ReL GoalD

(Reinforcement Learning for Goal Dependencies)

Derek Koleber

Ruben Acuña, Director

Ajay Bansal, Second Committee Member

**Index**

1. Introduction
   1. Abstract
   2. Purpose
   3. Motivation
   4. Scope of project
   5. Key Features
2. Related Work
   1. Neural Networks
   2. Goal Solvers
3. System Architecture
   1. World Model
   2. Dependency Model
   3. Problem Definition
   4. Agent Model
4. Implementation
   1. PlayerStates
   2. Design
   3. Description of ActionTargetSelectors/Agents
5. Results
   1. Context
   2. Data
   3. Analysis
6. Future Work
   1. Migration to a Framework
   2. Improvement of World Model
   3. Improvements to Selection Methods Benchmarking
7. Conclusion
8. References

# Introduction

**Abstract**

In this project, the use of deep neural networks for the process of selecting actions to execute within an environment to achieve a goal is explored. Scenarios like this are common in crafting based games such as Terraria or Minecraft. Goals in these environments have recursive sub-goal dependencies which form a dependency tree. The agent operating within these environments have access to low amounts of data about the environment before interacting with it, so it is crucial that this agent is able to effectively utilize its tree of dependencies and its environmental surroundings to make judgements about which sub-goals are most efficient to pursue at any point in time. A successful agent minimizes cost when completing a given goal. A deep neural network in combination with Q-learning techniques was employed to act as the agent in this environment. This agent consistently performed better than agents using alternate models (models that used dependency tree heuristics or human-like approaches to make sub-goal oriented choices), with an average performance advantage of 22.56% over the best alternate agent. This shows that machine learning techniques can be effectively employed to make goal-oriented choices within an environment with recursive sub-goal dependencies and low amounts of pre-known information.

**Purpose**

The purpose of this project is to explore the application of a deep neural network for the process of selecting actions to execute within an environment to achieve a goal, where goals have recursive dependencies on the completion of subgoals. More specifically, the purpose is to explore if the application of deep neural networks as a decision making component for the agent in such an environment can result in more efficient performance (performance being measured as the minimization of clock ticks per simulation) than agents using heuristics or human-style decision making processes.

**Motivation**

The primary objective of this project is to create an efficient, machine learning oriented approach for accomplishing goals with recursive sub-goal dependencies within an environment where there are low amounts of pre-known environmental data. Specifically, the objective is to create a machine learning oriented approach that doesn’t rely on perpetually storing data about the environment during execution (like a recurrent neural network with or without long short term memory would), but to rather create an approach which applies a learned heuristic to make goal-oriented decisions. The reason for preferring a learned heuristic over an approach with some form of retention of data over time is to stay consistently applicable and effective in any environment within the problem space regardless of pre-known or collected environment data.

**Scope**

This project will be confined to a custom game environment designed to emphasize the importance of efficiently selecting actions to pursue in an environment with goal dependencies This custom game draws game rules from commercial games exhibiting goal dependencies but doesn’t have any overly specific game rules, to make it easier to draw generalizations from the architecture/implementation/results from this project to other contexts outside of this custom game. There are existing frameworks for simulating game environments for machine learning purposes such as OpenAI Gym, but the overhead involved with creating a new game engine is negligible.

**Key Features**

Among the features that give this project its character, there are some notable structures to point out before delving into specifics (see Fig. 1).

* The dependency tree: Since goals have recursive sub-goal dependencies, it is necessary to map out the relation between a goal, its sub-goals, those sub-goals’ sub-goals, and so on. This is significantly different than other systems (reviewed in Related Work) with goals since some sub-goals aren’t immediately accessible and require completion of their children sub-goals first. This tree structure adds to the complexity of an efficient policy to complete a goal since it represents not only dependencies, but also choices that have to be made (in the case of orthogonal sub-goals which are discussed in System Architecture).
* The target environment: Many systems involve goals and sub-goals that exist within either a fully closed environment, or an environment where there is large amounts of pre-known environmental data. In the case of this project, the agent tasked with accomplishing a goal has access to either low or inexistent amounts of pre-known environmental data.
* The agent: The environment and the dependency tree are the two locations where the agent draws data in order to make sub-goal decisions. This means the agent needs tailored inputs and outputs to be useful in this problem context.

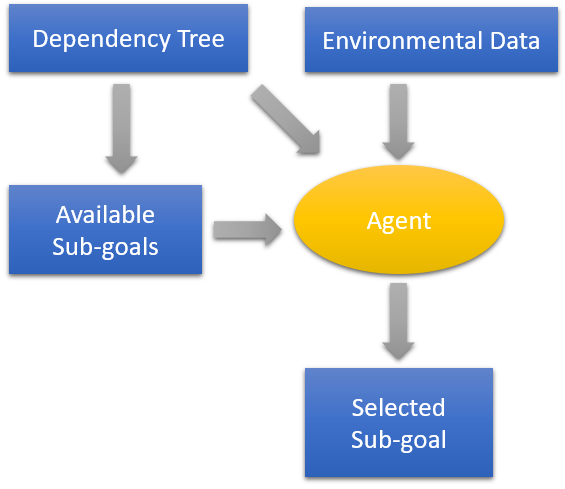


Fig. 1: The relationship between the agent, the dependency tree, and the environment.

# Related Work

**Neural Networks**

Attempting to find efficient policies for accomplishing goals with machine learning (ML) techniques is nothing new. In reinforcement learning (RL), one approach used by agents to complete goals is Q-learning, which uses an action-value function to select actions based on the maximization of their anticipated reward. This technique is merged with the utility of deep neural networks (DNNs) to create a deep q-network (DQN). DeepMind [1] has applied DQN’s to Atari 2600 games to produce results far beyond normal human performance , showing the potential for this combination of ML techniques.

**Goal Solvers**

Dealing with both recursive and non-recursive subgoals (as is implied with goal dependencies) is nothing new. Work in the area of goal sub-goal dependencies includes works such as *Handling Prioritized Goals and Subgoals in a Logical Account of Goal Change* [2] which handles goal prioritization in a system with goals and sub-goals, or *Planning with Goal Utility Dependencies* [3] which investigates how to plan goals based on the utility they provide. Using these two pieces as examples, it appears that sub-goal planning is a problem that can be solved given enough information about the sub-goals and their environment.

In the context of this project, there is not necessarily enough information to make an ordering of sub-goals that is optimal for all instances of a goal. It’s useful to have an example environment that by nature has low amounts of known environmental data in order to investigate sub-goal selection policies. None of the Atari 2600 games are of particular interest to this project since they don’t exhibit goal dependencies complex enough to warrant a significantly different approach to a normal DQN. This requires the selection of a game that does exhibit non-trivial goal dependencies: Minecraft.

Minecraft is a sandbox videogame where the player navigates a 3D environment made of open spaces or blocks. Some of these blocks are harvestable and, once harvested, can be used to craft other objects such as tools or other blocks. There have been a few projects that have been involved with accomplishing goals in Minecraft. *Project Malmo* [4] provides a platform for AI research based in Minecraft, with functionality such as specifying simulation variables like rewards and penalties to be applied to agents during training.

*Selecting Subgoals using Deep Learning in Minecraft* [5]investigates the use of a convolutional neural network (CNN) to select the appropriate (non-recursive) subgoal in order to overcome an environmental obstacle given visual input frames. The agent in this project was able to achieve 87.1% accuracy when selecting the appropriate sub-goal approach to combat randomly generated obstacles.

Neither of these projects ([4],[5]) involving Minecraft deal with recursive sub-goals (sub-goals that depend on other sub-goals). In fact, the use of ML to select sub-goals to accomplish top-level goals in an environment with low amounts of known information is fairly untrodden.

# System Architecture

**World Model**

When applying an algorithm to any system, a problem model must be present. In the context of this project, the agent must interact with a game world, thus necessitating a world mode. The more detailed the model, the more information an algorithm can potentially extrapolate from the model, giving it the chance to produce results highly specific to that problem. However, this comes with drawbacks like difficulty finding an effective algorithm due to a larger number of parameters to account for, or difficulty modeling the world to a high level of detail. With lightweight world models, an algorithm has less information about the actual problem space, meaning the algorithm must be sophisticated in order to be successful. An algorithm interacting with a lightweight model makes less assumptions, so this means it is also more portable. Since generalizability and portability are goals of the algorithm produced by this project, a lightweight world model is appropriate. This also makes modeling the world easier, and reduces the bias that might be introduced by producing a more in-depth world model.

This project models a two-dimensional world which is a simplification of the three-dimensional game Minecraft. This game will be referred to as BlockLand. Similar to Minecraft, the world in BlockLand is a grid of discrete positions; each position being occupied by nothing, a breakable block, or an unbreakable block, around which the player’s avatar navigates to accomplish goals. The world can be modeled with a two-dimensional grid representing the environment and an avatar which is located on that grid and can observe all blocks within a certain block radius.

There is a finite set of actions that can be taken within this environment, denoted by bold lettering in the following description of interactions a player can make in this environment.

*Resources are game objects that are contained in the player’s inventory, inclusive to tools. A resource can be acquired by* ***harvest****ing a block or by* ***craft****ing it from other resources. Some blocks require tools to harvest****.*** *A tool is a special resource which has both a class and level which determines which blocks is can be used to harvest. Some resources and all tools must be* ***craft****ed by interacting with a ‘crafting bench’ block. A player must* ***move*** *adjacent to a block to harvest it, and a player must move adjacent to a crafting bench to craft most items (but can craft the rest anywhere).*

*Examples:*

*Wood may be harvested with any tool.*

*Planks are crafted from wood.*

*Sticks are crafted from planks (which have to be crafted from wood).*

*A wooden pickaxe is crafted from planks and sticks with a crafting bench.*

*Stone can only be harvested by any pickaxe tool with level 1 or greater.*

**Dependency Model**

Based on this description of actions alone, it already becomes clear that a player has a tree of dependent actions that must be completed if they want to build a simple tool. The tree of dependencies grows when higher level tools (that require lower level tools to make) are brought into the game. This necessitates a model of the goal dependency tree in addition to the world model. Rather than having actions be dependent on one another, the player has a current **PlayerState** and a goal **PlayerState** that they seek to reach.

*A* ***PlayerState*** *describes the state of the player within the environment. A PlayerState includes information about the contents of the player’s inventory, and their position in the environment relative to certain blocks.*

*Example of a PlayerState: The player has 3 sticks and 2 planks in the inventory, and is adjacent to a crafting bench.*

**Problem Definition**

A player within the environment can have a target PlayerState that they’re trying to reach, so that player has to take actions in order to move from their current PlayerState to that target. A player can easily pick actions sequentially without regard to any heuristic or algorithmic action selection approach, but this wouldn’t be efficient since it is likely that the environment or the current state of the dependency tree can provide useful information about which actions on the dependency tree would be better to complete.

From this, the following problem arises:

*For goals with large dependency trees, how are sub-goals selected to efficiently achieve top-level goals?*

Optimum efficiency could be achieved within a static environment algorithmically, but this approach is neither scalable to large environments, nor large dependency trees. This means a generalized heuristic approach is a good choice to combat this problem.

**Agent Model**

In order to make choices about what sub-goals to pursue an agent can assess:

* the position on the dependency tree of the available sub-goals (which are leafs on the tree),
* the estimated cost of the sub-goals (which are defined statically prior to entering an environment), and
* the environment surrounding the agent.

It’s feasible to write a goal selection method specific to this environment to pick the optimal sub-goal every time, given complete knowledge of game rules (including knowledge of the available actions within the game as well as the impact those actions have on the player and the environment) and using optimal game strategy. However, this is not a general solution and is necessarily coupled to a specific game. A better solution that would be significantly more general is training an AI to handle the goal selection.

In this case, the AI takes the form of a DNN. The characteristics of DNN slightly limit the input that are available to the agent in the game, since a traditional DNN doesn’t have a dynamically sized input which would potentially correspond with the entire dependency tree or the available leafs. This leaves the agent with environmental input, which is fine since makes sense for the agent to only be able to observe a consistently sized area around itself (see Fig. 2). With this, input is accounted for, but output is not since it suffers from the same dynamic sizing problem as the input. Since all available actions are supposed to be known prior to entering the environment, one potential output for the DNN is a set of outputs for each action. In order to make these outputs useful to sub-goal decision making, aspects of Q-learning are implemented. Each of these network outputs can be used as discount factors for the estimated costs of all actions, which can be applied to the statically defined estimated cost for each action within the environment. Then, based on the available actions on the decision tree, an action is picked that has the highest anticipated reward (or the lowest anticipated cost).

Fig 2: An example of what the agent sees in the environment. The agent, represented by the single blue pixel in the lighter area, can only see a small set of blocks in its immediate vicinity. The raw pixel values of the grid that the agent can see is used as input in the DNN.

# Implementation

**PlayerStates**

As mentioned in the system architecture section, a goal within the environment is represented by a PlayerState (PS). In order to bridge the PS discrepancy between a goal PS and the current PS through actions, the goal PS’s can’t be directly connected to the current PS since more than one action may be required to bridge the states. The three main issues that must be considered in order to bridge the PS discrepancy between the goal PS and current PS include:

1. An action may produce part of the desired PS but can’t produce the rest of that PS
2. There may be more than one action that can produce the desired PS
3. There may be an action that must be executed more than once in order to produce the desired PS

These issues are solved by using a dependency tree with three layers per level. Before describing the layers that are necessary, some properties of a PS that facilitate this solution must be stated.

Since a PS may have multiple unique components (such as more than one resource type within the inventory and/or more than one item in the inventory in conjunction with a specification of the player’s relative position in the environment - referred to as adjacency), a PS can always be broken down into “attribute” PS’s which hold only one unique, independent PS component. Example:

*PS prior to breakdown into attributes:*

*{inventory: { stick: 5, wood: 2}, adjacent\_to: crafting\_bench}*

*Attribute PS’s after breakdown:*

*{inventory: {stick: 5}}*

*{inventory: {wood: 2}}*

*{adjacent\_to: crafting\_bench}*

The first layer of a level contains PlayerStateTargets (PSTs) which represent goals:

*A PlayerStateTarget holds the goal PS. Additionally, for each attribute PS of the goal PS, a PST holds a list of PlayerStateSolutions (described below).*

The second layer of a level contains PlayerStateSolution’s (PSS’s) which represent an approach to achieve a goal:

*A PlayerStateSolution represents an guaranteed solution to resolve the PS requirement of its corresponding attribute PS in its parent PST. It holds at least one child ActionTarget (described below).*

The third layer of a level contains ActionTargets (AT’s) which represent executable actions that belong to a specific solution to a goal:

*An ActionTarget holds an executable action that has a PS requirement that must be fulfilled before it can be executed, and a PS result that will be produced once it is executed. If there is no PS requirement for an AT, it has no children, but if there is, it has a child PST. If the AT has a fulfilled requirement or no requirement, it is considered a leaf.*

Each layer in this architecture references its children to create a tree-style data structure (see Fig. 3).

This three layer architecture resolves all three issues with mapping current PS’s to goal PS’s.

1. An action may produce part of the desired PS but can’t produce the rest of that PS
   * This is resolved by the goal PS being broken into attribute PS’s in the PST. This way, each attribute can be approached separately, ensuring the entire goal PS is fulfilled.
2. There may be more than one action that can produce the desired PS
   * This is resolved by the PST holding a list of PSS’s for each attribute PS. Within those lists, multiple approaches for fulfilling that attribute PS are present.
3. There may be an action that must be executed more than once in order to produce the desired PS
   * Since an AT may not completely fulfill the attribute PS in a PST, PSS’s are used to hold a collection of AT’s which all together fulfill the attribute PS.

Now that the setup of the dependency tree is clear, it is appropriate to start looking at the implementation of the entire system.

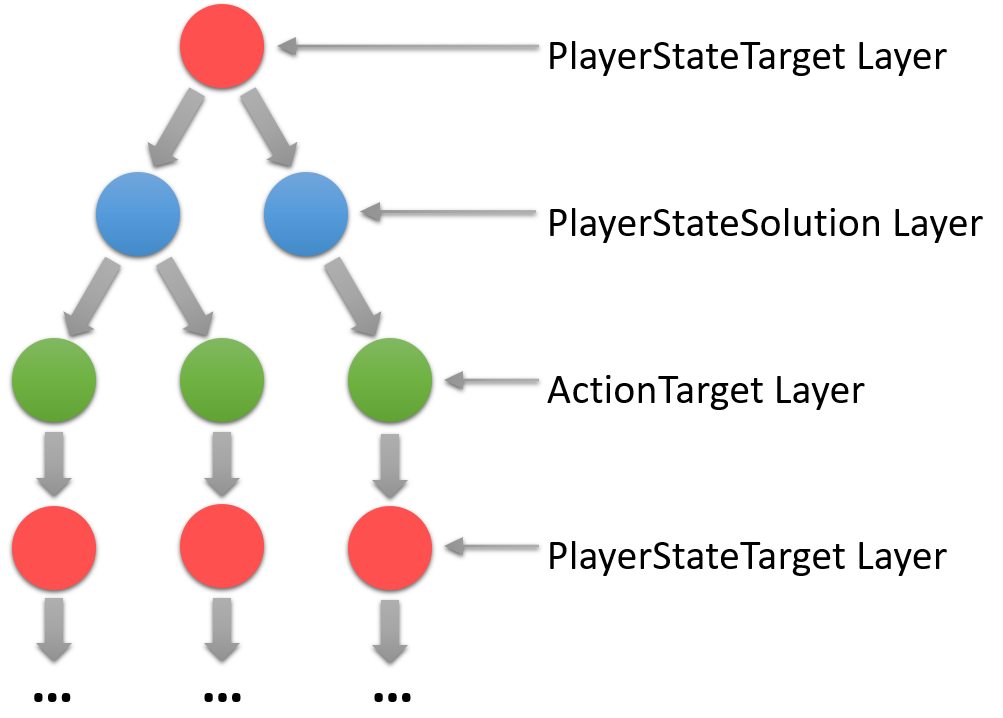


Fig. 3: An archetypal dependency tree.

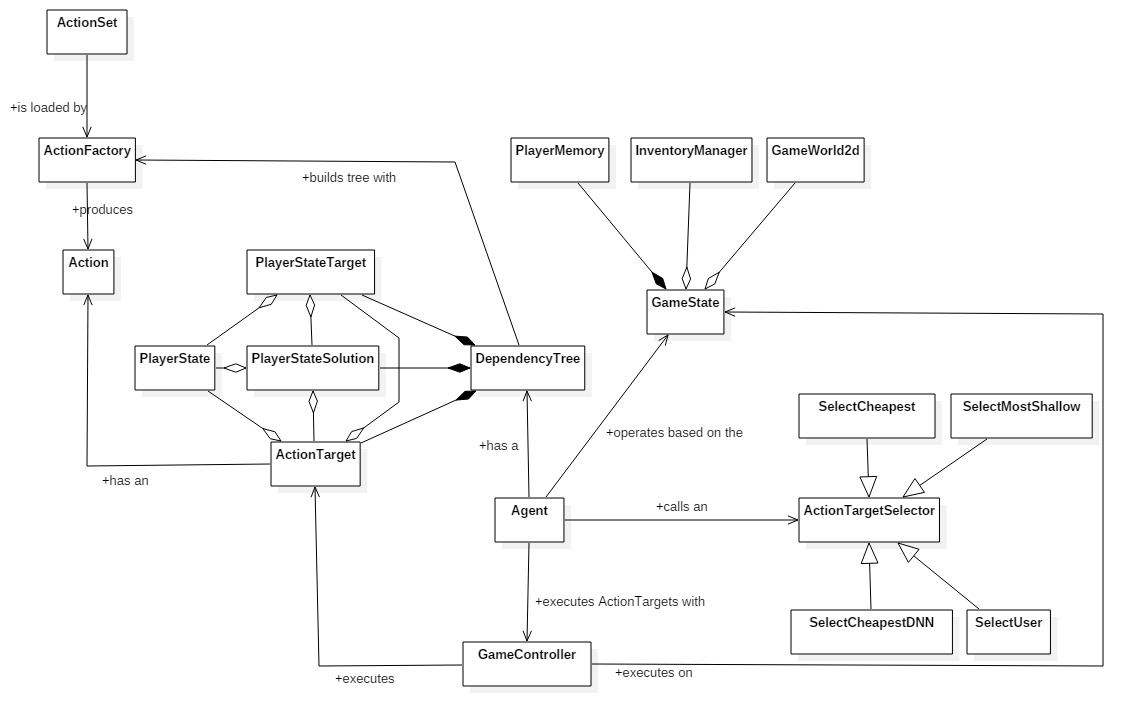
**Design**

Fig 4: Class diagram for the system

**Additional Class Descriptions**

* **Action** is an executable action with a PS requirement and PS result
* **ActionSet** represents a set of actions that can be performed in the environment
* **ActionFactory** is used to produce Actions that are necessary to resolve goal dependencies while constructing the dependency tree
* **ActionTargetSelector** embodies the decision making element of the system and therefore the focus of this project - how to make efficient ActionTarget selections. The specific functionality of each ActionTargetSelector subclass is described in the next sub-section
* **Agent** executes actions within an environment by using an ActionTargetSelector to pick an AT to execute from the set of AT leafs on the dependency tree
* **PlayerMemory** is used to hold execution overhead such as pathing information and collected data to train the model.

**Description of ActionTargetSelectors**

In order to assess the effectiveness of any AT selection method, alternative AT selection methods are needed to validate performance. As mentioned in System Architecture, the information available to an agent consists of environmental data and dependency tree data. Based on the information about a candidate AT’s position on the tree as well as its Action’s static data, a few potential heuristics seem obvious. Each action has a predetermined estimated cost for each environment, meaning an agent could use this without considering any other factors. Alternatively, an agent could use candidate ATs’ depth in the dependency tree to make a better selection. In order for a selection method to be viable, it needs to make progress toward the end goal at every game step, and must have a termination point where it achieves the goal.

Some of these options are not viable due to the fact that they inevitably result in a model that can never complete some goals. An example of this would be an agent that selects the deepest available AT in terms of its position on the dependency tree: a *SelectDeepest* model. One property that is not structurally explicit in the dependency tree is the fact that some series of actions must be executed in sequence (such as **moving adjacent to** then **harvesting** a resource). If the sequence is broken, some previously completed actions have to be rolled back (refer to Fig.5 while reading the following example).

With a *SelectDeepest* model, the deepest AT *at\_1* will be continuously selected for execution until it is completed (in this case, a **move adjacent to** action), exposing an action *at\_2* that must be completed in sequence (in this case, a **harvest** action) (Fig. 5, stage 1). *at\_2* might have the same depth as another unrelated action *at\_3* which the agent ends up selecting for execution. *at\_3* will be selected once for execution (Fig. 5, stage 2) but *at\_1* (the **move adjacent to** action) will be rolled back, making it the deepest AT once again. This causes *at\_3* to be rolled back (Fig. 5, stage 3), and the entirety of *at\_1, at\_2,* and *at\_3* have to be completed in their entirety.

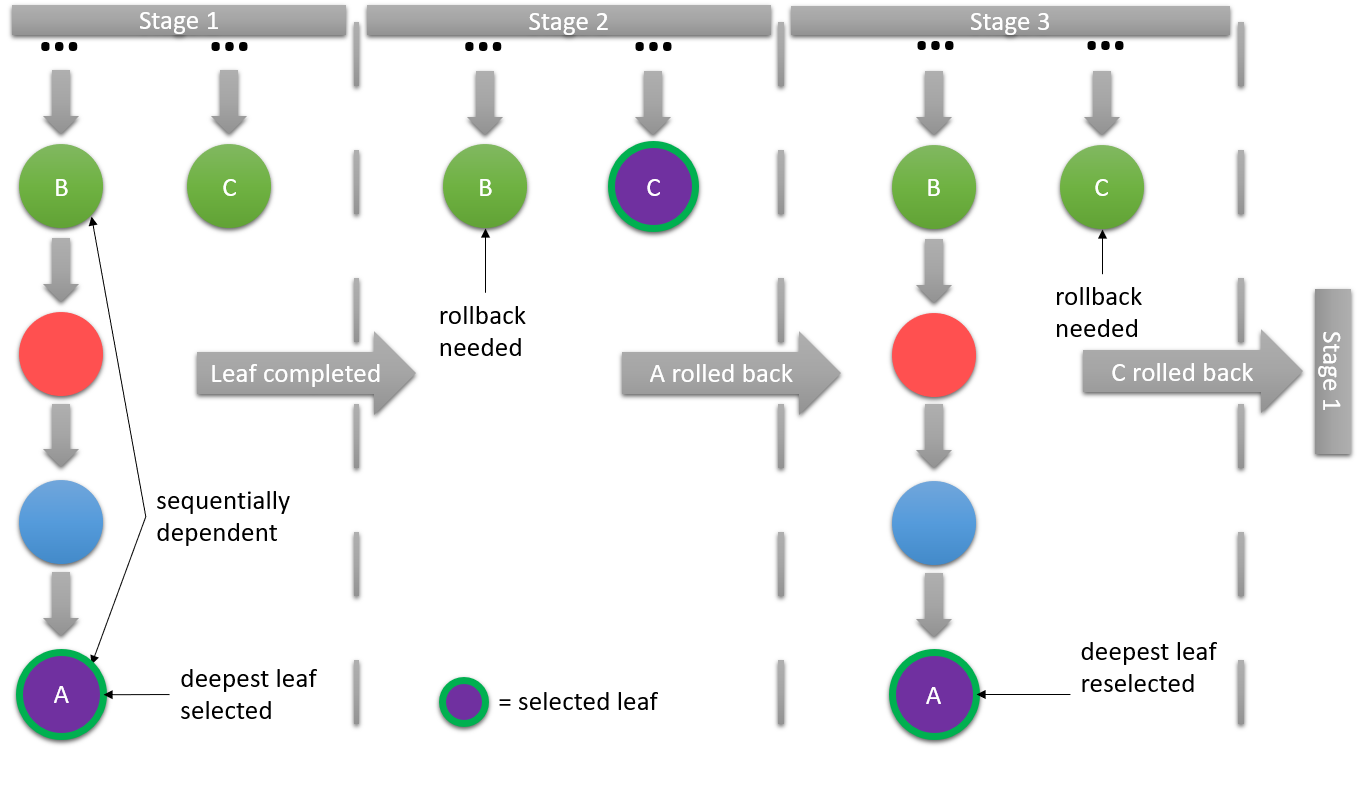


Fig. 5: An example rollback loop for *SelectDeepest*

This example demonstrates why some approaches inherently do not work as AT selection models due to the nature of the dependency tree. Other methods may not work due to characteristics of environmental data, such as *SelectMostExpensive* which selects the candidate action with the highest predetermined cost. Many of the sequential action chains have costs that decrease as the node’s depth decreases. For many situations, this means that cost acts as a proxy for node depth (at least with this set of predetermined estimate costs) so similar cycles occur as those that arise with *SelectDeepest*. However, there are some models that are guaranteed to complete, depending on game rules and predetermined estimate costs.

* *SelectMostShallow* is guaranteed to not run into a recursive roll-back loop so it works as a model, and selects the candidate AT with the lowest depth in the dependency tree.
* *SelectUser* explicitly looks for sequential dependencies and selects candidate AT’s with priority depending on their position within chains of sequential action sets. This is meant to represent a similar selection method as to what a player would likely use.
* *SelectCheapest* selects the candidate with the cheapest predetermined cost estimate. This has the potential to fail for the same reason as *SelectExpensive*, but with the existing set of predetermined estimate costs with cost acting as a proxy for node depth, *SelectCheapest* is guaranteed to complete for the same reason as *SelectMostShallow*. This issue is further mitigated by using a “switching cost” which disincentives roll-back loops.
* *SelectCheapestDNN* is identical to *SelectCheapest* except for the fact that ATs’ costs are discounted by scalars produced by a DNN which takes environmental data as an input. This approach resembles Q-Learning, since it minimizes anticipated cost (similar to maximizing anticipated reward since simulations with lower costs are preferred). This approach is preferred to *SelectCheapest* since costs are dynamically scaled based on environmental factors to make the decision making process reactive to the environment. This allows *SelectCheapestDNN* to perform better than *SelectCheapest* when it correctly identifies environmental factors that correspond with a minimized anticipated cost.

# Results

**Context**

The agent that incurs the lowest total cost to accomplish a goal is considered to have the best performance. If the goal of this project is to be achieved, a ML approach to action selection (manifested in *SelectCheapestDNN*) will result in the higher performance than alternate selection methods. In order to determine the relative performance of the *SelectCheapestDNN* selection method in comparison to the three alternate selection methods, models were searched for until a DNN instance showed a significant performance advantage (in this case, a ~30% advantage during training validation) over the alternate selection methods. Each selection method was simulated in six randomly generated environments demonstrating different environmental features. In order to statistically significant results, selection methods were sampled until selection method performance began to converge (which was approximately 20 samples). The results of these benchmark samples are shown below.

**Data**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Performance Advantage of *SelectCheapestDNN* Over Alternate Selection Methods | | |
| Simulation | *SelectCheapest* | *SelectShallow* | *SelectUser* |
| rv\_4 (20 samples) | 9.80% | 44.50% | 59.79% |
| rv\_3 (20 samples) | 8.73% | 24.93% | 18.65% |
| rv\_2 (20 samples) | 47.27% | 58.85% | 61.70% |
| rv\_1 (20 samples) | 3.69% | 42.09% | 38.70% |
| medium\_1 (20 samples)\* | 39.75% | 39.63% | 36.82% |
| medium\_2 (20 samples)\* | 5.72% | 20.04% | 21.17% |
| dense\_2 (20 samples) | 42.93% | 51.16% | 56.48% |
| average | 22.56% | 40.17% | 41.90% |

|  |  |
| --- | --- |
| compared to worst alternate agent | |
| compared to middle alternate agent | |
| compared to best alternate agent | |

**\*** For simulation medium\_1, *SelectCheapest* exhibited unusual relative performance. For this reason, medium\_2 (the same simulation as medium\_1 with very slight environmental variation) was executed which produced results that were in line with the rest, showing evidence that medium\_1 is was an outlier.

**Analysis**

In all simulation contexts, CheapestDNN out-performed all other selection methods. Most\_Shallow and User methods have significantly worse performance than CheapestDNN, showing that heuristics based exclusively dependency tree data is insufficient for high performance. More importantly, CheapestDNN has a consistently higher performance than Cheapest, meaning that using incorporating machine learning to analyze environmental data results in an improvement to a naive agent that doesn’t discount costs according to learned patterns.

The DNN used in CheapestDNN was had only three hidden dense layers and did not take inputs of rich environmental data. With even a lightweight model such as this one, there are still clear results indicating using a DNN improves performance of the agent.

# Future Work

**Migration to a Framework**

In its current state, this project contains a selection method that uses ML techniques to efficiently accomplish goals within a specific game environment. However, it does not have the functionality to be effectively applied to a context outside of BlockLand. This is the main area of recommended improvement for this project. In order to move this project to a place that allows it to act as a framework for environments that stand to benefit from the functionality demonstrated by this project, much of the functionality needs to be decoupled from example game environment to make it easily extensible.

* The definition of a PlayerState needs to be abstracted in order for context specific implementations of PS to be able to interface with the system
  + The inventory and adjacency components of a PS need to be refactored to represent attributes of the agent in the world that also have a priority component (inventory has priority over adjacency - inventory PS attributes have to be completed before adjacency is approached)
* The way the agent interacts with the world model needs to be abstracted.
  + As it is, the agent is able to directly manipulate the world model and therefore the world, but this may not be the case in a different context. The agent should be able to take actions in the world model but observe the outcome in the world model rather than assuming an outcome
  + The input that the agent receives every clock tick is partly specific to every environment. While every agent has access to the dependency tree, the environments that an agent is interacting in might be represented by pixels, audio samples, or other various types of data which need to be abstracted

**Improvement of the World Model**

One area needing improvement that is specific to the example environment but may be generalizable is the way the agent can interact with the world model. This includes areas of improvement such as:

* Allowing the agent to better manage existing resources
  + Allowing the agent to utilize excess or orphaned resources (resources that were produced but weren’t required by the AT’s parents or resources that were produced by a previously completed but rolled back AT)
  + Allowing the agent to repurpose resources / transfer ownership of resources between different PSS’s/PST’s

**Improvements to Selection Methods Benchmarking**

The focus of this project, *SelectDeepestDNN* has some reasonable data to compare with to determine the performance improvement provided by implementing ML techniques, but there

are some things that can be done to provide additional valuable comparisons.

* Add a user interface to allow users to play BlockLand. This would allow *SelectCheapestDNN* to be compared against data which represents human performance better than an approximate codification of human performance like *SelectUser*.
* Add non-ML selection methods that utilize environmental data. As it stands, none of the existing selection methods besides *SelectCheapestDNN* utilize environmental data. A selection method that utilizes environmental data but not ML techniques would help illustrate the advantage that ML provides over other methodologies.

# Conclusion

This project demonstrates how ML techniques can be used to complete goals with recursive sub-goals in an environment with low amounts of pre-known information.

In the project, an architecture of a dependency tree and agent was designed and implemented to operate in an example game named BlockLand. Blockland was designed to be mechanically representative of goal based games as to make it easy for findings from this project to be extended to similar contexts. The agent of focus in this system used a DNN with Q-learning aspects to select AT’s in order to accomplish the top-level goal. Alternate agents used other heuristics to make these AT selections. After comparing the agent of focus with the alternate agent, benchmark comparisons determined that the *SelectCheapestDNN* agent had consistently better performance than the alternate agents (that used heuristics based on the dependency tree or human-like selection methods). Depending on the environment, the agent of focus exhibited a performance advantage between 3.69% and 47.27% (with a mean of 22.56%) over the best alternate agent. This shows that ML techniques can be effectively employed to make goal-oriented choices within an environment with recursive sub-goal dependencies and low amounts of pre-known information.

Based on the areas of application of similar algorithms (ones that don’t deal with recursive sub-goals or only deal with closed environments), ReL GoalD has the potential to be applied to task scheduling, navigation, and particularly any goal involved with collecting and processing resources like mining/manufacturing supply chains.

# References

[1] V. Mnih, and K. Kavukcuoglu et al. “Human-level control through deep reinforcement learning”, *Google Deepmind,* 2015. [Online]. Available: <https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf>. [Accessed Mar. 10, 2018].

[2] S. M. Khan, and Y. Lespérance, “Handling Prioritized Goals and Subgoals in a Logical Account of Goal Change”, *EECS at Lassonde,* 2009. [Online]. Available: <http://www.cse.yorku.ca/~lesperan/papers/AAMAS09.pdf>. [Accessed Mar. 20, 2018].

[3] M. B. Do, and J. Benton et al, “Planning with Goal Utility Dependencies”, *CS & Eng at ASU,* 2006. [Online]. Available: <http://rakaposhi.eas.asu.edu/spuds-icaps06-wkshp.pdf>. [Accessed Mar. 20, 2018].

[4] M. Johnson, and K. Hofmann et al. “The Malmo Platform for Artificial Intelligence Experimentation”, *Proc. 25th International Joint Conference on Artificial Intelligence,* 2016. [Online]. Available:<https://github.com/Microsoft/malmo>. [Accessed Mar. 10, 2018].

[5] D. Bonanno, and M. Roberts et al. “Selecting Subgoals using Deep Learning in Minecraft: A Preliminary Report”, *Georgia College of Tech Computing,* 2016. [Online]. Available:

<https://www.cc.gatech.edu/~alanwags/DLAI2016/6.%20(Bonanno+)%20Selecting%20Subgoals%20using%20Deep%20Learning%20in%20Minecraft%20-%20A%20Preliminary%20Report.pdf>. [Accessed Mar 10, 2018].