Modern Game AI Algorithms
PCG and Kiting in 0 A.D.

Berend van Starkenburg
s1800604

Ernest Vanmosuinck
s3210359

Floyd Remmerswaal
s1521616

Raashid Khan
S2705745

Sava Sharif
s2685612

May 4, 2023

Abstract

Game AI is a rapidly evolving field, and with the ever expanding availability of algorithms, machine learning, and deep learning techniques we can create AI agents capable of mimicking human player behavior in an environment that is generated procedurally. In this project, we aimed to implement agents capable of kiting in the real-time strategy game 0 A.D. We found that agent was able to learn how to kite, but found that the same agent struggles when faced with a procedurally generated maze environment. We also implement the procedural generation of cities.

1 Introduction

This report aims to explain how in this project we have implemented an AI agent and procedurally generate a city for a game. 0 A.D. is a free open-source Real Time Strategy (RTS) game in a historical setting where players build their own base while trying to destroy the enemy base. The game stands out for its wide range of structures, units and terrain with a focus on historical accuracy.

In 0 A.D. players have to manage their economy and create armies. These armies consist of many different types of units, but in our project we focus on two units: the ranged cavalry unit and the spearmen. In 0 A.D. (and many similar games), there is a rock-paper-scissors balance between cavalry, 'regular' infantry, and spearmen. Spearmen deal very high damage to cavalry units, and are thus a big threat. For the cavalry archer to beat the spearmen, it has to continually damage the them while staying out of their range.

We employed Reinforcement Learning methods to teach the agent to kite. *Kiting* is a way of fighting where a (mobile) ranged unit fights a melee unit by attacking it and moving backwards, always staying out of range. As one unit has to be attacking from a distance and stay clear of a unit that can perform melee attacks, we decided to set our agent to a Cavalry unit which has a higher movement speed, fighting against an Infantry unit. Figure 1 shows kiting in action.

Additionally, we explored the use of Procedural Content Generation for the creation of world maps. We take two approaches, maze generation and city generation. The former is the generation of procedurally generated mazes of varying sizes. The second and more challenging aspect is the creation of procedurally generated cities with a focus on believability. These procedurally generated maps are applied to the roll-out of the agent to evaluate how well they perform in an unknown environment.

The repository containing our implementation and installation instructions can be found on https://github.com/floydremmerswaal/MGAI_0ad.

(a) The ranged cavalry unit attacks the spearmen.

(b) The spearmen get close to the cavalry archer.

(c) The cavalry archer retreats.

Figure 1: Kiting done by one cavalry archer. You can see that, when the enemy gets too close, the cavalry archer retreats after which he will attack again.

2 Reinforcement Learning

The first task of our research was to use machine learning to improve the tactical behavior of units in the game 0 A.D. As stated before, with the help of reinforcement learning, an agent will learn how to control a group of cavalry archer units. Apart from these units, the environment consists of a map where the game is played on, and a group of enemy infantry (spearmen) units. The goal of the task is to learn the agent how to kite the enemy units. Since the Pyrogenesis engine of 0 A.D. has a dedicated interface for reinforcement learning, a crude implementation for this problem along with a suite of helpful functions already exists [1]. With the help of the Python Ray library[4], it is easy to switch algorithms to solve the environment.

For our experiments, the following algorithms will be used: A3C, DQN, and PPO. The state space of the baseline implementation was designed as follows: the state space is continuous and is contained in a closed box in the euclidean space with the range $[0, 1]$. Each state is a coordinate of the map the game is played on, normalized to fit in the range of the box. The states are the normalized average distance between the cavalry archers and the infantry units.

The action space is discrete and consists of two actions. *Retreat*: the center point of the agent's units and the enemy units is calculated. The group of cavalry archers moves in such a way that the distance between the two centers is maximized in the next game step. *Attack*: the enemy closest to the center of the cavalry archers is calculated. Every cavalry archer attacks this enemy. The reward function behaves as follows: if all enemy units are killed, a reward of $+1$ is given. If all allied units die, a reward of -1 is given. While this implementation allows us to study a bit of the behavior of our units and the enemy units, from a first glance it is evident this model will take a very long time to converge, due to the lazy reward function, and the simple state and action spaces. Therefore, to actually learn efficient kiting behavior, the main focus of our implementation will be on improving the reward engineering, the state space, and the action space.

2.1 Reward engineering

The current model will take a very long time to converge, since only giving a reward at the end of each episode will make it difficult to end up with a competent policy. The action traces are too long per episode, so the agent will have a hard time learning what actions are good and which are not. With

this in mind, the following reward engineering strategies were implemented to improve the performance of our model.

AvoidantReward Whilst observing our model play the game, we noticed that the agent plays the game too risky. Since the probability of taking a retreat action and an attack action are the same at the beginning of training, the model will not take enough retreat actions to survive long enough to learn how to outrun the enemy units. Our first reward engineering strategy, therefore, aims to learn the agent how to stay alive longer. This is done by allotting the rewards as follows: -30 if all allied units die, $+30$ if all enemy units die, and $-.1$ reward for each percentage of missing allied units health compared to the previous game step, and -1 for each allied unit that dies.

DefensiveReward Eventually, the agent also needs to learn that there needs to be a priority on killing enemies one by one instead of damaging the whole group. The threat of a bigger group snowballs, and this needs to be mediated by focus firing enemies. Therefore, the second strategy also awards a reward of 1 if an enemy is killed.

HealthDeathReward The final iteration of the reward functions also awards a reward of .1 for each percentage of missing health of the enemy units compared to the previous game step.

2.2 State and action space

The current state and action space only uses the relative distance between the center of the cavalry archer group and the center of the spearmen group. This leads to two problems. Firstly, the group of cavalry archers always moves as one group, and always attacks as one group. It would be more beneficial if the allied units were able to learn how to split up, since the spearmen would have a harder time keeping up if the pack of cavalry archers is less consistent. Also, moving as a group to one center is inefficient. When the first unit arrives at the center, the other units will still try to reach that position, even as it is blocked by a unit, wasting precious game ticks. Attacking one enemy as one group could be efficient for focus firing, but an individually attacking agent should be able to learn this as well. Sometimes, it is also more beneficial to focus individual units, if the most damaged unit is too hard to reach. The second problem, is that the agent assumes the map is infinitely large. Since it only knows the relative positions, when it encounters the map border, it needs to learn how to follow the map border. In addition to using relative positions, using absolute positions would prevent this problem. In the improved state and action space, the relative distance between each individual cavalry archer and each individual spearman is used. Also, three new actions are added to the action space: `AttackClosest`: each cavalry archer attacks the unit closest to its position, `MoveClockwise`, and `MoveAntiClockwise` instead of moving in the opposite direction of the enemy units, the units walk to the left, or to the right of the group of spearmen.

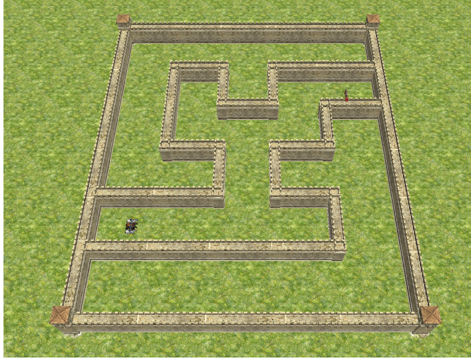
Experimenting on the agent in 0 A.D. with varying level of difficulties is performed by providing an environment that is unknown to it. This enables us to effectively test the agent's training. Two high level difficulty maps were chosen for this purpose of which its generation is explained next section.

3 Procedural Content Generation

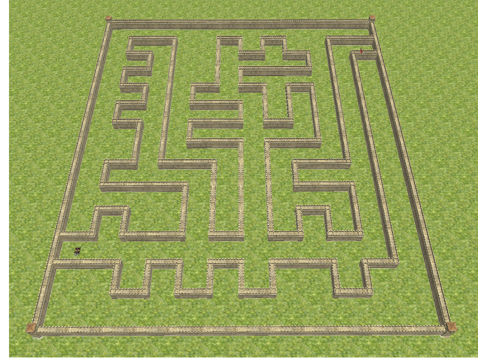
Two approaches have been taken for procedural map generation. We have implemented a maze in order to experiment with using the reinforcement learning agent and the second part consists of creating a believable city. In the game the 'map' that is played on is called a *scenario*. These scenarios consist of two parts: an XML file that specifies where buildings and units are placed, and a PMP file. The PMP file is a binary file that specifies the terrain elevation and textures, and is hard to meaningfully edit because of the binary nature of the file. Scenarios can be manually made using the Atlas world editor that is bundled with 0 A.D. and consist of two files. These two files will be generated using Python scripts and represent the generated cities.

3.1 Maze Generation

As a baseline for simple map generation, we used procedural techniques to generate mazes in which the agents would learn to move and live, restricting their movements to move around the maze to avoid the enemies and defeat them. We employed Prim's Algorithm [5] to generate a 2D representation of the maze and its structure using 0 A.D. entities by placing walls on the path. The maze's dimensions can be changed or randomized to render the agents adaption task more complex. A representation of two mazes of different dimensions can be seen in figures 2a and 2b.



(a) 5×6 maze



(b) 12×14 maze

Figure 2: Different mazes dimensions.

3.2 City Generation

To create a believable city we used a procedural modeling method that constructs a circular city within 0 A.D. The method is based on the approach of [2] and consists of defining city bounds, district generation based on Voronoi diagrams[3] and populating the districts using the architectural style of the civilization chosen.

City Bounds City bounds are generated using the current map size. 0 A.D. allows the creation of a custom map using the Atlas scenario editor. The editor supports map sizes up to 2048×2048 units. First a circle is computed slightly smaller than the current map size. Then we define a smaller circle within. The smaller circle will be the heavily populated city core. Districts are generated within the bounds of the circles and can be tuned by specifying the number of districts and the amount of splits of circle. The splits were originally intended to create sufficient space between districts so that highways can be built that lead in and out of the city, but can be used as a way to get a more uniform distribution of district centers.

District Generation Using the random district center coordinates, a Voronoi diagram is constructed. The Voronoi diagram partitions the city into several regions that will be city districts. For each center coordinate a region is created and consists of all coordinates in the circle whose euclidean distance is less than or equal to other district centers. The regions are in the shape of a polygon. Voronoi regions can have ridges that are infinite or go out of bounds. This initially caused walls to go beyond the map. To solve this problem, we constructed a Voronoi diagram bounded by a rectangle. This way, there are no infinite ridges. Ridges that still go beyond the circle bounds, are clamped to the circle and reconstructed. Finally, walls are built for only distinct ridges and gates are placed. A generated Voronoi diagram is shown in Figure 3 with outer circle parameters $N_{dist} = 10$, $N_{split} = 3$ and inner circle parameters $N_{dist} = 1$, $N_{split} = 20$. Note that we did not add noise to the polygons to achieve a more realistic look since walls in 0 A.D. do not attach to each other and are straight, which wouldn't look seamless.

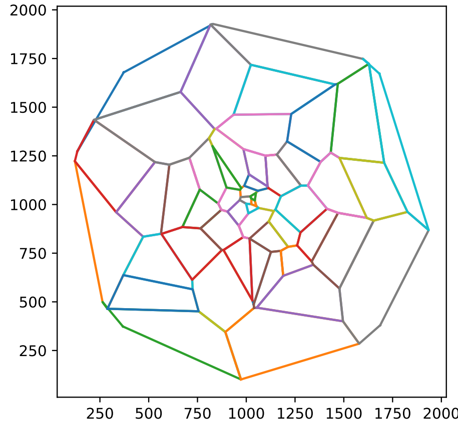
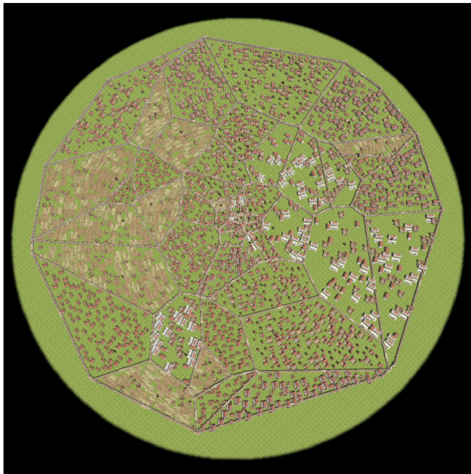
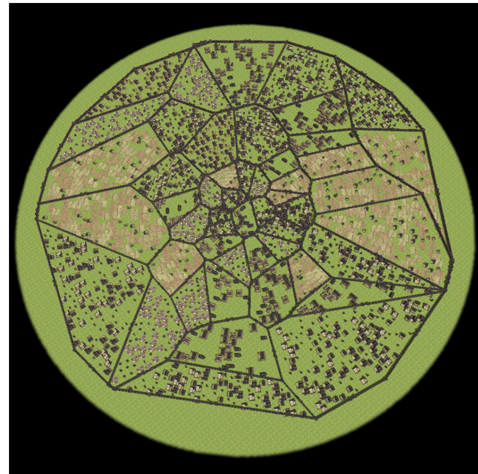


Figure 3: Bounded Voronoi diagram.

District Population All districts are populated with structures. To achieve a more believable city we first categorized different structures within 0 A.D. We have the following district types: military, agriculture, livestock, civil and housing. Each category consists of structures with their respective weights. For example, within the agriculture category, we prefer placing fields over placing forges as more than a handful of forges wouldn't be realistic. A district type is also weighted. It is more realistic to have agriculture in the outer circle, than in the city core. To prevent structures clipping into walls, we scale the polygons and we also define a distance between structures within the polygon. Structures are facing the center which gives a less chaotic look. Figure 4 shows the final result.



(a) Roman architecture.



(b) Mauri architecture.

Figure 4: Generated cities.

4 Experiments

To test how effective our implementation is to learn an agent how to kite, the following experiments will be carried out.

4.1 Improved versus baseline

First, to see if the improved state space, action space, and reward functions were effective, the improved model will be compared with the baseline implementation. For the evaluation criteria, we use the average reward per episode, and the average episode length. Because the baseline implementation

uses a different reward function, it might seem odd to compare the average reward per episode. However, since only rewards of $+1$ and -1, are awarded, we can normalize the reward of the improved model to fit in the range $[-1, 1]$. Since the largest rewards are awarded if the game is won or lost, the closer the average rewards lies to -1, the more games are lost, and the closer the average rewards lies to $+1$, the more games are won. The average time per episode criterion is helpful, since it not only shows the survivability of the agent, but when the agent has learned how to kite, it is also an indicator of how fast the agent can defeat the enemy units.

The models were run with the following parameter settings:

	γ	α	batch size	hidden layers	hidden units	hidden activation	output activation
DQN baseline	.99	.0005	32	2	[256, 256]	tanh	ReLU
DQN improved	.99	.0005	32	2	[256, 256]	tanh	ReLU

4.2 Algorithm comparison

Since the Python ray library allows for easy switching between algorithms for models, the second experiments were performed to see which algorithm works best for the agent to learn how to kite. The improved reward function, state space, and action space were used. The algorithms tested were: *PPO*, *DQN*, and *A3C*. The same evaluation criteria were used. The models were run with the following parameter settings:

	γ	α	batch size	layers_a	units_a	layers_c	units_c	actor activations	critic activations
DQN	.99	.0005	32	2	[256, 256]	None	None	[tanh, ReLU]	None
A3C	.99	.0001	200	2	[256, 256]	2	[256, 256]	[tanh, ReLU]	[None, ReLU]
PPO	.99	.00005	4000	2	[256, 256]	2	[256, 256]	[tanh, ReLU]	[None, ReLU]

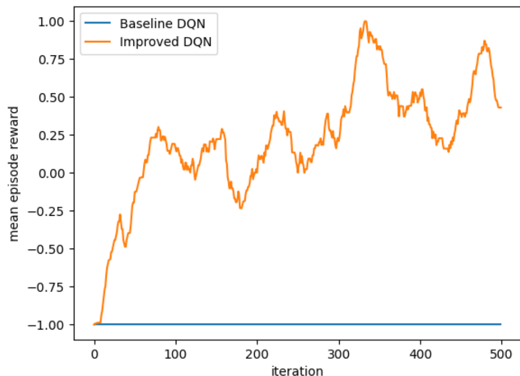
4.3 Testing adaptability

Finally, when the model that is best in learning kiting behavior is known, we want to find out how well it can produce a policy for maps other than the default map. The maze maps as discussed in Section 3 will be used for this experiment. The best method to test the adaptability of a model is to hotswap maps for each episode. This way, the agent does not learn how to rely to heavily on the environment of a single map. Unfortunately, we were not able to figure out how to do this, so a single procedurally generated maze is used as the environment. During the testing, we found out the *A3C* algorithm paired with the improved model learns the best kiting behavior. This model is therefore used for this experiment.

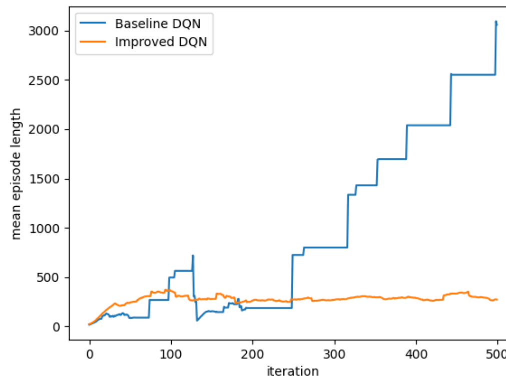
5 Results

This sections shows the results of the various experiments we have performed.

Baseline versus improved:

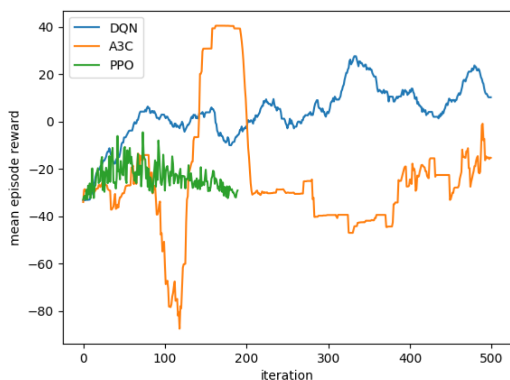


(a) The mean reward per iteration for the improved model and the baseline implementation.

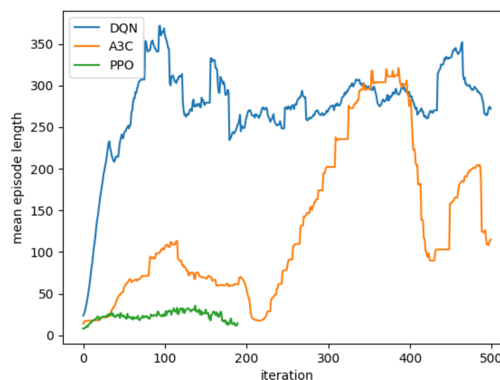


(b) The mean episode length per iteration for the improved model and the baseline implementation

Algorithm comparison:

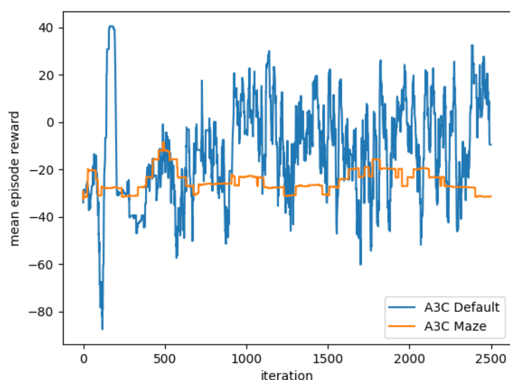


(a) The mean reward per iteration for the DQN model, the A3C model, and the PPO model.

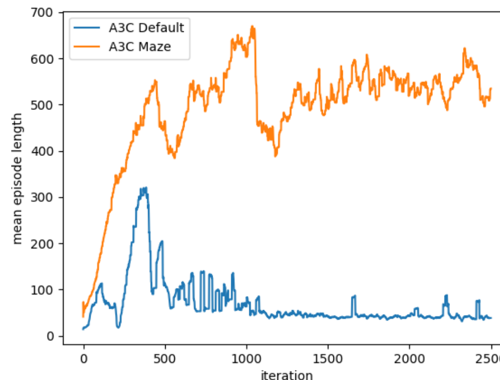


(b) The mean episode length per iteration for the DQN model, the A3C model, and the PPO model.

Testing adaptability:



(a) The mean reward per iteration for the A3C model on maze maps.



(b) The mean episode length per iteration for the A3C model on maze maps.

6 Discussion

Figure 5a shows the reward of both the baseline, and our final improved version. Both are running DQN. The baseline implementation only rewards +1 or -1 on defeat and victory respectively, so we can see that the baseline agent did not manage to learn the environment in 500 iterations. The improved agent uses a better reward function, which better displays the progress the agent has made. This improved agent manages to win the cavalry archer vs spearmen scenario. An interesting insight can be found in Figure 5b. We see that the length of an episode gets quite high for the baseline implementation. This suggests that the agent is trying to stay alive, rather than win, as the improved version has episode of roughly constant length, in which it does manage to win.

Figures 6a and 6b show an episode reward and length comparison between the three different algorithms we ran. The PPO results stop earlier than the other algorithms, as the computation time needed for PPO is very high. These $\tilde{2}00$ iterations took about 12 hours to run. In this comparison, we see that DQN is relatively stable, whereas A3C is a less stable, and PPO doesn't seem to improve that much.

Finally, Figures 7a and 7b show the episode reward and length of the A3C agent, trained on the default empty and the maze map. We see that the agent has trouble learning this maze environment, as the reward stays negative. Some progress is made, but the empty map is, understandably, much easier to learn. Also, the average length of the episodes is considerably higher in the maze scenario which makes sense as well, the agent has more obstacles to work around.

7 Conclusions

In this paper we have shown that existing reinforcement learning agents for the game 0 A.D. can be significantly improved upon by tinkering with the reward function, the state space representation, and the possible actions the agent can perform.Our results show that our agent learns how to kite in an empty map, but the same agent is also capable of learning this in a maze environment. We also showed that it is possible to procedurally generate cities within 0 A.D. The city generation is based on Voronoi diagrams and has shown to a relatively simple and intuitive way to create complex historic cities without needing extensive knowledge about urban planning. Given more possibilities within 0 A.D. one could also manipulate terrain achieving more realistic cities.

8 Future work

The Reinforcement Learning agent could be further improved upon, by more extensive tuning of the reward function. Another approach might expand the action space of the agent to allow the agent to learn a more complicated strategy. Also, it might be possible to find state space representations that perform better.

We planned on merging the PCG and RL more intimately, by letting the agent train on continually generated maps. Technical limitations prevented this from working, but future endeavors might find a way to get this to work. This might allow an agent to be trained that is able to generalize better.

For procedural content generation we have implemented two different maps based on chosen level of difficulties for the AI agent. The road ahead is to increase the adaptability of the generated structures which can be achieved by adding a variety of terrain features. For example - rivers, elevated surfaces, flora and fauna. The map generation algorithm thus will take in account the feasibility of generating content in such situations. Another improvement in the aesthetics of the maps can be made in generating districts by utilizing the district placement algorithm as proposed in the referenced paper [2] by considering the suitability of certain structures on different areas of map such as distance to highways, rivers and neighbouring districts.

Structures are placed randomly where they can fit, but in the future we want to create a street network within a district. This could be done by using the same Voronoi approach within the districts, but by using the Manhattan distance instead of the Euclidean distance resulting in more square regions.

References

[1] B. Broll. Zero_ad_rl., Jan 2021.

[2] S. Groenewegen, R. M. Smelik, K. J. de Kraker, and R. Bidarra. Procedural city layout generation based on urban land use models. In *Eurographics*, 2009.

[3] R. Klein and A. Lingas. A linear-time randomized algorithm for the bounded voronoi diagram of a simple polygon. In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, SCG '93, page 124–132, New York, NY, USA, 1993. Association for Computing Machinery.

[4] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging AI applications. *CoRR*, abs/1712.05889, 2017.

[5] F. Ramadhian. Implementation of prim's and kruskal's algorithms' on maze generation. 12 2013.