Universidade de Lisboa - Instituto Superior Técnico

Departament of Mathematics

BSc. Applied Mathematics and Computation

Monte Carlo Tree Search Algorithm

Author:
Ricardo Quinteiro
83653

Supervisor:
Professor Pedro Alexandre Santos

September 24, 2018

Contents

1 Introduction 2

2 Background of the Algorithm 3

3 Monte Carlo Tree Search Algorithm 5

3.1 MCTS 4 Main Steps 5

3.1.1 Selection 6

3.1.2 Expansion 6

3.1.3 Playout 6

3.1.4 Backpropagation 7

3.2 Some Problems Related to this Algorithm 7

3.2.1 Rapid Response Time 7

3.2.2 Very Large State Spaces 8

3.2.3 Limited Playout Budget 8

4 First Practical Test of the MCTS 10

4.1 Explanation of the Context 10

4.2 Application of the MCTS in this Game 11

4.2.1 Discussion about the Number of Iterations and *Selection* and *Playout* Depths 11

4.2.2 Discussion about the *Playout* heuristic 12

5 MCTS in Don't Starve Together 13

5.1 A Few Details About Don't Starve Together 13

5.2 Tools Used to Interact with the Game 13

5.3 MCTS Approach Based on Game Properties 14

5.3.1 Pre-World State, World Model and Limited Number of Actions 14

5.3.2 Pre-World State Abstraction 15

5.3.3 World Model Abstraction 16

5.3.4 List of Possible Actions in the Model 17

5.3.5 Macro-Actions 18

5.4 Discussion about the Number of Iterations and *Selection* and *Playout* Depths 19

5.5 Discussion about the *Playout* heuristic 20

6 Conclusion 22

1 Introduction

This project main objective for me was to get introduced to some Artificial Intelligence concepts, and in this case specifically to the algorithm *Monte Carlo Tree Search*, and to use them in a practical context. I also wanted to learn how this algorithm functions and to understand the changes that need to be done in order make it work well.

As my field of study is mathematics and it serves as a background for this algorithm, it made perfect sense that I should learn more about it. Also, my curiosity about this subject and its vast applications in nowadays activities led me to the choice of doing this project.

This work was divided in three distinct phases. The first one was to search and learn the algorithm in question and some theory behind it. The second was to test it in a game with few actions, in order get more practical knowledge of the algorithm and how some changes influence its result. The third, and most important of them, consisted in applying the algorithm to make decisions in the game *Don't Starve Together*. This is a widely known survival game and, opposed to the previous one, this is complex due to the variety of actions that are available and the unpredictability of the world state.

Doing this project, my expectations were to become more aware of how Artificial Intelligence works and to learn in which ways it can be used. Even though this is more related to video games, I hope this enables me to have a more general idea of how to use this approach in real-life circumstances.

2 Background of the Algorithm

There are a few theories and problems that were investigated and studied before the MCTS emerged. Some algorithms use similar structures and methods to that of the MCTS and it is important to know a bit of the theory behind it to be able to make the changes needed for it to operate correctly. Bellow there are some introductions to these themes, which are a good starter for understanding the algorithm in question.

Decision theory [1] [3] [6] is the mathematical study of strategies for optimal decision-making between options involving different risks or expectations of gain or loss depending on the outcome. It uses probability theory and utility theory to base the decisions on both evidence and the agent's will.

Game theory [1] [3] [6] is an extension of the *decision theory* that incorporates multiple agent contexts in which they interact with one another. It has the following components:

- $S = \{s0, s1, ..., sn\}$: a set of states
- $A = \{a0, a1, ..., ak\}$: a set of actions
- $T : S \times A \rightarrow S$: a transition function
- $R(s)$: a reward function
- $S_T \subseteq S$: a set of terminal states
- $m \in \mathbb{N}$: the number of agents
- $\rho : S \rightarrow (0, 1, ..., m)$: a function that returns the player to act

Only the last three components differ from the *decision theory*'s.

A *policy* [6] is a function $P : S \rightarrow A$ that, in each state, returns an action depending on its properties.

Games start at a state s_0 with a defined player. For a state s_t the player defined by ρ selects an available action a_i from A. The transition model is applied to (s_t, a_i) to reach the state s_{t+1}. The process is repeated until a terminal state is reached. The way an agent selects an action in a state is defined by the *policy*. Usually the policy will select an action that has a high probability to lead to a state s_t with a high reward.

Theses frameworks can be utilised to help define a world model for the MCTS to run. Also they help understanding how the simulation of the algorithm should work.

Bandit Based Methods [1] [6] are a class of sequential decision problems on which *Monte Carlo Tree Search* is based. They try to solve the problem of maximising the cumulative reward when there are several different actions that can be done at a given point and finding the optimal sequence of actions.

In this context, exploration is trying a new move instead of doing one that has already been tried. Exploitation is exactly the opposite, which is trying the action that has the highest reward instead of trying a new one. The difficulty in these problems is to find an ideal balance between exploration and exploitation because the reward distribution is unknown.

Upper Confidence Bounds [1] (UCB) are a solution to this problem. UCBs are mathematical formulas that select the action that should be performed by maximising its value. These formulas use the current value of an action, the number of times it was played and the total number of times the simulation was done. Their value has a component based on exploitation and another based on exploration.

Upper Confidence Bounds for Trees (UCT) is a variation of the UCB1, which was proposed by Auer [1] [6], used for trees. As the MCTS is a tree-based algorithm, the UCT is used in the *Selection* process to solve the problem of the *exploration vs exploitation*.

$$UCT = \overline{X}_j + C\sqrt{\frac{\ln n}{n_j}} \tag{1}$$

In the UCT, $\overline{X}_j$ is the current value of an action, n is the total number of simulations, n_j is the number of times the given action was tested and $C > 0$ is the exploration parameter. If C is bigger, the exploration will be favoured in relation to exploitation.

3 Monte Carlo Tree Search Algorithm

Monte Carlo Tree Search is a tree-based algorithm that runs simulations of future states, starting in the initial state. It stops at a determined state (which can be the final state, in case there is one), and then, after evaluating the status of it, the algorithm goes back until the initial state and propagates this evaluation to the successive states.

Each node of the tree has a statistic that is used to measure the quality of the action chosen to get to that node. This statistic is updated when the evaluation is propagated to a node.

This procedure is run several times to make the statistic more reliable and, after that, an action is chosen based on the results.

In order to do this efficiently, it uses 4 steps. Each one of them has a different procedure.

3.1 MCTS 4 Main Steps

The four steps of MCTS are *Selection*, *Expansion*, *Playout* and *Backpropagation*.

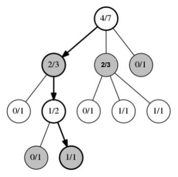

Figure 1: Selection

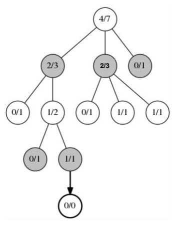

Figure 2: Expansion

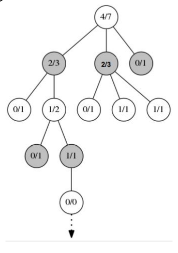

Figure 3: Playout

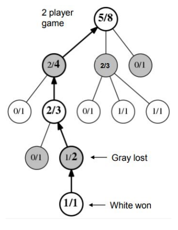

Figure 4: Backpropagation

3.1.1 Selection

Selection is the first step to take in MCTS and it lasts while the nodes have statistics to select the move to take. In the example above (3.1), the selected nodes are circled with a bold line.

In that case, it shows the statistics of a two players game in which the white nodes correspond to one player's actions and the gray ones to the other player's. The circles contain a fraction whose numerator (Q) and denominator (N) correspond, respectively, to the number of times the selection of that node resulted in a win and the total number of times that node was selected. Sometimes the value of Q is a real number between 0 and 1 due to the inability of reaching a final state in the game.

These values are the ones who will serve as a statistic to measure the quality of an action. The way to do so is by using the UCT Selection (1) which balances the decision between exploitation and exploration. This means that the UCT, based on the values of the child nodes, will likely select the node with the best results but will not completely ignore the other nodes. What this prevents is a case in which the best move eventually leads to a loss but, because not all the options were weighed, this information is not correct because, more often than not, it will lead to victory.

3.1.2 Expansion

Expansion begins when there is no statistical information to choose a given action. This happens when the algorithm gets to a point in which there are actions that were not tested yet.

This step is the simplest of all and what is done here is just initializing a new child node. The action that leads to the new node is selected randomly and it starts with both values at 0.

3.1.3 Playout

In the *Playout*, a simulation of the game is run and the actions are randomly selected or a heuristic can be used to make this selection less random. This step consists of moving down the tree until the end of the game or until a predefined depth of the tree is reached. After that, a reward is calculated and it will be backpropagated in the next step.

The *Selection* and *Expansion* have also a depth limit. Even though the *Playout* can be done until the game reaches an end, this will not work for games with infinite state spaces. Also, sometimes the game does not necessarily have a victorious end. This is the case of *Don't Starve Together*.

In the case that the depth of the *Playout* is chosen, the reward should be measured in a different way. It can be calculated using a heuristic and based on the world state that was attained. Its value should be in the interval $[0, 1]$, otherwise the constant in the UCT has to be altered.

3.1.4 Backpropagation

The next step is the *Backpropagation*. It consists of taking the value of the reward and a node, increasing the node's number of visits by 1 and adding the reward to the value of Q in that node.

After finishing the *Selection* and *Expansion*, the result is a node that is then used as an input for the *Playout*. The *Backpropagation* starts in this node and goes all the way up in the tree until it reaches the initial node while doing the procedure explained before.

By doing this, a part of the path that leads to the last node in the *Playout* is entirely evaluated based on the properties of the state in that node.

3.2 Some Problems Related to this Algorithm

This algorithm, in its simplest form, is not actually useful in most of the situations we would like to use it. In spite of this, it can be reconfigured, either by using heuristics or by making some adaptations in its steps, to function with different domains.

I will enumerate and explain below three limitations this algorithm has and how they can be overcome.

3.2.1 Rapid Response Time

This problem occurs when an action is needed quickly. In a survival game, such as *Don't Starve Together*, taking too long to decide can be a setback and it needs to be analysed in order to be properly dealt with.

Because MCTS can be stopped at any time, one solution is to do precisely that and return the best action at the moment.

As the previous alternative may not to be so reliable, another approach is to balance the quickness and the depth of the algorithm. What this means is that, in order to fix this issue, one should find an ideal number of times for the algorithm to run and its depth. Sometimes, even though it may seem counterintuitive, it is actually better to reduce the depth of MCTS (see Sections 4.2.1 and 5.4).

3.2.2 Very Large State Spaces

This problem has two different strands: High Branching Factor and Infinite Action Space.

The first one is a common problem with tree algorithms. What happens is that when you start searching down the tree, it will get more complex. For example, if you have four different actions and they can all be done at any point, at depth three there will exist $4^3 = 64$ different states. This growth is exponential and so, it is very difficult to try every different possibility as the depth of MCTS is increased.

The MCTS already uses the UCT in the Selection so that the choice of the child node is less random. However, a solution for this issue is using other heuristics to choose even less randomly from the child nodes.

The second problem occurs when there are infinite possible actions, or even a large amount of them.

The solutions for this are based on selecting a shorter amount of actions, which can be done in different ways. One of them is to select a subset of the action space which covers it uniformly. When this cannot be easily done, another way to do so is by using a heuristic to create a sample of the action space.

Another solution, which can be used for both of these problems, is creating macro-actions. Macro-actions are not usually in the action space but instead they are a subset of atomic actions that are supposed to be done in a certain order. For example, the sequence of actions *'Walk to the river'*, *'Equip a fishing rod'* and *'Catch fish'* can become a macro-action called *'Fish'*.

3.2.3 Limited Playout Budget

In some contexts, it is impossible to run a *Playout* until the end of the game. Since the *Playout* simulates the game, if it has too many features MCTS can be very slow to do so. When using the MCTS in real-time decisions this is totally the opposite of what is expected, which is fast decision making.

In order to make it run efficiently, when this happens, instead of running the *Playout* until the end, it can be stopped at some point (usually at a predefined depth) and using a heuristic to calculate the reward of that state.

Another way of fixing this is by creating a simpler model of the game world. This model should have less but concise information about the state of the game. Having a model with these characteristics makes the *Playout* run faster.

These two solutions can also be combined together to improve the speed of the *Playout*.

4 First Practical Test of the MCTS

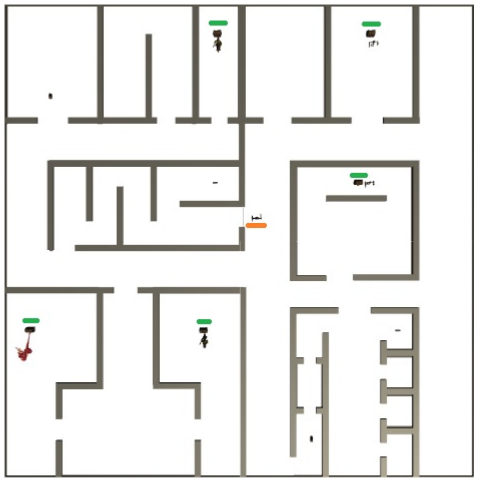

Figure 5: Image of the game used to test MCTS

4.1 Explanation of the Context

For my first practical approach, I was given a simple and deterministic game for which I had to develop the decision making of the only playing agent. The game was developed in Unity and it was originally created to be used in the course *Artificial Intelligence for Games*.

This game consists of a character (underlined with orange in Figure 5) that has to move in a maze filled with monsters, chests and potions. The goal is to collect all the money (25 in total), which is equally distributed by the five chests (overlined with green in Figure 5) that are spread around the map, without getting killed and below 200 seconds. The somewhat difficult part of the game is that these chests are each protected by a monster.

There are three different kinds of monsters, each one having their own attack damage and health points (HP). The main character has three different levels, starting the game in level 1 and possibly moving up to level 3. Killing the monsters gives the experience points to level up and this upgrade changes the maximum HP and restores the character to full health.

4.2 Application of the MCTS in this Game

For this game, I used a simple heuristic to calculate the reward in the *Playout*. This heuristic returns a value between 0 and 1 and the idea is to calculate this value using the game state properties, such as the character's HP, the time that has passed since the start of the game, the total money collected...

Another thing must be taken into account, which is the algorithm's tree depth and the number of iterations done by it. In order to get better and fastest results, these factors must be optimized.

4.2.1 Discussion about the Number of Iterations and *Selection* and *Playout* Depths

As was said before, Monte Carlo Tree Search has some upsides and downsides to it. In this particular discussion, it is important remember one problem of this algorithm: the limited playout budget (3.2.3).

Even though it is feasible to test this game until the end, this is not necessarily good. I tested the algorithm with the following combinations:

Case	Iterations	Selection Depth	Playout Depth	Time (s)
(A)	50000	5	5	-
(B)	1000	5	5	-
(C)	1000	2	2	112
(D)	1000	1	1	100
(E)	20	1	1	100

In the situation (A), the decisions the MCTS was outputting were good but as it was too slow, I stopped running it. In the situation (B), in 5 tries, the character always died. In case (C) it finished the game in 112 seconds and in the cases (D) and (E) in 100 seconds.

These results seem to be counterintuitive because MCTS is supposed to work better with high depths. The explanation for this is that if the total depth is 10 and, even if the number of actions available is two (which is not), the different ways the game can be played is $2^{10} = 1024$. This number is actually higher than the number of iterations chosen in situation (B), which is why the result is bad. The cases (C), (D) and (E) are curious because it is not expected that the MCTS performs well with low depths. The interpretation for this fact is that in this game, due to its simplicity, the choice is almost binary. If an action does not lead to death then it probably leads to victory. So what happens with low depths is that the character will always be choosing to survive in a short term and it will eventually lead to winning because he avoided dying every time.

Having this into consideration, the best option should be (E) because its lowest depth and number of iterations make it the fastest while performing extremely well in terms of time.

4.2.2 Discussion about the *Playout* heuristic

Considering the facts above, I tried a heuristic that gave a reward of 0 if the character died and 1 otherwise, in the situation (C). The result was good as the character managed to win, however it took around 165 seconds to do so.

The results above were achieved using a different heuristic that is the following:

$$Heuristic = \begin{cases} 0 & \text{if } hp \leq 0 \\ 1 & \text{if } m = 25 \\ e^{\frac{-t}{80}} & \text{otherwise} \end{cases}$$

Note: the game ends when the character collects 5 chests, each with 5 MONEY, making a total amount of 25. t is the amount of time that has passed since the beginning of the game, m is the amount of money gathered and hp is the heath points of the character.

This heuristic returns a lower value if an action takes more time to do than the other. This way, the victory is achieved faster than using the one I referred before.

5 MCTS in Don't Starve Together

5.1 A Few Details About Don't Starve Together

Don't Starve Together is an open world survival video game and so its objective is to stay alive for as long as possible.

The main character has three features which are Health, Hunger and Sanity. When his Health reaches the value 0, the character dies. When his Hunger bar is empty, he starts losing Health. When his sanity is very low, he starts being chased by monsters who attack him and make him lose Health. There are also other monsters that spawn around the map that can attack him. At night, if he does not have some type of light (a torch or a fire), he is attacked by ghosts and loses his sanity very fast.

The world has objects that can be picked by the character and be used to build items which can then be used to get other objects. There are also objects that can be eaten to decrease the hunger which can be picked in the wild.

5.2 Tools Used to Interact with the Game

To implement MCTS as a decision making for the character in *Don't Starve Together*, I used FAtiMA Toolkit [5]. This is a set of tools that was designed and is being developed to create characters with social and emotional intelligence. Even though I did not use social and emotional interactions, this toolkit allowed me to implement MCTS for this game. Besides this, I used an application developed by a student for his Master's degree thesis [4]. What it does is establishing a connection between the game and FAtiMA so that information can be extracted from the game and decisions can be sent back to the character. FAtiMA Toolkit has a Knowledge Base which stores the information about the in-game objects and the character. This information is what I used to build the World Model. I will explain the World Model in more detail bellow.

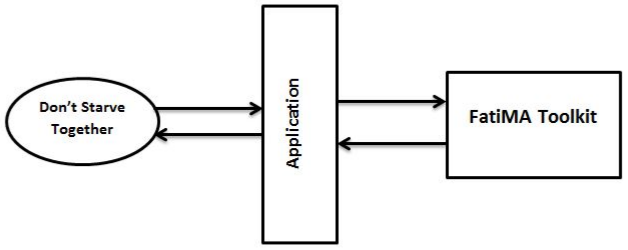

Figure 6: Scheme of the interactions

5.3 MCTS Approach Based on Game Properties

Being an open world game and having too many available actions are characteristics that make this game unfit for using a standard version of MCTS. In order to deal with the problems referred above in Subsection 3.2, I used some of the solutions described there. The ones I used are listed below:

- Simpler world model
- Limited depth in playout
- Usage of a playout heuristic
- Macro-actions
- Limited number of actions

5.3.1 Pre-World State, World Model and Limited Number of Actions

To create a simpler world model for MCTS, I used the concept of Pre-World State to store only the information I needed for the actions I wanted to use. To do so, I accessed the Knowledge Base and made a selection of the world objects I needed, and I also kept the information on the equipped items and the inventory. By choosing the items, I highly reduced the number of actions that could be done by the character.

Besides this, I also preserved the information about the character state (Health, Hunger and Sanity) and the current phase of the day (night, dusk or daytime).

Instead of creating a mechanism for each action to check if it could be done at a given point, the way I did it was by creating a set of available actions when I created the World Model. This is one of the most important features of the Pre-Word State because it keeps the information about the items and it enables the creation of a World Model with the available actions and less information about the objects.

When an action is performed, the simulation withdraws the ones that stopped being available and adds, if it is the case, the new available actions.

It is important to note that each item in the world has a GUID which is an unique identifier for it.

Another advantage of the Pre-World State is that, the way I implemented it, it stores only the GUID of the closest object of each type which is better than keeping all the information. This meant that the number of actions was again highly reduced, which was one of the solutions to make MCTS run faster. For example, if there are 50 pieces of grass in the world, instead of having 50 actions to pick grass, there is only one that is performed over the closest of them.

5.3.2 Pre-World State Abstraction

The Pre-World State is created with a Knowledge Base and has the following components:

- **Walter**: it is an object of the type Character which is a class that I created to store the information about the character (HP, Hunger, Sanity and Position)
- **Cycle**: it is a float that represents the stage of the day
- **CycleInfo**: it is an array which stores the amount of time each phase of the day lasts. The phases are day, dusk and night and they can vary according to the season
- **Entities**: it is a list which stores the information of the objects that exist in the world. I created a class called ObjectProperties to do so and its components are the name of the object I chose*, the name of

the object in the Knowledge Base, the GUID of the closest of each kind, the quantity, the Position and three boolean values that say if the object is Collectable, Pickable, Mineable or Choppable

- **Inventory**: it is a list of 3-tuples that contain the name, the GUID and the quantity of an object in the inventory
- **Equipped**: it is a list of pairs (2-tuples) that contain the name and the GUID of each equipped object
- **Fuel**: it is a list of 3-tuples that contain the name, the GUID and the quantity of the items that can be used as fuel for fires
- **Fire**: it is a list of 3-tuples that contain the name (there are two kinds of fire) and the position of the fires in the world
- **KnowledgeBase**: it is the database which has all the world information

*In some cases, several objects that are different in the world are stored as the same. For example, there are two kinds of berry bushes but they are stored in the same index of the list Entities.

5.3.3 World Model Abstraction

The World Model is created with either a Pre-World State or another World Model and has the following components:

- **Walter**: it is the same as in the Pre-World State
- **WorldObjects**: it is a list that stores in each element the name, the quantity and the position of each of the objects that exist in the World
- **Fire**: it is the same as in the Pre-World State
- **PossessedItems**: it is a list that stores in each element the name and the quantity of the items in the inventory
- **EquippedItems**: it is a list that stores in each element the name of the equipped items

- **Fuel**: it is a list that stores in each element the name and the quantity of the objects that can be used as fuel for fires
- **Cycle**: it is the same as in the Pre-World State
- **CycleInfo**: it is the same as in the Pre-World State
- **AvailableActions**: it is a list of actions that the character can do in the current state. It is created based on the Pre-World State's information and it is updated when an action is done.
- **Parent**: it is a World Model which originates the current one. The current World Model is obtained from its parent by doing an action.

5.3.4 List of Possible Actions in the Model

- **Grab** - this action can be done to grab the following items:
  - Log*
  - Rocks*
  - Twigs**
  - Flint
  - Cut Grass**
  - Carrot
  - Berries***
  - Axe****
  - Pickaxe****
  - Torch****
- **Stay** - this action can be done to stay near a fire during the night.
- **AddFuel** - this action can be done to add fuel to a fire.
- **Build** - this action can be done to build the following items:
  - Campfire
  - Fire Pit

  - Torch
  - Axe
  - Pickaxe

- **Eat** - this action can be done to eat if the character has any of the following items in the inventory:
  - Berries
  - Carrot

- **Equip** - this action can be done to equip a torch. It is not used to equip axes and pickaxes because those are integrated in the macro-actions described in the next subsection.

- **Unequip** - this action can be done to unequip a torch. It is not used in any other case because when the action equip is performed, it automatically unequips if there is an equipped item.

- **Wander** - this action can be done to move randomly around the map but close to the previous location.

* these are two of the macro-actions described in the next section.
** these can be picked up from the floor or can be picked from other existing items. Cut grass can be gathered by picking grass and twigs by picking saplings.
*** berries can be either picked up from the floor, picked from regular berry bushes and picked from juicy berry bushes. When they are picked from juicy berry bushes it corresponds to the macro-action described in the next section.
**** these are usually built.However, sometimes they exist on the map and can be picked up.

5.3.5 Macro-Actions

Macro-actions, as said before, are a useful tool to simplify the simulation of the world because they group atomic actions that do not need to be run individually by the MCTS. I used the following macro-actions:

- Grab_berries = Pick Berrybush + Pick Up Berries
- Chop_tree = Equip Axe + Chop Tree + Pick Up Log

- Mine_boulder = Equip Pickaxe + Mine Boulder + Pick Up Rocks

These macro-actions are important to use because the probability of this sequence of atomic actions occurring, if not grouped, is not very high. However, these macro-actions make sense and are significant for the survival of the character.

5.4 Discussion about the Number of Iterations and *Selection* and *Playout* Depths

With *Don't Starve Together*, the results achieved by changing depths in the *Selection* and *Playout* were very similar to the previous ones.

However, the explanation for this is slightly different. Increasing the depth always means increasing the number of iterations, otherwise the results will be poor. This means that, in order to run MCTS fast, it is better to experiment using lower depths.

As this is a survival game, the long term results do not matter as much as short term results because to survive for long periods of time, one needs to keep surviving in short periods of time. This was proven by the MCTS results in this game. In order to survive during the night, the character needs to have light (the easiest way is by building a torch) and if he fails to do so, he will die. I tested outside the game, with a Knowledge Base previously generated in-game, a situation where the character was very close to the night time and he had a torch in the inventory but it was not equipped. The mandatory action in that situation was to equip the torch, otherwise he would die.

When I tested this, with both depths equal to two, the MCTS choice was to equip the torch. However, when I changed both depths to five but maintaining the number of iterations, it did not always make the obvious choice of equipping the torch. What I concluded from this was that the action of equipping the torch was being made eventually and, since the reward is based in the last state, if he had the torch equipped at that point the reward would be similar to other situations disregarding how early the torch was equipped.

By having a smaller depth, this situation is somewhat prevented because it has less options in the timeline to do the action. Based on these facts, I opted to use a *Selection* depth of 2 and an equal *Playout* depth.

5.5 Discussion about the *Playout* heuristic

To survive in *Don't Starve Together*, what is essential is to have food to keep the Hunger bar high and to have light for the night. Taking this into account, the heuristic I chose gives the reward based only on these two factors.

The first experiments I did, measured other factors such has having an axe or a pickaxe or having more items. The results I got with those heuristics were not bad but then I tried to use a simpler heuristic with only the crucial components. This had a very positive result because the character was focusing on getting torches for the night and getting food.

The last heuristic I used and tested is more like two heuristics in one, it has a reward for the daytime and a different one for the nighttime.

During the nighttime its value is 90% for having a torch or having a fire and being close to it (LightValueNight) and the other 10% are what I called the FoodValue. The FoodValue is used both during the day and during the night but, while it is daytime, it has a more significant impact on the total value, being it 30%. The other 70% are the LightValueDay and it measures the amount of items that have been collected that can be used to have light and if the character actually possesses a light item (mainly a torch). The FoodValue has a component that measures the amount of food items possessed and another that measures the hunger.

The heuristic is the following:

$$Heuristic = \begin{cases} 0 & \text{if } hp \leq 0 \\ \frac{9LightValueNight\ +\ FoodValue}{10} & \text{if it is night} \\ \frac{7LightValueDay\ +\ 3FoodValue}{10} & \text{if it is day} \end{cases}$$

$$FoodValue = \frac{6InventoryFoodValue\ +\ 4HungerValue}{10}$$

$$InventoryFoodValue = \begin{cases} 1 & \text{if } f \geq 5 \\ \frac{f}{5} & \text{otherwise} \end{cases}$$

$$InventoryFoodValue = \begin{cases} 1 & \text{if } h \geq 100 \\ \frac{1}{(\frac{h-150}{50})^2} & \text{otherwise} \end{cases}$$

$$TorchValueNight = \begin{cases} 1 & \text{if has torch or } df < 6 \\ 0 & \text{otherwise} \end{cases}$$

$$TorchValueDay = \begin{cases} 1 & \text{if has torch and it is not equipped} \\ 0.6 & \text{if *} \\ 0.3 & \text{if **} \\ 0 & \text{otherwise} \end{cases}$$

* does not have torch and has 2 cutgrass and 2 twigs or has 2 logs and 3 cutgrass or has 2 logs and 12 rocks
** does not have torch and has cutgrass or twigs or logs or rocks

Notes:

1. building a torch requires having 2 cutgrass and 2 twigs, building a campfire requires having 2 logs and 3 cutgrass and building a firepit requires having 2 logs and 12 rocks.

2. hp is the character's health points, f is the number of food items in inventory, h is the character's hunger and df is the distance to the closest fire.

This heuristic, during daytime, values gathering items that will eventually build into some sort of light to survive the night. During the night, as the character must either have a torch equipped or be close to a fire, it values this almost exclusively. At the same time, it always values having some food and not being hunger, which is also a cause of death in this game.

The maximum survival time I had achieved before using this heuristic was 2 days. However, after implementing it, the character managed to survived another day reaching a total of 3 and was very close to surviving the night between days 3 and 4.

6 Conclusion

Reaching the end of the project, I think the goals I set in the beginning were achieved. I learned more about Artificial Intelligence and also about programming, which is a very advantageous skill nowadays. Besides, I think I learned the essential about Monte Carlo Tree Search and how it can be applied in real contexts.

This was a challenging project because I had to learn new things and I had to use tools that were developed by other people, which is something I was not used to doing.

Overall, I think the results were positive but there is always space for improving. I did not test everything I could in the MCTS but I learned it and in the future I am sure it will be useful in some way.

As I am a mathematics student, I think this provides an added value because I explored some new contents that I did not study during my degree. Also it allowed me to investigate an area that I seem to like and may be studying more deeply in the near future.

References

[1] Browne, C. et al. (2012). *A Survey of Monte Carlo Tree Search Methods*, IEEE Transactions on Computational Intelligence and AI in Games, 4(1), 49.

[2] Dias, J. and Santos, P. A. (2018). Artificial Intelligence for Games. *Lectures PowerPoint*

[3] Russel, S. J. and Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, New Jersey: Prentice Hall

[4] Almeida, F. V. (2018). *Creating an Agent-Based Framework for Don't Starve Together*, MScs in Informatics thesis, Instituto Superior Técnico - Universidade de Lisboa, Lisbon

[5] Santos, P. A., Guimarães, M., Jhala, A., Mascarenhas, S. (to appear). *Emergent social NPC interactions in the Social NPCs Skyrim mod and beyond*, Game AI Pro, Vol 4

[6] Dagerman, B. (2017). *High-Level Decision Making in Adversarial Environments using Behavior Trees and Monte Carlo Tree Search*, MScs in Computer Science thesis, KTH Royal Institute of Technology, Stockholm