at most one option per state. To make our comparison more fair, we limited our NLP agent to construct at most one option per state. In states where multiple options could be constructed, we chose the option that contained the fewest number of steps from starting state to terminal condition. We do this so that our algorithm won't flood a given state with an overwhelming number of option choices.

For Zork, we use two variations of our NLP-Options. The first only uses options with navigation, moving from one location to another, regardless of reward. The second uses NLP-Options, but the extra steps from a human generated walk-through are added to the policy so that each generated option has a reward.

Results and Discussion

Beginning with the Gemworld environment, after generating sets of options for both ARL and our NLP version, the ARL algorithm created a total of 13 options, while the NLP algorithm only created 11. One of the primary differences was that ARL algorithms tend to dove-tail into each other and do not typically overlap. This means that as one option terminates at a specific state, ARL generally has another option at that state ready to continue. This also means that ARL options tend to have shorter numbers of steps before termination. NLP options on the other hand tend to be longer, and some of the options do overlap, so that an option might have a start state of having gem #2, but a terminal state where it opens the door to the exit. This is a very long option with a policy based on prior traces, that will get the agent closer to the overall goal. Because the options tend to be longer and there are fewer option choices to reach the final goal, this lets NLP outperform ARL. In the results graph in figure 5 it can be seen that NLP options tend to converge to optimal results, in this case from a starting position of (4,6), and reaching that maximum point value of 149 in about 550 episodes, where ARL takes about 650 episodes and Q-learning takes nearly all 1000. On this particular problem, fewer options and longer options with successful historical policies decreases the number of episodes an agent requires to get to very good or even optimal outcomes.

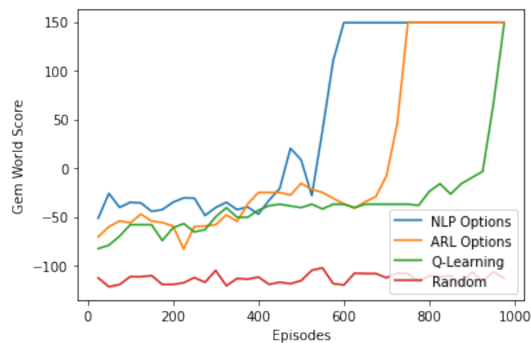

Figure 5: Gemworld Results for 1000 episodes

The Zork environment provided two distinct challenges for the agent beyond the Gemworld environment. First, the size of the available action space has grown, giving the agent many more choices at each state to choose from. Secondly, there are many parallel tasks the agent can try to accomplish: gather the egg, break into the house to go to the cellar, and then steal a painting, find some coins in a maze, or rob a bank. If the agent chooses to go to the cellar first, this gives a much bigger immediate reward, but unless the agent picks up the lamp and lights it, once it goes down the cellar, it will be locked in and die before it can go back for the egg.

All three learning agents eventually learn to grab the egg and navigate into the house, achieving the 5 points for the egg and 10 points for entering the house. Another 25 points is gained by going into the cellar, but 5 points is usually lost if done without the lamp. On individual runs, both the DQN and the Navigation-only Options would often make it into the house but struggled in their 150 moves to make it to the cellar, which requires multiple steps to achieve the reward of moving a rug, opening a trapdoor, and then going down into the cellar. Even though the agents could potentially earn up to 106 points, none really came close to that either on individual runs or improved their averages and often just made it to the cellar. The pre-requisite of getting the lamp and lighting it first was too challenging as it required two extra steps beyond just going into the cellar. Overall, the NLP-Options that had reward showed that they could get the egg, enter the house, and navigate to the cellar on average faster. Unlike Gemworld, there is no cost for taking a step in Zork and so most agents do not lock themselves in the cellar with no light and end the episode. In the runs that are shown NLP-Options with navigation only out-performs the DQN, and the NLP-Options that receive rewards at their conclusion out-performs all. A big challenge for all agents on this problem is the size of the action space. There are so many choices that sometimes it is more luck on any given run if the agent performs well by picking or landing randomly on a good location and choosing the one correct action to receive a score. Given that NLP-Options with rewards essentially guarantee a reward if one of their options are chosen, after the typical rocky start, it eventually settles into easily mak-

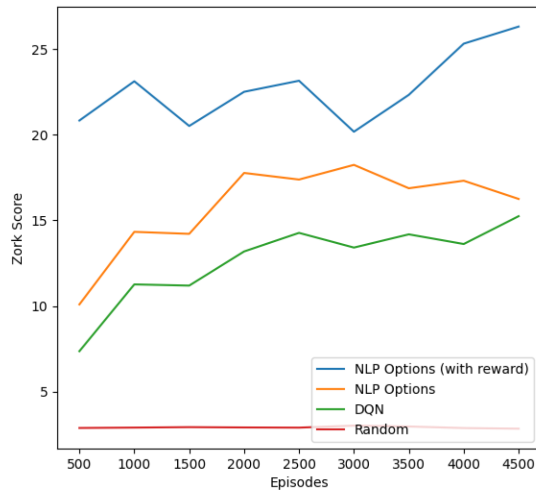

Figure 6: Zork averaged results for 5000 episodes across 10 runs

ing it to the cellar regularly. Notice that with a good set of natural language options that include rewards, this agent can overcome the early randomness and on occasion succeed in not getting stuck in the cellar, whereas the navigation options agent still gets stuck in the cellar and tries to avoid it.

Conclusion

In this paper we proposed a new approach to discovering options using natural language descriptions from prior knowledge of a problem. The results on smaller problems show that with appropriate natural language descriptions of the steps of a problem, and translating those natural language descriptions into abstract nodes structured into options, that options with longer policies borrowed from prior successful traces allows for quicker learning for an agent than traditional Q-learning and statistical approaches to developing traces like ARL. On larger problems we can incorporate options based on natural language instructions to increase the abilty of the agent to successfully navigate the game. Also it appears that giving options direct rewards on conclusion helps DQNs reinforce the choice for a given option. The ability for natural language to abstract large sets of states into groups having some similarity suggests that future work should be performed against more complicated environments with larger action spaces and bigger state spaces. One area of note is that many of the actions for any location in a game are sometimes irrelevant. Adding an ability to logically eliminate or cull the action/option space would greatly improve the performance as it would allow the agent a higher chance of selecting positive actions or options.

References

Borgelt, C. 2012. Frequent item set mining. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, 2(6): 437–456.

Dai, P.; Strehl, A. L.; and Goldsmith, J. 2008. Expediting RL by using graphical structures. In *Proc. AAMAS*, 1325–1328.

Dietrich, R.; Cochran, L.; Peterson, S.; Randle, B.; Gilmore, J.; Taylor, I. L.; and Thomas, S. 2024. DUNGEON (Zork I). https://github.com/devshane/zork.

Goyal, P.; Niekum, S.; and Mooney, R. 2021. PixL2R: Guiding Reinforcement Learning Using Natural Language by Mapping Pixels to Rewards. In *Conference on Robot Learning*, 485–497. PMLR.

Hausknecht, M.; Ammanabrolu, P.; Marc-Alexandre, C.; and Xingdi, Y. 2019. Interactive Fiction Games: A Colossal Adventure. *CoRR*, abs/1909.05398.

Infocom. 1980. Zork: The Great Underground Empire – Part I. [Floppy].

McGovern, A.; and Barto, A. G. 2001. Automatic discovery of subgoals in reinforcement learning using diverse density. *University of Massachusetts Amherst Computer Science Department Faculty Publication Series*.

Mobley, D.; Goldsmith, J.; and Harrison, B. 2020. Discovering Hierarchies for Reinforcement Learning Using Data Mining. In *The Thirty-Third International Flairs Conference*.

Parr, R.; and Russell, S. J. 1998. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems*, 1043–1049.

Şimşek, Ö.; and Barto, A. G. 2004a. Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *Proceedings of the 21st International Conference on Machine Learning*, 95. ACM.

Şimşek, Ö.; and Barto, A. G. 2004b. Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *Proc. ICML*.

Şimşek, Ö.; Wolfe, A. P.; and Barto, A. G. 2005. Identifying useful subgoals in reinforcement learning by local graph partitioning. In *Proc. ICML*.

Stolle, M.; and Precup, D. 2002. Learning options in reinforcement learning. In *International Symposium on Abstraction, Reformulation, and Approximation*, 212–223. Springer.

Sutskever, I.; Vinyals, O.; and Le, Q. V. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27.

Sutton, R. S.; Precup, D.; and Singh, S. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2): 181–211.

Tasrin, T.; Al Nahian, M. S.; Perera, H.; and Harrison, B. 2021. Influencing Reinforcement Learning through Natural Language Guidance. In *The International FLAIRS Conference Proceedings*, volume 34.

Yang, T.-Y.; Hu, M. Y.; Chow, Y.; Ramadge, P. J.; and Narasimhan, K. 2021. Safe reinforcement learning with natural language constraints. *Advances in Neural Information Processing Systems*, 34: 13794–13808.