Université de Liège - Faculté des Sciences Appliquées
Institut Montefiore

Mémoire de fin d'études réalisé en vue de l'obtention du grade de
Master ingénieur civil en informatique

Sequential decision making under uncertainty in randomly generated games :

A Minecraft intelligent agent for resource gathering

Author
Lorent Pierre

Supervisor
Prof. D. Ernst

Année académique
2013 - 2014

Abstract

Video games have significantly evolved since the emergence of first arcade games. They have become more and more complex and now allow for the simulation of advanced environments. In these last years, some titles became popular for their efficiency at modeling reality. Despite it does not focus on amazing and accurate graphics, Minecraft is a good example of such a game. It creates open worlds that are procedurally generated and that can be altered by the players. These worlds also contrain resources that can be gathered and then used to build items. The construction of these items is governed by a technology tree such that advanced items require more rare resources. This often leads the players to go deep underground for finding resources that are required to obtain a valuable inventory.

These characteristics make Minecraft a challenging framework for the design of intelligent agents. This work focuses on the development of an agent that aims at collecting a given type of ressource in the game. For this purpose, the game dynamics are described in a generic decision model that is then used to formalize the goal of the agent as a sequential decision-making problem under uncertainty. The approach used to approximate the optimal behaviour of the agent is iterative and alternates between exploration phases and runs of a batch mode reinforcement learning algorithm. Finally, the quality of the resulting control policies is evaluated, with a particular emphasis on the evolution of policies as the number of iterations increases.

Contents

SECTION 1

Introduction

1.1 Minecraft

Minecraft was developed by Markus Persson, published by Mojang and released in 2011 on PC. The game aims at offering the player the opportunity to start from a wild open world and to turn it into whatever he wants. This encompasses, among other possibilities, the construction of incredible structures (e.g. houses, castles, etc.), the construction of deep mines that lead to rare ressources (e.g. iron, diamond, etc.) and the cultivation of plants.

But the game also requires the player to be able to collect resources before building constructions and these resources must be gathered in the world, sometimes requiring to go deep underground for some rare materials. Moreover, a player cannot directly access these resources if he does not fulfill some pre-requirements. For example, coal is valued as fuel to melt minerals into ingots, making it a very important resource, despite its relative abundance. But before gathering coal, the player needs to build a pickaxe.

To evolve in the game, the players thus have to increase their knowledge about the technology tree in order to progress and to become more efficient, more resilient and to conquer a larger area of the world.

Figure 1: In-game screenshot of a structure built by a player (source [11]).

1.2 World generation

Minecraft worlds are randomly generated and have different types of biomes (e.g. forest, desert, grassland, etc.). This allows for a wide variety of maps and environments to explore. There is no limit to the exploration of these worlds because they are progressively extended when the player discovers new areas. The randomness in the generation of worlds is part of the challenge as resources cannot be found at predefined locations. It means that a player or an artificial agent needs to adapt its strategy as new areas of the environment are being revealed.

1.3 In-game artificial entities

Many computer-controlled entities are already implemented in the game. For example, exploring a world in Minecraft might lead to the discovery of a village, where non-player characters are living. They are designed to look like intelligent entities and they feature social and commercial abilities in order to interact with the player. Some animals might also be tamed. However, these entities only seem intelligent because the tasks they need to fulfill are so narrow that a control algorithm consisting of some predefined rules is usually sufficient. They are controlled by static algorithms which do not learn from experience and thus are not able to evolve.

This technique is often used in video games because it is easy to develop when it comes to low complexity tasks. One the drawbacks of such an approach is that the artificial agents do not learn and they are therefore doomed to fail when facing a situation that had not been accounted by developers. Another problem arises when an artificial agent needs to fulfill more advanced tasks as it becomes very difficult to anticipate all the situations that the agent is likely to encounter. In this work, we explore how machine learning and, in particular, reinforcement learning, can be use to address these issues.

1.4 Structure of the Thesis

This Thesis is structured as follow. Section 2 (Problem statement) clearly states the problem that is addressed and describes how the environment has been modeled. In Section 3 (Algorithms), the approach used to train the artificial agent is presented along with the various algorithms that constitute the elementary building blocks of the approach. The testing conditions and the results are analyzed in Section 4 (Results) and, finally, Section 5 (Conclusion) concludes and identifies some potential future work directions.

The development environment and the code related to this work can be downloaded at `http://www.student.montefiore.ulg.ac.be/~s094483/tfe-lorent-pierre.zip`.

SECTION 2

Problem statement

In this paper, we address the problem faced by an intelligent agent that aims at collecting efficiently a given resource in the three dimensional environment of the game. To do so, the agent has a set of available actions like moving in different directions to explore the world, collecting some resources and building items using the previously collected resources.

This section gradually introduces different concepts and uses them to formalize the decision-making problem. Note that given the complexity of the game mechanics, some choices have been made to exclude rules that are not relevant to accomplish the selected goal.

2.1 Representation of the world

The world consists of a 3D discrete space of equally sized blocks. It can be modeled by a function $map(\cdot)$ mapping discrete coordinates $\mathbf{p} \in \mathbb{Z}^3$ to a type of block. The set $\mathcal{B}$ represents the different block types (e.g. sand, dirt, gravel, rock, air, etc.). The world is also dynamic, meaning that the interactions between the entities (i.e. the players or the artificial agents) and the world can alter it. The world state thus depends on time. We can write:

$$map(\mathbf{p}, t) : \mathbb{Z}^3 \times \mathbb{R} \mapsto \mathcal{B}$$

Some of the blocks represent resources (rock, wood, etc.). These block types are represented by $\mathcal{B}_r \subset \mathcal{B}$ and they will be described further in this chapter.

2.1.1 World view

Depending on its position, an entity can only see a fraction of the whole world. A block is visible by the entity if it is not too far away and if it is reachable by a straight line from the entity's position by crossing only air blocks. The set of blocks an entity can see defines its field of view (FOV).

At time t, given an entity e at position $\mathbf{p}_e$ and a block B at position $\mathbf{p}_b$:

$$B \in FOV_e \iff \begin{cases} d(\mathbf{p}_e, \mathbf{p}_b) = \|\mathbf{p_e} - \mathbf{p_b}\| < D_{view} \\ \\ map(\mathbf{p}, t) = air \ , \quad \forall \mathbf{p} = \alpha \mathbf{p}_e + (1-\alpha)\mathbf{p}_b \ , \quad \alpha \in]0;1[\end{cases} \tag{1}$$

with D_{view} the maximal view distance of e.

From FOV_e we can define an environment descriptor as a vector $\mathbf{e_t} \in \mathcal{E}$ that models the environment surrounding the entity at time t. Values of $\mathbf{e_t}$ represent:

- the number of each resource block included in the FOV;
- the dominant block type in the four horizontal axis-aligned direction;
- an estimation of displacement costs in the main directions;
- the mean view distance in the main directions.

The main directions are the ones in which the agent is able to move and are defined by the following set of vectors:

$$\mathbf{v} = v_x * \mathbf{x} + v_y * \mathbf{y} + v_z * \mathbf{z} \ , \quad \forall (v_x, v_y, v_z) \in \{-1, 0, 1\}^3 \backslash \{0, 0, 0\} : v_x^2 + v_y^2 <= 1 \tag{2}$$

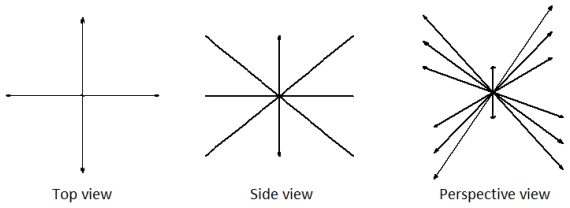


Figure 2: Main directions of Equation 2.

The displacement costs represent an estimation of the time that is required to move in these directions over a predefined distance. The estimations of displacement costs are provided using a heuristic path-finding algorithm. The algorithm used to compute these values is defined in details in section 3.2.1. If no path can be found in a specific direction, the corresponding cost is ∞.

The domain of $\mathbf{e}$ is thus $\mathcal{E} = \underbrace{\mathbb{N}^{|\mathcal{B}_r|}}_{\substack{\text{\# of each} \\ \text{resource block}}} \times \underbrace{\mathcal{B}^4}_{\substack{\text{dominant block} \\ \text{in main directions}}} \times \underbrace{\mathbb{R}^{+14}}_{\substack{\text{estimation of} \\ \text{displacement costs}}} \times \underbrace{[0; D_{view}]^{14}}_{\substack{\text{mean view distance} \\ \text{in main directions}}}$.

2.2 Inventory modelling

In the game, a lot of items are available. These are resources, tools, furnitures, etc. From all these items, only a few were deemed relevant for the agent to accomplish the requested task. The other items are ignored. Tools also have a maximum number of uses that has to be taken into account by the agent to make the good choices as this leads to the destruction of the item.

Given the set of all existing items $\mathcal{O}$, the set of selected items corresponds to a subset $\mathcal{O}' \subset \mathcal{O}$. Items from $\mathcal{O}'$ can be divided into three non-overlapping subsets: resources, tools and others (item construction surplus). We have:

$$\mathcal{O}_r \cup \mathcal{O}_t \cup \mathcal{O}_o = \mathcal{O}'$$
$$\mathcal{O}_r \cap \mathcal{O}_t = \emptyset \quad ; \quad \mathcal{O}_t \cap \mathcal{O}_o = \emptyset \quad ; \quad \mathcal{O}_r \cap \mathcal{O}_o = \emptyset$$

The set of possible inventories is labelled $\mathcal{I}$. An inventory $I \in \mathcal{I}$ containing only objects from $\mathcal{O}'$ is represented by a vector $\mathbf{i} \in \mathcal{I}' = \mathbb{N}^{|\mathcal{O}'|}$, such that each component is defined as:

$$\mathbf{i}_x = \begin{cases} \text{amount of } x \text{ in I} & \text{if } x \text{ is a resource } (x \in \mathcal{O}_r) \\ \text{remaining uses of } x & \text{if } x \text{ is a tool } (x \in \mathcal{O}_t) \\ \text{amount of } x \text{ in I} & \text{if } x \text{ is another item } (x \in \mathcal{O}_o) \end{cases}$$

In case of an inventory containing items which are not included in $\mathcal{O}'$, the corresponding representation ignores these items. Two distinct elements of $\mathcal{I}$ can thus have the same representation in $\mathcal{I}'$.

2.3 State Space

Given $\mathcal{S}$ the set of all possible states of the entity, the state at step t is described by $\mathbf{s}_t \in \mathcal{S}$ and is defined by:

$$\mathbf{s}_t = (T_t, \mathbf{p}_t, \mathbf{\Delta p}_t, \mathbf{i}_t, \mathbf{e}_t)$$

with

- T_t, the elapsed time at step t since the start of the game;
- $\mathbf{p}_t = (x_t, y_t, z_t)$, the position of the agent at step t;
- $\mathbf{\Delta p}_t = (\Delta x_t, \Delta y_t, \Delta z_t) = \mathbf{p}_t - \mathbf{p}_{t-1}$, the displacement vector between positions at steps $t-1$ and t;
- $\mathbf{i}_t$, the vector representing the inventory of the entity at step t as described in section 2.2;
- $\mathbf{e}_t$, the environment descriptor at position $\mathbf{p}_t$ as described in section 2.1.1;

The set $\mathcal{S}$ is thus defined by:

$$\mathcal{S} = \mathbb{R} \times \mathbb{Z}^3 \times \mathbb{Z}^3 \times \mathcal{I}' \times \mathcal{E}$$

2.4 Actions and System Dynamics

The action space $\mathcal{A}$ represents the possible actions for the agent. These actions are:

$$\mathcal{A} = \mathcal{A}_{move} \cup \mathcal{A}_{extr.} \cup \mathcal{A}_{build}$$

such that

- $\mathcal{A}_{move}$ is the subset of actions that corresponds to moving in a given direction;
- $\mathcal{A}_{extr.}$ represents the extraction actions of the resources;
- $\mathcal{A}_{build}$ contains the actions that construct tools using collected resources.

These are described more precisely below. If the entity performs an action a_t from state s_t, the system evolution is governed by a transition function $f(\cdot) : \mathcal{S} \times \mathcal{A} \times \mathcal{W} \mapsto \mathcal{S}$, such that:

$$\mathbf{s}_{t+1} = f(\mathbf{s}_t, a_t, \mathbf{w}_t)\,, \text{with} \;\; \mathbf{s}_t, \mathbf{s}_{t+1} \in \mathcal{S}\,;\; a_t \in \mathcal{A}\,;\; \mathbf{w}_t \in \mathcal{W} \tag{3}$$

where $\mathcal{W}$ is the set of possible realizations of a random process representing the non-deterministic behavior of the system. This non-determinism is due to the uncertainty associated to the blocks from the hidden part of the world (blocks not included in FOV_e) that will be revealed when the entity explores the world or breaks other blocks.

2.4.1 Moving actions

A movement action is defined by one of the main directions from the set defined earlier (see Equation 2). Out of these actions, only those for which a path can be found are possible from a given state. For example, a path that rises vertically is very unlikely to exist if there are no blocks in this area. These directions are considered as impossible. This also enables to handle special cases like deep rifts. The exact path computation is made using an external path finding algorithm that associates a displacement $\mathbf{\Delta p}$ approximating the corresponding direction with $\|\mathbf{\Delta p}\| \approx D_{move}$ (constant). As the environment is not completely known at the target position, the environment descriptor is updated following a stochastic process. After execution of the action, the inventory can also be modified because a displacement may require breaking some blocks, thus increasing the amount of resources and reducing tools durability.

The transition

$$\mathbf{s}_t = (T_t, \mathbf{p}_t, \mathbf{i}_t, \mathbf{e}_t) \quad \overset{a_t \in \mathcal{A}_{move}}{\longrightarrow} \quad \mathbf{s}_{t+1} = f(\mathbf{s}_t, a_t, \mathbf{w}_t) = (T_{t+1}, \mathbf{p}_{t+1}, \mathbf{i}_{t+1}, \mathbf{e}_{t+1})$$

resulting from taking action $a_t \in \mathcal{A}_{move}$, is defined by:

$$\begin{cases} T_{t+1} = T_t + \text{time}(a_t, \mathbf{w}_t) \\ \mathbf{p}_{t+1} = \mathbf{p}_t + \mathbf{\Delta p} = \mathbf{p}_t + D_{move} * \mathbf{v} + \epsilon(\mathbf{w}_t) \\ \mathbf{i}_{t+1} = \mathbf{i}_t + \mathbf{\Delta i}(\mathbf{w}_t) \\ \mathbf{e}_{t+1} = \mathbf{e}_{t+1}(\mathbf{e}_t, \mathbf{w}_t) \end{cases} \tag{4}$$

Where:

- $\text{time}(\cdot) : \mathcal{A} \times \mathcal{W} \mapsto \mathbb{R}^+$ measures the amount of time that has been required to perform the action.
- $\epsilon(\mathbf{w}_t) \in \mathbb{Z}^3$ represents the uncertainty concerning the new position.
- $\mathbf{\Delta i}(\mathbf{w}_t) = (\Delta i_0, \cdots, \Delta i_{|\mathcal{O}'|-1}) \in \mathbb{Z}^{+^{|\mathcal{O}_r|}} \times \mathbb{Z}^{-^{|\mathcal{O}_t|}} \times \{0\}^{|\mathcal{O}_o|}$ models the possible effect on resources and tools, $\mathbf{\Delta i}(\mathbf{w}_t) = \mathbf{0}$ if no block have been destroyed.
- $\mathbf{e}_{t+1}(\mathbf{e}_t, \mathbf{w}_t) \in \mathcal{E}$ is the environment descriptor corresponding to the field of view from the new position $\mathbf{p}_{t+1}$ of the entity.

2.4.2 Resource extraction actions

The extraction actions act mainly on the inventory as they correspond to collecting the resources associated to blocks. There exist as many extraction actions as resources recognized by the agent ($|\mathcal{O}_r|$). Each of these actions is available to the entity as soon as the corresponding block is within its field of view (FOV_e) and the required tool is in the inventory of the entity (if there is one). If more than one block of a given resource is in sight, the action of extracting this resource is performed on the closest block. In the same way, if there is more than one corresponding tool in the inventory, the fastest one will be used.

When the action is executed, the amount of the gathered resource is incremented in the inventory while the durability of the tool that has been used is reduced. The transition

$$\mathbf{s}_t = (T_t, \mathbf{p}_t, \mathbf{i}_t, \mathbf{e}_t) \quad \overset{a_t \in \mathcal{A}_{extr.}}{\longrightarrow} \quad \mathbf{s}_{t+1} = f(\mathbf{s}_t, a_t, \mathbf{w}_t) = (T_{t+1}, \mathbf{p}_{t+1}, \mathbf{i}_{t+1}, \mathbf{e}_{t+1})$$

is in this case represented by:

$$\begin{cases} T_{t+1} = T_t + time(a_t, \mathbf{w}_t) \\ \mathbf{p}_{t+1} = \mathbf{p}_t \\ \mathbf{i}_{t+1} = \mathbf{i}_t + \mathbf{c} \\ \mathbf{e}_{t+1} = \mathbf{e}_t \end{cases} \tag{5}$$

where $\mathbf{c}$ is a vector such that if x is the target resource and y the used tool:

$$c_x = 1, c_y = -1, c_i = 0 \ \forall i \neq \{x, y\}$$

Depending on the distance between the entity and the target resource block, extraction may also require a displacement. This is performed using the same path finding algorithm as for pure displacement actions and implies some uncertainty on the new position of the entity. As a result of this action, the same effects as for movement actions apply. The general case is thus:

$$\begin{cases} T_{t+1} = T_t + time(a_t, \mathbf{w}_t) \\ \mathbf{p}_{t+1} = \mathbf{p}_t + \epsilon(\mathbf{w}_t) \\ \mathbf{i}_{t+1} = \mathbf{i}_t + \mathbf{c} + \mathbf{\Delta i}(\mathbf{w}_t) \\ \mathbf{e}_{t+1} = \mathbf{e}_{t+1}(\mathbf{e}_t, \mathbf{w}_t) \end{cases} \tag{6}$$

Where:

- $\epsilon(\mathbf{w}_t) \in \mathbb{Z}^3$ represents the uncertainty on the final position.
- $\mathbf{c}$ is the vector defined earlier.
- $\mathbf{\Delta i}(\mathbf{w}_t) = (\Delta i_0, \cdots, \Delta i_{|\mathcal{O}'|-1}) \in \mathbb{Z}^{+|\mathcal{O}_r|} \times \mathbb{Z}^{-|\mathcal{O}_t|} \times \{0\}^{|\mathcal{O}_o|}$ models the possible effect on resources and tools during displacement. $\mathbf{\Delta i}(\mathbf{w}_t) = \mathbf{0}$ if no block has been destroyed to move.
- $\mathbf{e}_{t+1}(\mathbf{e}_t, \mathbf{w}_t) \in \mathcal{E}$ is the environment descriptor corresponding to the field of view from the new position $\mathbf{p}_{t+1}$ of the entity.

2.4.3 Construction actions

The game defines some recipes that describe how to craft new items. Constructing an item following a recipe implies two effects: some items will be consumed in order to create other ones. The effect has thus only an impact on the inventory.

To reduce the number of possible construction actions, only actions aiming at building tools can be taken by the agent. But in the game, constructing items require to craft multiple intermediate ones and some of these intermediate items may only be produced in a minimal quantity. This induces some border effects leading to creation of extra items from $\mathcal{O}_o$. These extra items are stored in the inventory and will be re-used to reduce the amount of resources consumed for future tool construction actions. This allows to comply to the basic mechanisms of the game without requiring the agent to learn the complete technology tree and building the intermediate items one by one.

Example 2.1. Figure 3 depicts the dependency tree that leads to the construction of an iron pickaxe. The numbers represent the amount of resource required and produced. Resources are represented in green. Items in orange are extra items while the item in red is never overproduced. The orange items have to appear in the inventory (and thus are part of the state) as they influence the action result. Conversely, the red item does not exist from the agent's perspective.

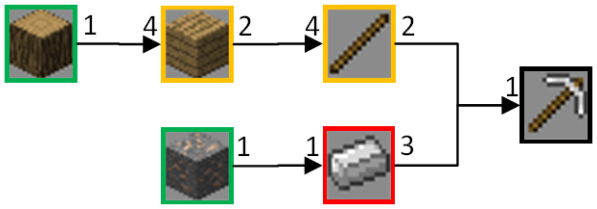


Figure 3: Dependency tree of the iron pickaxe.

We can describe recipes using pairs of inventories $\langle \mathbf{r}|\mathbf{b}\rangle$ where $\mathbf{r}$ is an inventory containing the required items and $\mathbf{b}$ is an inventory containing the produced items ($\mathbf{r}, \mathbf{b} \in \mathcal{I}$). $\mathbf{r}$ can only contain resources and intermediate items (elements from $\mathcal{O}_r \cup \mathcal{O}_o$) and $\mathbf{b}$ only contains intermediate items and a single tool (elements from $\mathcal{O}_t \cup \mathcal{O}_o$).

A construction action $a \in \mathcal{A}_{build}$ is represented by a set $\mathcal{R}_a$ of recipes aiming at building the same tool. For a given state $s \in \mathcal{S}$, action a is available if:

$$\exists \langle \mathbf{r}|\mathbf{b}\rangle \in \mathcal{R}_a : \quad \mathbf{r} \leq \mathbf{i}_t \Leftrightarrow r_x \leq i_x, \quad \forall x \in [0, |\mathcal{O}'| - 1]$$

If more than one recipe is available from s, the one which minimizes the amount of resources used will be selected.

The construction actions do not require any displacement and are fully deterministic, which means that, for the following transition

$$\mathbf{s}_t = (T_t, \mathbf{p}_t, \mathbf{i}_t, \mathbf{e}_t) \quad \overset{a_t \in \mathcal{A}_{build}}{\longrightarrow} \quad \mathbf{s}_{t+1} = f(\mathbf{s}_t, a_t, \mathbf{w}_t) = (T_{t+1}, \mathbf{p}_{t+1}, \mathbf{i}_{t+1}, \mathbf{e}_{t+1})$$

the changes in the components of the state vector are governed by

$$\begin{cases} T_{t+1} = T_t \\ \mathbf{p}_{t+1} = \mathbf{p}_t \\ \mathbf{i}_{t+1} = \mathbf{i}_t - \mathbf{r} + \mathbf{b} \\ \mathbf{e}_{t+1} = \mathbf{e}_t \end{cases} \tag{7}$$

2.4.4 Reward

The goal of the agent is to collect a given resource. This implies that the reward must be directly linked to the gathered amount of this particular resource. Collecting other resources is allowed but is not directly part of the goal, the reward will thus not directly depend on these. Another important parameter is time. The agent has to be efficient and to collect as many resources as possible in a given amount of game time. The reward thus includes time as a negative term, which assigns a penalty to time losses. These two aspects lead to the following reward function $r(\cdot) : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$:

$$r(\mathbf{s}_t, a_t, \mathbf{s}_{t+1}) = \beta * \Delta i_{goal} - (1 - \beta) * \Delta T \,, \tag{8}$$

with $\beta \in [0; 1]$ a weighting factor and

$$\Delta i_{goal} = i_{goal,t+1} - i_{goal,t} \,,$$
$$\Delta T = T_{t+1} - T_t \,,$$

i_{goal} representing the amount of the objective resource that is in the inventory of the agent.

SECTION 3

Algorithms

3.1 Reinforcement learning algorithm

3.1.1 Selection

One of the main difficulty for the agent to learn how the environment reacts to its actions is the size of the state space. In this particular application, the complete version of the state is made of 60 dimensions, which is quite large for reinforcement learning algorithms.

In addition, some actions are highly valuable but have a lot of requirements, increasing the difficulty for the agent to find out the succession of actions leading to fulfill the requirements.

Example 3.1. To gather efficiently resources, the best approach is first to build advanced tools like an iron pickaxe or at least a stone one. Doing so requires the following actions to be taken:

- Gather enough wood when the entity is above the ground;
- Build a wood pickaxe;
- Dig into the ground to reach stone;
- Gather some stone;
- Build a stone pickaxe;
- Eventually search for iron, collect it and build an iron pickaxe.

Decomposed into individual actions, this requires at least 8 successive steps, which is unlikely to occur and will thus not be easily learned by the agent without prior knowledge.

In order to help the agent dealing with the dimension of the problem and to allow it to find a good strategy, a dataset of expert (i.e. human-made) trajectories are generated and then used as a learning set for a batch mode reinforcement learning algorithm. The goal with this approach is to give some clues to the agent on how to achieve good rewards. Thanks to this method, the agent started with a good idea of a good strategy in various basic situations.

This method has the particularity to hide the negative decisions the agent can take. Indeed, a human player never tries to build a tool without having the required items or to go upward if he is already at the top of a hill. This is both a drawback and an advantage:

- it means the agent does not have any knowledge about the negative impact of useless or unauthorized actions;
- it also means that the agent is likely to diverge from the given trajectories. This way, it leaves to the agent the possibility to discover a better policy if it exists.

An example of this expected behaviour is depicted in Figures 4 to 7. These represent a Q-function, which is a function associating to each state s and action a a value $Q(s, a)$ representing the quality of this combination. The objective is to derive a state-action policy by choosing the action maximizing $Q(s, a)$. Note that this is an artificial Q-function for demonstration purposes only.

Figure 4 represents the view of the expert player using a given strategy that he considers as the best one. When the expert trajectories are generated and a new Q-function is learned from these samples, we obtain Figure 5 which is an optimistic approximation of the first one. After some exploration and learning, this optimistic view is refined and the optimistic behaviour tends to decrease. Finally, the agent might find a new strategy more efficient than the one shown by the player, like in Figure 7.

3.1.2 Fitted-Q iteration

The fitted-Q iteration algorithm aims at building an approximation of the long-term expected reward given the current state and an action the agent might want to perform. By choosing the action corresponding to the best long-term expected reward, the agent is thus able to take decisions with a long-term objective instead of simply taking the action with the highest instantaneous reward. The model built by the algorithm is used to approximate an optimal policy. Given recorded transitions from one state to another, it is able to determine the reward that a succession of states might give at best.

To do so, each step in the system is considered as a 4-tuple $(\mathbf{s}_t, \mathbf{s}_{t+1}, a_t, r_t)$ representing the transition:

$$\mathbf{s}_t \xrightarrow{a_t \in \mathcal{A}_{build}} \mathbf{s}_{t+1} = f(\mathbf{s}_t, a_t, \mathbf{w}_t)$$

At first, the approximation of the maximum long-term expected reward is null for each combination of state and action:

$$Q^0(\mathbf{s}_t, a_t) = 0$$

Then, each iteration will increase by one the number of steps taken into account in the approximation of the long-term reward. To predict the best expected reward from state $\mathbf{s}_t$ using action a_t with a horizon of i steps, the predictions of the expect reward with a horizon of $i - 1$ steps are reused such that:

$$Q^i(\mathbf{s}_t, a_t) = r_t + \gamma \max_{a \in \mathcal{A}} Q^{i-1}(\mathbf{s}_{t+1}, a)\,, \tag{9}$$

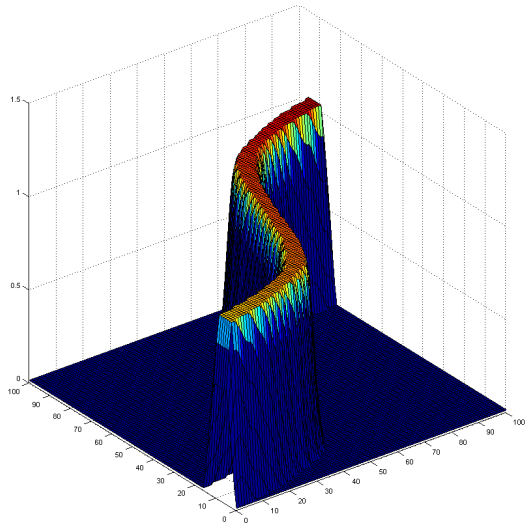


Figure 4: Q-function as seen from the player perspective.

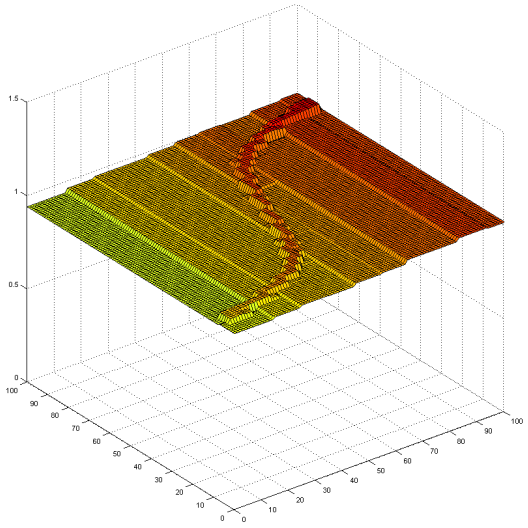


Figure 5: Q-function learned with expert trajectories only resulting in an optimistic view.

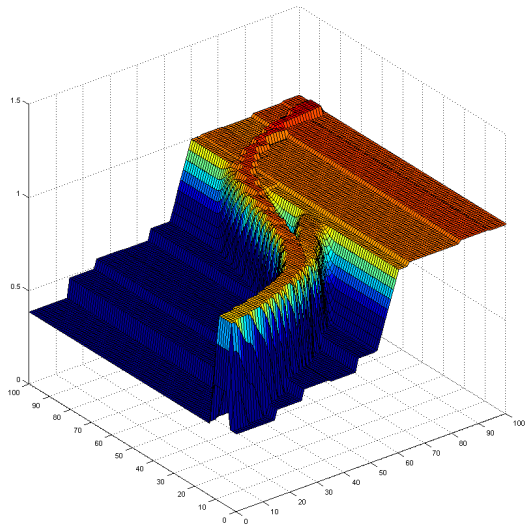


Figure 6: Intermediate Q-function during the learning phase, refined by the agent while exploring its environment.

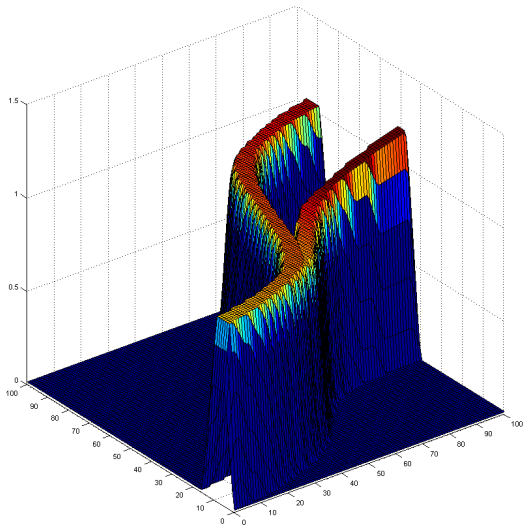


Figure 7: Resulting Q-function, with a new ideal trajectory discovered.

$Q^n(\cdot)$ being the expected reward after n steps and $\gamma < 1$ a discount factor ensuring convergence. At each step, a learning algorithm is used to predict the result of $Q(\mathbf{s}_t, a_t)$ over unknown combinations of $\mathbf{s}_t$ and a_t. This model is trained using the learning samples $(\mathbf{s}_t \| a_t, Q(\mathbf{s}_t, a_t))$. The final predictions, or more precisely the model built on it, allows to derive an estimation of the optimal policy by choosing the action a_{ideal}:

$$a_{ideal}(\mathbf{s}_t) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q(\mathbf{s}_t, a) \tag{10}$$

3.1.3 Model choice

To benefit from the expanding exploration described previously, the algorithm predicting the expected reward for the fitted-Q algorithm has to be optimistic over the unknown areas. This is why trees and forests were chosen preferentially. Indeed, these have the property to split the state space into clusters containing approximately the same quantity of data samples each. Unexplored areas will thus be predicted depending on their closest neighbours which are the data samples close to the explored area boundaries. As the model is first built on efficient realizations, this leads in the optimistic behaviour we want to obtain. Random forests are also interesting when the state space size is important as they don't require to perform a lot of tests at each split compared to other learning algorithms.

3.1.4 Impossible actions handling

The decisions cannot be made using only the state-action policy because it is deterministic. This means that if an impossible action is attempted in a given state and if the state does not change, which is probable in case of failure, the agent will try to perform the same wrong action again and again. To solve this issue, each decision step produces an array of actions sorted by the score assigned to this state-action combination. The action performed will always be the one with the highest score. If it fails and the state is still the same, the next action in the queue is chosen instead of repeating the whole decision process leading to the same action. This way, the agent is not stuck into an infinite loop of failure while there exists at least one feasible action.

3.2 Heuristic algorithms

The actions presented in the modelling of the game in section 2.4 are not directly available in the game. This level of abstraction must thus be provided by a software interface between the game code and the implementation of the decision process. This interface constructs high level actions based on the low level actions available in the game. This mainly concerns path finding allowing to compute which destinations are reachable and the corresponding fastest paths using only displacements from the current discrete position to an adjacent location. The information used in the state vector is another example of the high level view offered to the decision process.

This section will describe more in detail how this is done and which algorithms were used.

3.2.1 Path-finding

As already announced, the entity has the possibility to choose between multiple directions when it wants to move. Once a direction has been chosen, a path must be computed to go in this direction. But the environment has the particularity to allow modifications. This means moving in one direction does not only imply to find a way but in a lot of cases it also means create a custom way through the obstacles. These modifications are also costly in time, as it mainly requires to break the blocks on the way, and the cost of breaking a block is not the same for each block type.

As the objective was to design an agent with capabilities similar to a player, the entity also has only a limited view on its environment, implying some information is not available to compute the ideal trajectory.

These two additional properties, an alterable world with variable cost and a limited field of view, lead to the implementation of a heuristic based algorithm which tries to find a good path corresponding to the requested direction depending on the field of view of the entity, the surrounding blocks and the inventory which might contain useful tools.

Basic algorithm

When the entity decides to move in a specific direction $\mathbf{v}$, a fixed objective point can be computed as:

$$\mathbf{p}_g = \mathbf{p}_s + D_{move} * \mathbf{v}$$

with p_s the initial position, p_g the goal position and D_{move} the displacement length. As the world is composed of constant sized blocks and as we have both the starting point and the objective point, A* was chosen to be the base algorithm using the straight line distance as heuristic estimation of the cost between two positions. Nodes of the graph are all the position vectors $\mathbf{p} \in \mathcal{Z}^3$.

A* is an iterative algorithm that progressively builds a set of reachable nodes like Dijkstra does and keeps information about the links used to reach the nodes in this set. It starts with a set containing a single node which represents the current position. At each step, the node

which minimizes the cost to reach it and the heuristic cost of going from this node to the final destination is selected. We have:

$$n_{cur} = \underset{n \in \mathcal{N}}{\operatorname{argmin}}(cost(n_{start}, n) + distance(n, n_{dest})) \tag{11}$$

with $\mathcal{N}$ being the set of reachable nodes, n_{start} the starting node and n_{dest} the destination node.

From n_{cur}, the cost of travelling to all the one-hop neighbours is computed. If one of these neighbours is not in the set or if the cost to reach it through n_{cur} is lower than the previously recorded one, it is inserted in the set and the corresponding link is updated with the one from n_{cur} to the neighbour. Once all neighbours are evaluated, the current node is removed and a new node is selected in the set using the same criteria. Once the selected node matches the destination, the algorithm rebuilds the path by following the links associated with each node from the destination node to the initial one.

The main advantage of this algorithm is that it is more efficient than Dijkstra as we are only interested in going in a specific direction.

World edition

For a player, the world can be edited in two ways: by removing a block or by placing a new one. Removing a block consists in breaking it and the time required is sensitive to the tool used to do so. If the right tool is used, the block can also be collected once it is destroyed. The quantity of this block possessed can thus increase. Placing a block however implies much more complications as the block must be possessed before being placed and placing is subject to a lot of restrictions about the adjacent blocks. For example, it cannot be placed if there is no adjacent block.

Not using the ability to break block was not an option as it is one of the base mechanisms of the game. At the opposite, placing blocks is not a mandatory action. It can generally be compensated by one or multiple breaking actions. The implementation thus does not consider this possibility.

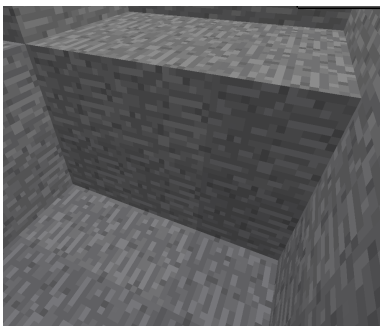


Figure 8: Obstacle.

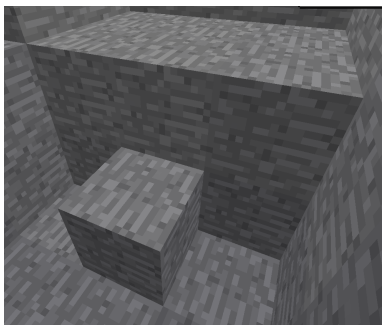


Figure 9: Solution by adding a block.

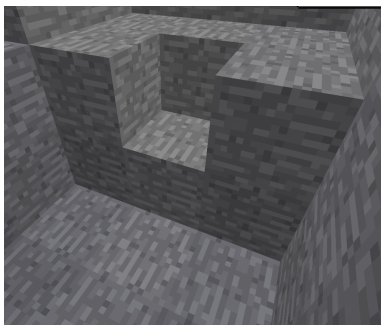


Figure 10: Solution by removing a block.

Allowing to edit the world implies to maintain knowledge about previous modifications performed when computing the way to each of the intermediate positions. This must be done to avoid using wrong information coming from the unmodified version of the world. For each reachable node, a virtual instance of the world is memorized. At each step, when A* selects a reachable node, its neighbours are computed from the virtual instance of the world assigned to this node. For each neighbour reachable from the current node, the virtual world instance resulting from the displacement is recorded for next iterations.

The information contained in the virtual world instances is also used when the path is complete to know effectively which blocks need to be removed at each step of the selected trajectory.

Field of view limitation

To stay consistent with the field of view limitations of the entity, path-finding must deal with this constraint too. This is why, once A* reaches the boundaries of the field of view, it is interrupted. The selected path will be the one allowing to be as close as possible to the target position, which is the one minimizing the remaining distance to the objective and already explored in previous iterations.

This constraint thus implies to re-compute the best path multiple times and decompose the global trajectory into smaller pieces. Nevertheless this decomposition might lead to a live-lock. The entity computing a path from position A to position C might interrupt the algorithm when an intermediate point B at the border of its field of view is selected. Once at position B, the second part of the trajectory is computed but stops at position A, which might not be visible anymore. This kind of situation is depicted at Figure 11. The resulting behaviour will be an oscillation between A and B. To solve this issue, the visible area is accumulated through the entire displacement. In this kind of situations, blocks visible from position A will thus still be considered as visible despite they are not once the entity reached B. The cache containing the visible blocks is cleared once the final position C is reached.

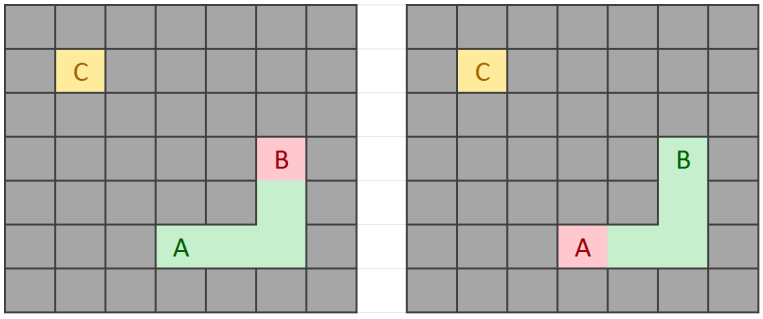


Figure 11: Example of live lock. Grey cells are breakable walls.

3.2.2 Field Of View

To be able to determine whether or not blocks are visible by the entity, a field of view computation had to take place. To do so, ray-casting was used. Ray-casting consists of

computing the intersection between a line coming from the entity's eye position and the first opaque surface. This is repeated several times in all directions using a small angular increment. This computation might require a long time if the result has to be very precise but within the discrete environment considered, this is not the case and the computation can be accelerated by only checking for obstacles on block boundaries.

3.3 Architecture

One of Minecraft's strengths is to let the community participate in the game development by allowing players to build mods. Mods are packages containing new blocks, new items or new entities inserted in the game to extend it. This eases the addition of new features in the game by accessing directly the game's code. In this particular application, the Minecraft Forge [8] development environment has been used.

To insert the agent in the game, choice has been made to create a new remotely controlled robotic entity. This allows the player to stay free and inspect the behaviour of the controlled entity.

3.3.1 User interface

Bootstrapping the learning phase using expert trajectories requires to generate the first learning set, LS_0, thanks to a human player. This means the player must have the same capabilities as the agent. For example the field of view of the player which is normally limited by the graphic rendering options required to be truncated to 10 meters.

Building a tool also takes a long time for a player using the game's default GUI compared to the intelligent entity building it in an insignificant amount of time. This thus required to implement in the game a new user interface allowing the human player to control the recorder. Thanks to this interface, the player can use the same algorithm as the agent would to create instantaneously tools (granted that the required resources are possessed). The same user interface is used to inform the recorder if a movement is performed to gather an in-sight resource block or if the movement is made to explore the world, which is completely different from the agent's point of view.

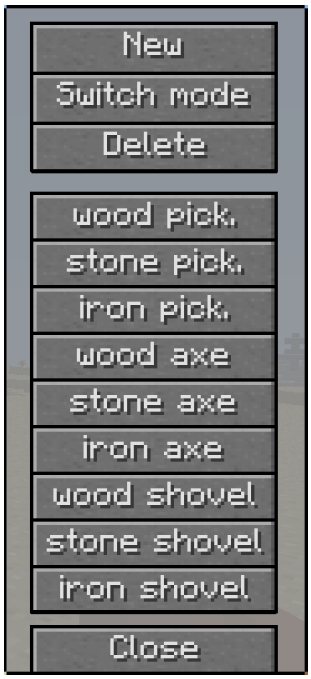


Figure 12: In-game recorder GUI.

3.3.2 Communication

The extra-tree implementation used is the one from scikit-learn. This library runs using python and some dependencies making it difficult to directly interact with java code. The implementation is thus split into 2 parts:

- A python server which takes all the decisions about which action to perform.

- The java code embedded in the game which holds the implementation of the heuristic algorithms and performs pre-processing of the in-game data.

Each time a new artificial entity spawns in the game, a network connection is established with the python decision server and is kept alive until the end of the game to avoid overhead of a creating a new connection multiple times. The communication protocol is simple. The embedded code generates a state vector of a predefined size and transmits it to the server. The response is an integer between 0 and $|\mathcal{A}| - 1$ corresponding to the index of the action.

To keep the responsibility of the decision making on the server's side, the python code is responsible of the ϵ-greedy behaviour. The custom module used to interact with the scikit-learn tools can also be loaded directly from a python terminal to trigger learning phases on new samples and backup new decision engines.

SECTION 4

Results

4.1 Methodology

4.1.1 Learning process

As fitted-Q is a batch mode algorithm, the learning process can be decomposed in steps, each step leading to a new dataset and a new model of the system. The initial step relies only on the expert trajectories. The learning set LS_0 regrouping the samples generated by an expert is used as starting point for the construction of a model Q_0.

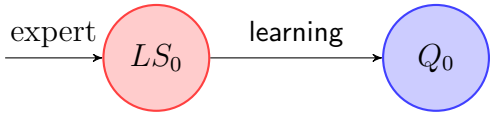


Figure 13: First step using bootstrapping.

The next steps contain two phases: exploration and learning. Figure 14 presents the sequence of operations required to complete a step. Starting from a model Q_i, the agent performs multiple rounds of simulation each round representing a realization. These rounds generate samples, which are regrouped into a new set of data S_i. The union of S_i and LS_i forms the new learning set LS_{i+1} used to build the next model Q_{i+1} by applying fitted-Q from scratch.

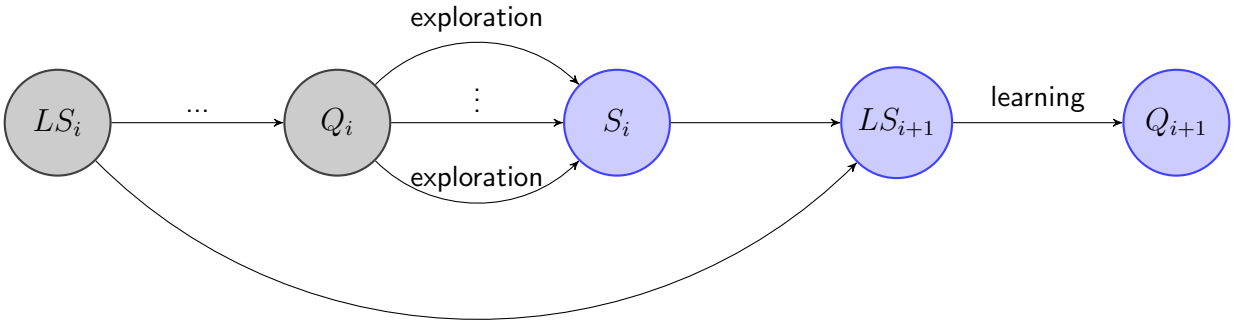


Figure 14: Processing of one step.

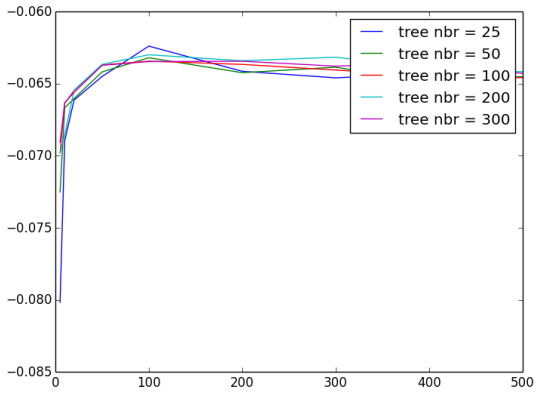
Despite the process is bootstrapped using expert samples, it still needs to keep exploring its environment. This is why the exploration phase of each iteration uses an ϵ-greedy policy on Q_i to generate the new samples.

4.1.2 Model complexity evaluation

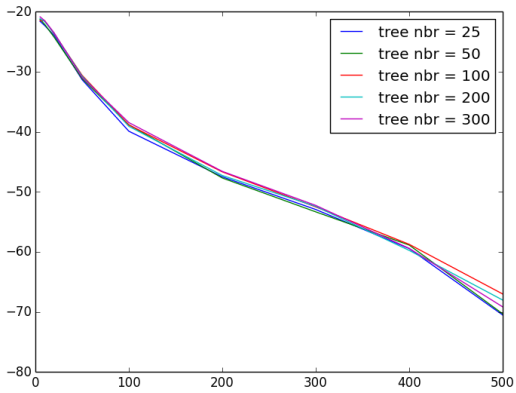
For the constructed model to be reliable with the system dynamic, the possibility of overfitting or underfitting the data must be taken into account. To avoid these unwanted side effects, a standard solution is to control the complexity of the generated model to obtain the best one. Here, the best one means the one complex enough to obtain an efficient model but general enough to be robust against noise in the learning samples.

To obtain a good idea of the ideal complexity, k-fold cross-validation was used on the samples set. K-fold cross-validation consist in splitting the samples into k subsets such that each subset is predicted using a model trained over the $k-1$ remaining subsets. By measuring the mean error committed by the k trained models, we obtain a value representative of the error we might expect by using the tested model. Repeating this operation using different parameters also allow to have hints about the ideal values to use.

The cross-validation was used multiple times on the hole training dataset before applying the fitted-Q algorithm. The objective was to adapt the complexity while the agent explores the state space. Given the size of the dataset, 3-fold cross-validation have been used with a mean squared error metric.



(a) Cross-validation results on LS_0 (containing only expert samples).



(b) Cross-validation results on LS_2 (containing the expert samples and the same amount of agent samples).

Figure 15: Cross-validation results.

Results of the cross-validation are depicted on Figure 15. The varying parameter is the minimum number of samples to split a node of the tree as the tested models are based on Extra-trees.

We can observe that the ideal parameter value is initialy around 100. This value is quite high but this is due to the way the samples were recorded. The initial samples are all generated by an expert using the same policy and thus share a lot of information. The generated trees can thus keep them grouped in the leaves and be more robust against noise.

After inserting samples generated by the agent while exploring new areas of the state space, the ideal value decreases to a very low value. When exploring, the agent generates

samples completely divergent compared to the ones given initially. This means that the model now needs to be more precise to predict correctly the behaviour of the system.

When applying the same test on the following iterations, the results are similar to Figure 15b. We can thus conclude that the ideal complexity of the model starting from a biased dataset tends to increase and reach a complexity close to the one of the modeled system.

4.2 Practical application

The application of this process first required to generate the first learning set, LS_0, thanks to a human player. A total of 30 games of 20 to 30 minutes have been recorded using this technique. In this period of time, the player gathered about 128 units of coal. All these games represent a total of 6.300 samples.

Using the process described in Section 4.1, the agent was then trained through multiple steps. About 8 games of 30 minutes have been recorded at each step, leading in about 2000 to 3000 samples. The number of games was chosen such that approximately $\frac{1}{3}|LS_0|$ were incorporated in the initial learning set at each step. This is a tradeoff to avoid losing too much time exploring the same area of the state space and to avoid rebuilding a new model with too few reliable data compared to the previous sample pool.

4.2.1 Learning

As already exposed, at the beginning of the learning phase, the agent does not have any knowledge about what happens if an impossible action is taken. This means it is likely to try them and an adequate feedback must be provided. The first approach was to freeze the agent for 5 seconds as a light penalty, causing it to lose time and resulting in no state modification. The outcome of these actions is thus a negative reward.

After performing some learning iterations, it turns out that this is not high enough as a penalty. The agent did limit the use of these bad state-action combination but was still using them regularly as they do not highly degrade its reward. This is because at the beginning, the agent often start by digging in the ground without any tools resulting in actions taking a lot of time up to more than 5 or even 10 seconds for worst cases. The agent thus considered in some situations that taking impossible actions and not moving anymore is the best option available. Obviously it is not the desired behaviour.

The second attempt used a high penalty without any time freeze. This penalty was set such that it is 50 times worse than the longer possible action. This way, the agent had no choice but to choose a feasible action. The continuous curves on Figure 16 to 21 represent the performance evolution of the models generated at each iteration. These performance tests were performed using a completely greedy policy ($\epsilon = 0$).

During the learning phase, the agent went through different distinct behaviours. The first policy used by the agent is to always choose the action of gathering coal, which is impossible while no coal is in sight but had always a positive score in the expert samples. This is why the percentage of failed actions is close to 100% at first iteration. The agent then progressively improved its strategy.

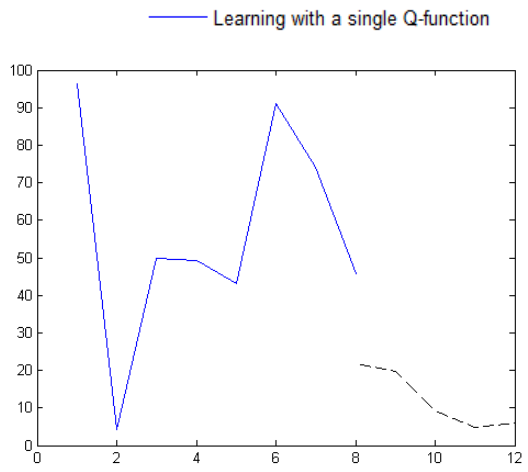


Figure 16: Percentage of missed actions.

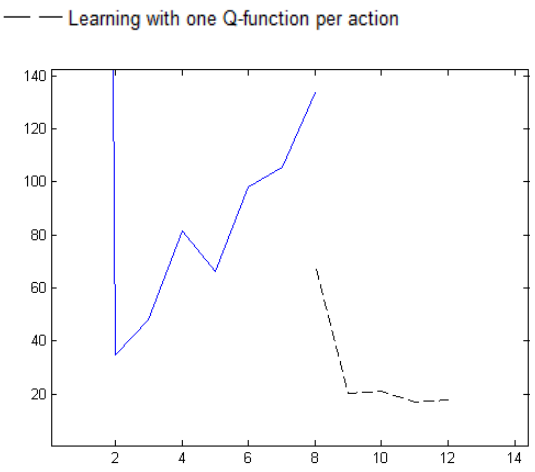


Figure 17: Number of missed actions after 20 minutes.

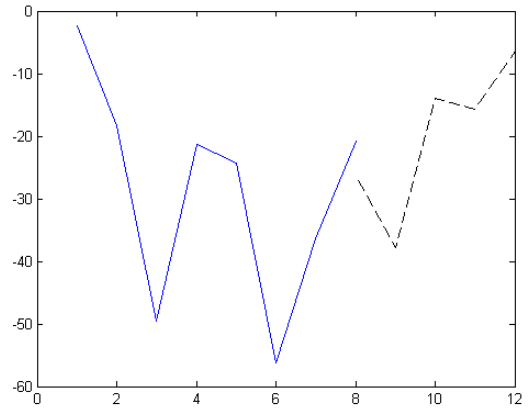


Figure 18: Accumulated reward after 100 actions (without failed actions penalty).

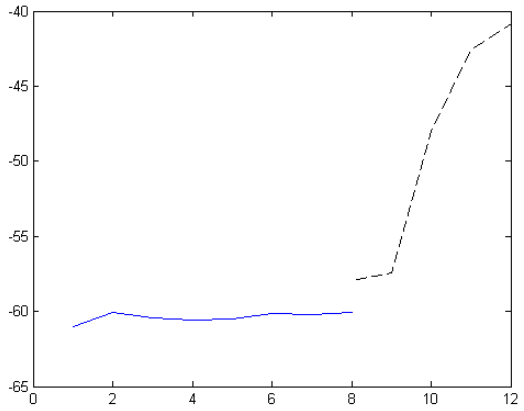


Figure 19: Accumulated reward after 20 minutes.

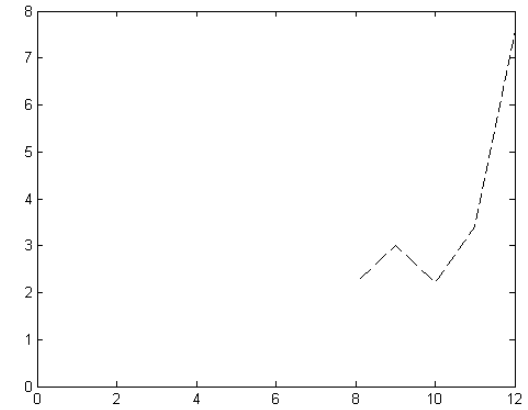


Figure 20: Coal gathered after 100 actions.

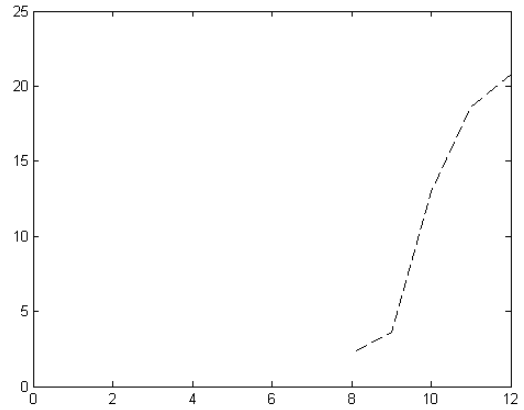


Figure 21: Coal gathered after 20 minutes.

It tried to go underground without tools. This allows to reduce drastically the amount of missed actions as these actions take a long time without tool but also decrease the accumulated reward over 100 actions, as time is part of the reward. Progressively it decided to build tools, often without having the correct items, to collect wood, etc.

After 8 iterations, the agent had an increased knowledge about possible and impossible actions. However, it was not able to select a precise action. From the agent perspective, actions are represented by their index, a number between 0 and $|\mathcal{A}| - 1$. These action indexes are sorted by action type such that displacement actions are represented by the lower indexes, the crafting actions are represented by the higher ones and the gathering actions are represented by intermediate ones. The result of this implementation detail is that the learning algorithm fitting a single Q-function on the data does not differentiate correctly the actions with close indexes while they have completely different goals. As a result, once the entity finds some coal and should ideally gather it, the decision engine often begins by trying to collect iron, stone or even dirt because the score attributed to all the gathering action is similar and too sensitive to noise. The first learning steps also recorded a lot of failed tentatives to get coal using the corresponding action, which might lead to reduce the importance of the positive samples from the expert and underestimate the potential of this action. This is why the percentage of failed actions on Figure 16 is still high even after 8 steps.

To solve this issue, the way to process the data was adapted to fit one different Q-function per action, as suggested by [1]. This seems legitimate as they all have completely different requirements to be feasible and completely different effects. It also implies to change slightly the fitted-Q algorithm. First, the samples are splitted into groups of samples resulting from the use of the same action. For each group, a different model is built, leading to $|\mathcal{A}|$ models to represent the whole Q-function. At each iteration of the fitted-Q algorithm, the state to long term expected reward matching must be updated based on the maximum of the prediction from each of the models instead of concatenating the action id to the state.

This modification resulted into models much more precise and specialized on a single action. Taking back the learning set generated by the previous method, LS_8, the resulting state-action policy was already able to collect some coal while the previous one trained on the same dataset did not. This new methodology is thus the one which was used for the last iterations of the process. It corresponds to the dashed curve on Figure 16 to 21.

Note that running the whole learning process takes a lot of time as it requires to simulate multiple games and run fitted-Q on an increasing learning set. This is why the learning set generated by the single Q-function method was reused to obtain an estimation of the perfomance we might expect using the second method. These results should ideally be completed by results obtained when applying the second method from LS_0 to confirm the effeciency of the method when used from scratch. The time remaining before the deadline did not allow to do so.

4.2.2 Performances

The actual performances of the agent are lower than the expert results. In about 20 minutes, an expert player can gather about 100 units of coal while the agent only collects about 25 units. This can be explained by two key points.

At first, the decision process is stationary, which seems to be a bit inefficient in this particular application. For example, Figure 22 shows the field of view of an entity collecting coal from a given position (point I). Once all the visible blocks of coal are gathered, the agent decides to move to position I+1 (Figure 24) and dig a new gallery but cannot see that there are other blocks of coal nearby as it is completely blind during its displacement. In this particular case, a player would inspect the area to be sure there is no more units to gather, thus increasing highly his efficiency. This is due to the lack of information about the previous actions performed by the entity. Nothing in the state vector gives information about the presence of coal in the previous steps.

The second handicap of the agent compared to a real player is that it does not explore the existing galleries efficiently. Human players often use these existing galleries to explore the area faster and it often happens that coal is embedded directly in the walls. For this purpose, the heuristic displacement costs and the view distances were given in the state vector but the learned policy seems to prefer using another method. Another explanation might be that this information is contained in a lot of variables (28 values) and has a high variance, which might lead to be considered as noise.

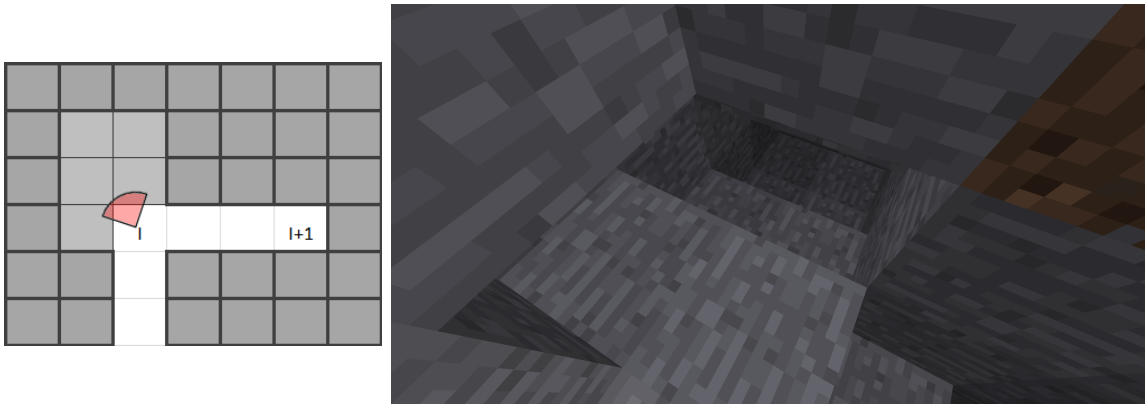


Figure 22: Situation after collecting all coal blocks in sight.

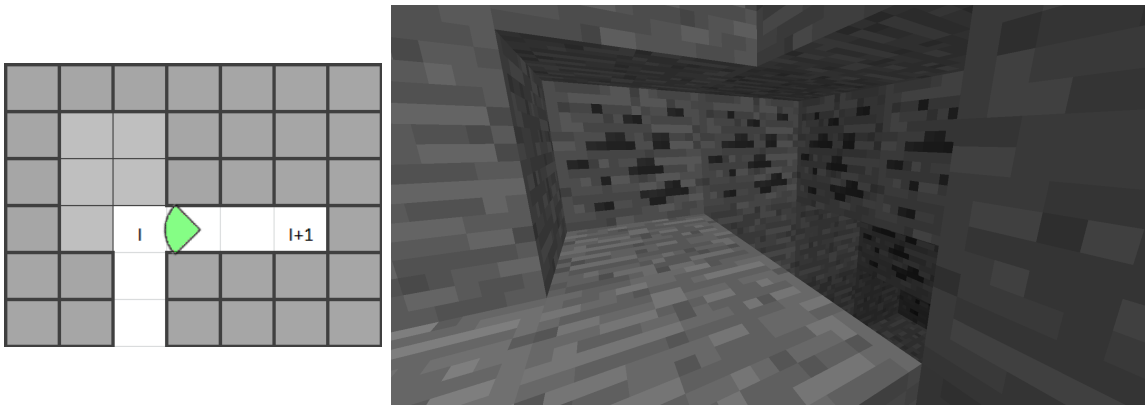


Figure 23: Intermediate view during the displacement action.
Other coal blocks are revealed.

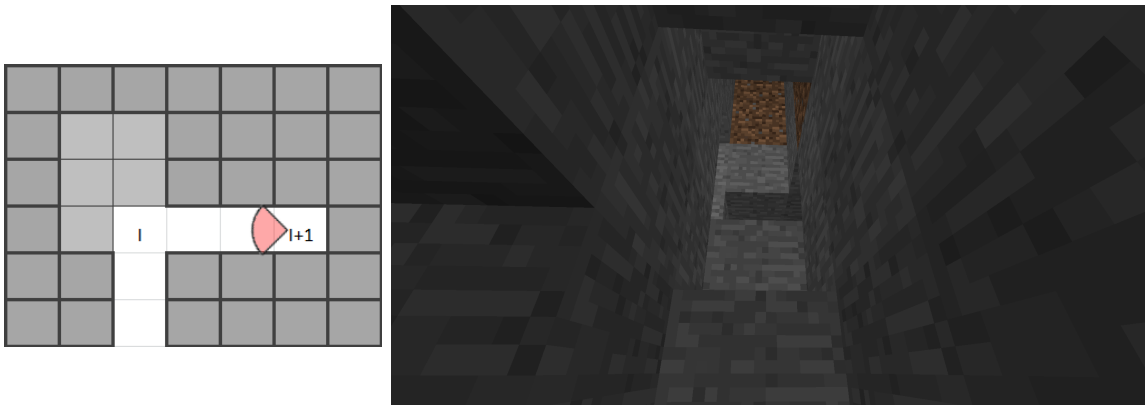


Figure 24: Situation at the end of the displacement action. Coal is not visible anymore.

SECTION 5

Conclusion

From the obtained results, we can conclude that despite the size of the state space, the combination of fitted-Q with extra-trees when fitting one Q-function per action succeeded in learning the system dynamic. The resulting agent is able to build the adequate tools, does not build the useless ones and even builds higher level tools when the resources are available.

However, it was not able to do so using only one Q-function and cannot compete with the player score because of the stationarity of the decision process.

5.1 Future work

First of all, the learning process used to produce the results presented in this work is a combination of two methods. To confirm the results, the learning process should be restarted from LS_0 using one Q-function per action to see if it leads to a similar evolution as the one observed. It might be more efficient and converge much faster to a final policy or less efficient as using one Q-function per action increases the complexity of the model and might lead to overfitting.

Another interesting test is to restart the learning process without using the initial expert samples. This would allow to conclude whether or not these are mandatory for the agent to discover the sequence of actions leading to collect coal.

5.2 Possible improvements

A highly valuable improvement would be to solve the problem of missed coal explained in the performance comparison. A possible way to do so is to modify the displacement actions execution such that it is interrupted when coal appears in the field of view. This will modify the behaviour of the system from the agent point of view but allow the agent to react correctly in this kind of situations. An alternative might be to modify the model and include information about the presence of coal in a specific direction during previous steps but it would also increase a lot the size of the state space which is already quite important.

Another improvement perspective is to generalize to other resources like iron or gold. These resources are harder to find and gather but the basic mechanisms are similar to coal.

Bibliography

Academic documents

[1] Damien Ernst, Pierre Geurts and Louis Wehenkel, "Tree-Based Batch Mode Reinforcement Learning", Journal of Machine Learning Research 6 (2005) 503-556.

[2] Karl Tuyls, Same Maes and Bernard Manderick, "Reinforcement learning in large state spaces", Computational Modeling Lab, Departement of computer science, VUB.

[3] Quentin Gemine, Firas Safadi, Raphaël Fonteneau and Damien Ernst, "Imitative Learning for Real-Time Strategy Games".

[4] Richard S. Sutton and Andrew G. Barto, "Reinforcement Learning: An Introduction", MIT Press.

[5] Raphael Fonteneau, Susan A. Murphy, Louis Wehenkel and Damien Ernst, "Model-Free Monte Carlo-like Policy Evaluation".

[6] Shie Mannor, Ishai Menache, Amit Hoze and Uri Klein, "Dynamic Abstraction in Reinforcement Learning via Clustering", Faculty of Electrical Engineering, Technion, 32000 Israel.

Web resources

[7] Minecraft website, https://minecraft.net/.

[8] Minecraft forge forum, http://www.minecraftforge.net/forum/index.php.

[9] Minecraft forge French forum, http://www.minecraftforgefrance.fr/.

[10] Techne, the minecraft modeler tool, http://technemodeler.tumblr.com/.

Illustrations

[11] Minecrafter camp, http://www.minecraftercamp.com/.