Building a Planner for Minecraft[*]

Mart Kartašev, Julian Neubert, Carlo Rapisarda, Nicolas Zimmermann
KTH Royal Institute of Technology
kartasev@kth.se, juliann@kth.se, carlora@kth.se, nzim@kth.se

Abstract

In this work, we exploit the open-source framework Malmo to build an Artificial Intelligence connected to the popular computer game Minecraft. We explore planning methods with hierarchical actions to control the main character and achieve predefined goals. Our final results were encouraging, as our agent was able to successfully build a complex structure resembling a house. We believe that with additional development we could further optimize the planner to reduce the computational overhead and return more efficient sequences of actions towards the goal.

1 Introduction

Minecraft is a sandbox survival video game that revolves around gathering resources, crafting new objects, and building structures of practically any complexity, making it an excellent platform to test an intelligent agent. Each resource present in the world (for instance stone, wood, water, etc.) is represented in the form of *blocks* of equal size. As an interface to the game, we used an open source agent simulation framework named *Project Malmo* and developed by Microsoft [Johnson *et al.*, 2016]. Malmo is essentially a fully featured Artificial Intelligence research platform; it is installed into Minecraft as a *mod*, and allows us to interact with the game-world and control the character in the same way a human player could do. In addition, Malmo provides access to a live video-feed (useful to experiment with Computer Vision algorithms), as well as a stream of observations about the state of the world, such as the position of the character and the type of each block around us.

Our objective for this project was to build an intelligent agent capable of interacting with the Minecraft world, accomplishing simple to complex tasks, looking for construction material, gathering what's necessary, and building structures. To achieve this, we formulated our goal as a planning problem, and started from the broad existing research on the matter.

[*]Official website: https://minecraft.net

2 Related Work

Most of the research related to planning applied to games focuses on controlling the behaviour of non-playing characters (or NPCs), with the objective of enhancing the experience of the human player. In some simple cases, these characters follow plans that are pre-computed offline (either manually or automatically), and some rules are put in place so that the human player cannot interfere with them [Kelly *et al.*, 2008]. This technique is used to cut down the computational cost of online planning (that is, planning executed at run-time during the game), but often results in a less engaging gaming experience. By contrast, if the human player is allowed to interact and perturb the plan of a NPC, the planner needs to be able to handle these changes and generate a new sequence of actions accordingly [Menif *et al.*, 2014]. In our case, unforeseeable perturbations don't come from a human player, as we only have one playing character controlled by AI, but rather from the non-deterministic nature of the Minecraft world. Despite this difference, in order to build our planner we can apply the same techniques used for NPCs.

In modern video games, two main approaches stand out: Goal-Oriented Action Planning (GOAP) and Hierarchical Task Networks (HTN). In the first case, the developer defines the set of all possible actions, each having preconditions and effects; then, for a given initial state and goal, the planner performs a backward search with a path-finding algorithm, typically A*, generating an appropriate sequence of actions [Orkin, 2003]. The second method – Hierarchical Task Networks – seems to be a more popular choice for the latest games, as it tends to perform better in many situations [Menif *et al.*, 2014]; in this case, the developer defines a hierarchy of tasks, sub-tasks, and primitive actions, which are linked together as a network; the HTN planner will then produce a plan containing only primitive actions [Menif *et al.*, 2014]. This research was an essential part in the development of our project, as we opted to implement a Goal-Oriented planner with hierarchical tasks.

Another important aspect to consider is how to structure the information about the possible actions. A common answer to this question is the Planning Domain Definition Language (PDDL) [McDermott *et al.*, 1998] and its numerous extensions. Problems defined in this way can be directly fed to the planner, making it easy to add, remove, or redefine actions, fluents, and goals, often without the need to modify existing implementations of the main architecture. For large projects, this is obviously an enormous advantage, but for our purposes we decided not to use it, as an implementation with PDDL would have added an initial overhead incompatible with our time restrictions. However, in sight of future developments, we built our system modularly, exploiting advanced features of object-oriented programming.

3 Case study

As our case study, we decided to have the AI player interact with a simple flat world without hostile agents. In addition, we decided to force the character to stay at ground level, avoiding jumping on structures or falling into holes, thus reducing the dimensionality of the path-finding problem to two axis rather than three. These constraints allowed us to focus more on the planning aspects of the project, rather than having to deal with difficulties related to the game-play. The goal of the agent was updated throughout the development of this work; we first focused on building a wall, then we moved on to building a taller wall with multiple materials, and finally set the goal to building a structure similar to a house.

4 Approach

We started by defining the list of every possible action we would need for the highest level action of building a house (simplified as a box), dividing them into 3 categories: low, middle or high level actions. We also defined the list of precondition fluents associated with those actions and the effects they would have. We defined these in a PDDL-like format so that we could easily use a planner to solve given problems. The low level actions were predefined by the APIs of the Malmo project (such as *"move left"* or *"craft"*), so we started by implementing a set of middle level actions.

To get started, we developed a basic planner without optimizations. We kept improving it as our agent became more complex to accommodate all of the tasks. The planner has knowledge of all the possible actions and the correlations between them; in addition, the system stores a list of the precondition fluents required to accomplish a given result, together with the actions that must be performed to satisfy the preconditions.

We divided the development of both the actions and the planner into incremental steps sorted by complexity, adding new capabilities as time went on and as different sub-goals were reached. Our main objective was to obtain a working planner as fast as possible, so that we could immediately experiment with the actions and develop the project in a controlled manner.

4.1 Observations from the world

This first fundamental part consists on retrieving information about the world state; this comes in the form of a list representing the set of blocks around the character, but some preprocessing is required to extract the relevant data. To solve this problem, we used Google's Gson library to simplify the interpretation of JSON observation strings and automatically convert them into well defined Java objects. This step allows us to take advantage of static type-checking for faster debugging.

4.2 Simple movement

We then implemented a simple action to reach any accessible point in space using the low level actions provided by Malmo's APIs. Originally we used a very simple movement algorithm, but as we saw that this was not enough a more complex search-based movement was implemented. We find a path around obstacles by using a Breadth First Search algorithm, although as discussed earlier (section 3) this was constrained to a 2D plane for practical purposes.

4.3 Looking around

Since to interact with blocks we need to point the camera at them, another important action is the one that allows us to control the *pitch* and the *yaw*. The development of this idea was not a simple task, mostly due to the non-standard reference system used in Minecraft, as this action requires trigonometric calculations. Because many other actions entirely depend on looking at the correct block, special care was taken to ensure that this action worked properly.

4.4 Selecting items

Selecting items is also an important task in the Minecraft world. To achieve this, we had to implement correlation mappings to select the correct tool for gathering resources and selecting the block we want to place. The improvements to the planner were mainly about dynamics as we focused on building a versatile structure that could independently evaluate a plan as we saw it.

4.5 Gathering blocks

The action of gathering blocks is essential to build items of any kind; to solve this problem, we locate the interesting block taking advantage of the observations, then we move next to it, look at the correct location and gather the material, making sure that we have an unobstructed line of sight. This last requirement is fulfilled by simply imposing that the character is at exactly 1 unit of *Manhattan distance* from the target block when gathering, so that if the block is reachable line of sight is guaranteed to be free.

4.6 Placing blocks

This action is used when the agent wants to place a block in the world to build a structure; this is more complex than it seems, because when forming structures the agent needs to be aware of the position of the existing blocks and their spacial relationships. Again, this action was implemented by imposing a required distance from the target position, although placing blocks above head level required coordination with the jump action through synchronized delays.

4.7 Crafting items

Sometimes we want to place a block different from the one we gathered, or we want to create new tools. This can be implemented using crafting, essentially combining items from the inventory together to obtain new items.

4.8 Planner optimization

The optimization can be further divided into these sub-tasks:

- Dealing with failed states
- Plan memorization
- Implementing cost to evaluate preferable actions
- Reduction of actions to explore more efficient and sorting based on cost. Essentially a loose constraint approach

Some of these ideas on optimization were implemented in our planner, while some others were not fully explored during the development of this project due to the time restrictions, see section 8 for more details.

5 The Planner

The planner is structurally divided into two primary components based on functionality. Plan determination and execution. Both of these make use of another important concept which we call the evaluation of fluents. When evaluating a set of fluents used as preconditions we usually call that evaluation of preconditions.

The planner is always initiated with a goal state - a set of fluents that need to be true in order for the planner to consider the plan completed. The goal can either be a single fluent or a conjunction of a variety of fluents. No special fluents were defined other than methods to construct lists of fluents corresponding to more complex concepts. It might also be worth noting that our high level actions are only conceptual in the same way. They exist only in the context of combining a set of fluents into a larger one, so that a list of middle level actions results.

Once the goal state has been input into the planner, the first step is to determine the initial plan. Semantically speaking a plan, in our approach, is an ordered list of actions. When the plan has been created, executing it is a matter of performing the actions in the order they are given. Figure 1 shows the primary execution loop.

For each action, there are a few considerations when it comes to their execution. For instance, it could happen that by the time we try to execute an action in the plan, the effects it tries to achieve have already been completed by something else that has happened in the game world. In that case we want to skip that action entirely as performing it is no longer necessary. It might also be possible that, during the execution, something causes the preconditions of an action in the plan to become false, in which case we need to re-evaluate the preconditions and amend our plan with additional actions before performing that one.

It might also happen that during the execution of any given action we end up in a failure state. This is when some of the preconditions for that action are no longer true after execution has started, but the effect fluents are not true either. In such cases we know that we have done something that has changed the state of the world, but we have not achieved the desired change in our effect fluents. When such a condition happens we must again re-evaluate the plan based on the set of effect fluents that failed. The ones that succeeded do not need to be re-evaluated.

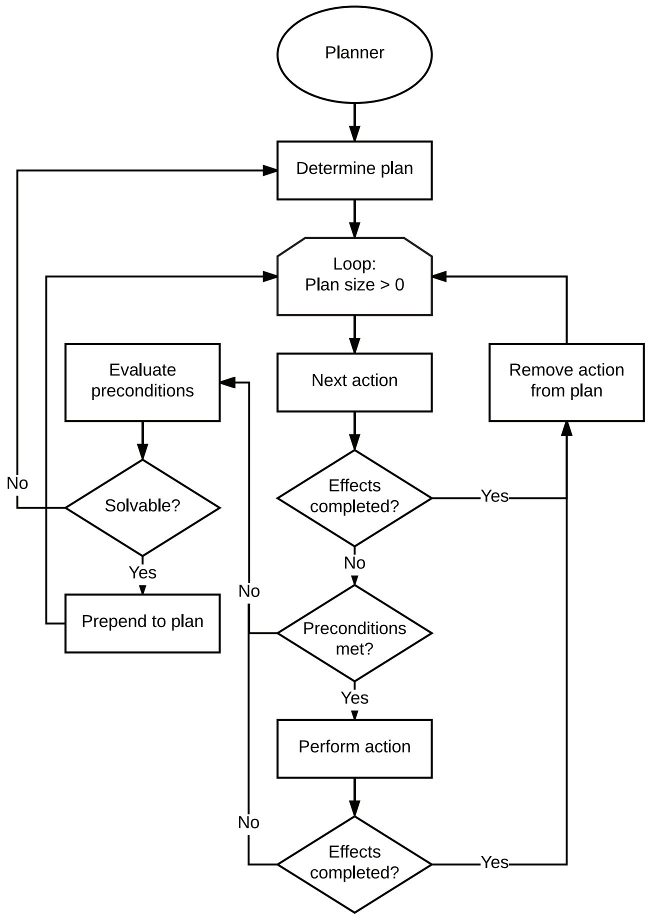

Figure 1: Overview of the plan execution logic

The evaluation of fluents is described by figure 4. The recursion is centered around the effects and preconditions we have defined for each action. For every goal fluent we check for a list of possible actions that have that fluent as an effect. We choose the cheapest action and then do a similar check for each of the preconditions for that action, again checking for all the possible actions that provide the precondition fluent as an effect. We do this recursively until we find actions that either have all their preconditions fulfilled or have no preconditions. Note that in order to execute any given plan we need at least one action that has no preconditions, otherwise the planner will never be able to execute any plan.

Determining the plan consists of a series of recursive precondition evaluations. The general concept of determining the plan is described in figure 3. We evaluate the actions needed for each incomplete goal fluent separately and create our plan based on that. In addition, we experimented with creating a loosely constrained problem by looking only at some of the fluents or actions for the initial plan and then sorting the actions based on cost.

It might happen that due to the state of the planner, we have created a situation for ourselves where we can no longer solve the problem. In practice, this means that we receive an empty list of possible actions when evaluating a fluent. Instead of giving up right away, it might be prudent to try to plan everything again, creating a completely new plan instead of amending the old one. Sometimes changing the plan multiple times creates situations which the planner can not reason itself out of. This additional check helps catch some of the false negatives and stops the agent from giving up too early.

There were a few ideas that we had for the planner, but didn't have time to implement. The first would be to incorporate action or fluent based filtering into the original planning mechanism, instead of doing the filtering after the plan is generated. This would not only help optimize the plan during execution but also effectively improve the time it takes to generate it.

In addition, each action is currently chosen based on a greedy search through all of the possible actions corresponding to a given fluent. As the cost of actions is based on preconditions there are two possible approaches to improve this: either improve the cost functions so that they better represent the cost of each action and their preconditions, or implement a search algorithm that evaluates each action and the possible tasks that are required to permit those actions. This way we create simpler cost functions for the actions, but instead have a more advanced search mechanism. Out of the two, the latter seems to be a better approach, as it would not only ease the creation of new actions, but possibly also reduce the time complexity of the planner.

Another shortcoming of the whole construct at this time is that the agent is not able to respond to threats. For example, in a more complex scenario where there might be hostile non-playing characters in the world, we would need some type of mechanism in order to dynamically interrupt our plans and react to such threats. In the current context, the planner would have a hard time adjusting to such occurrences and would likely simply fail to react appropriately. Handling these cases would require some redesigns, but it is something we considered exploring in the future (see section 8).

6 Results

By the end of this project, we managed to achieve most of the objectives that we were aiming for. In particular, among these are the following capabilities:

- Movement
- Inventory management
- Interaction with the world
- Resource gathering
- Use of blocks for basic building and crafting
- Building a simple wall
- Building a tall wall with multiple materials
- Building a simplified house

During the development, we encountered numerous difficulties that we were unable to foresee from the beginning of this work (such as moving in the third dimension, testing for line of sight and path-finding issues), and this meant that we couldn't dedicate much time to other challenges like the optimization of the planner. For instance, the computation of the very first plan is particularly expensive as expected (since the final goal is further away), but the actual time required to perform this operation went beyond our predictions; in addition, the resulting plan is often not the most efficient and might include actions that turn out to be unnecessary. Despite this, given the overall effort of the team and the time constraints, we are very satisfied with the obtained results (see figure 2).

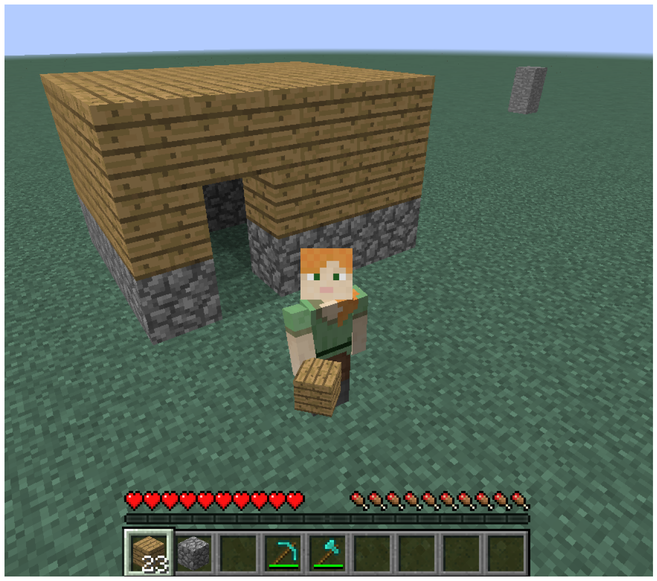

Figure 2: Complex structure built by our agent

7 Conclusion

During the development, we encountered some unexpected problems unrelated to planning, which unfortunately impacted our progress.

In particular, some actions were difficult to put in place, requiring complex space geometry (for instance calculating if the *line of sight* from the character to a block is free, or computing the location of the spot to target to perform a certain action on it); the scale of these challenges was substantially increased by the fact that the Minecraft world is a 3D world.

Implementing the planner as a mix of various fluents and actions was difficult and took the largest portion of our time, although the advantages of using a modular structure quickly proved to greatly simplify the task of planning. Once we had all the definitions and ideas implemented as we wanted them, playing around with different preconditions, effects and action mappings provided some very interesting insights into AI planning. The way the program ended up working seems rather natural and it's easy to see how it can be expanded to do more. Having the planner infrastructure in place actually makes further development seem rather simple (more on future work in section 8).

In conclusion, we found this project more challenging than expected, but the difficulties only pushed us to learn more about Artificial Intelligence. We saw a lot of parallels to other very interesting research topics, in particular AI in video games and robotics. We were fascinated by the research papers we read on these subjects and wished we had more time to implement our ideas. Every team member had at least one topic in the domain of AI which he will further look into in the future, and regardless of this, we plan on continuing the development of this project in our spare time.

8 Future Work

As discussed in the previous sections, the initial objectives of this work were achieved during its development. However, there are still many areas in which the system could be improved, even without requiring additional research.

First of all, the planner would benefit from the implementation of several optimization techniques, such as caching of previous results, limiting the length of each plan, or pruning based on context, in order to cut out branches of the search space that would lead to unwanted states [Orkin, 2003]; these changes would result into a faster planner. Another step towards the same direction would be the implementation of a more complex search algorithm such as A*, similarly to what is done in state-of-the-art planners for NPCs in video games (as discussed in section 2).

Using quicker algorithms and techniques would not only mean less time required for computations, in fact having a faster planner would also enable the system to react to sudden changes appropriately, simply by re-planning from the current state [Orkin, 2005]. For example, with such an implementation we could remove some of the limitations imposed in our case study (section 3), having the character move on harsh terrain or even introduce hostile agents into the simulated world (both factors that lead to non-deterministic scenarios).

We also discussed about the possibility of exploring fairly different techniques than the one we used, such as Behavior Trees (BT); this technique in particular is well researched and employed in several areas, from video games with advanced AI logic to control systems and robotics [Colledanchise and Ögren, 2017], and we think it could work well for our purposes. We have yet to look deeper into this complex subject, but based on a preliminary analysis, implementing such a model would require mostly additions to our existing code, making it worth exploring BTs for future developments.

Acknowledgements

This work was developed as a final project for the Artificial Intelligence course (DD2380) held by professors Jana Tumova and Patric Jensfelt at KTH Royal Institute of Technology; we want to thank them for giving us the opportunity to explore this subject and for providing the necessary support.

References

[Colledanchise and Ögren, 2017] Michele Colledanchise and Petter Ögren. Behavior trees in robotics and ai: An introduction. 2017.

[Johnson *et al.*, 2016] Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. *The Malmo Platform for Artificial Intelligence Experimentation*, page 4246. AAAI Press, Palo Alto, California USA, 2016.

[Kelly *et al.*, 2008] John-Paul Kelly, Adi Botea, and Sven Koenig. Offline planning with hierarchical task networks in video games. 2008.

[McDermott *et al.*, 1998] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. *PDDL – the Planning Domain Definition Language*. Yale Center for Computational Vision and Control, 1998.

[Menif *et al.*, 2014] Alexandre Menif, Éric Jacopin, and Tristan Cazenave. *SHPE: HTN Planning for Video Games*, pages 119–132. Springer International Publishing, Cham, 2014.

[Orkin, 2003] Jeff Orkin. Applying goal-oriented action planning to games. 2003.

[Orkin, 2005] Jeff Orkin. Agent architecture considerations for real-time planning in games. 2005.

Appendix

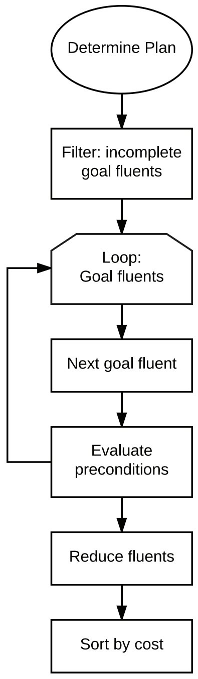

Figure 3: Procedure used to generate a plan

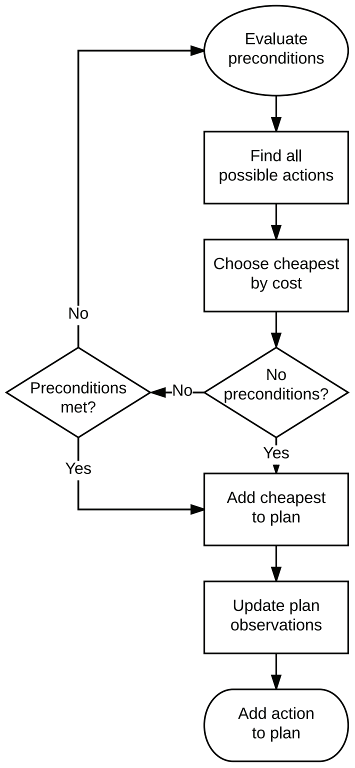

Figure 4: Procedure used to evaluate the preconditions