University of Bucharest
Faculty of Mathematics and Computer Science
Computer Science Department

Master Thesis

Real time map generation using Deep Reinforcement Learning

by

DOSPRA CRISTIAN-VASILE

Supervisor: Lect. Dr. Păduraru Ciprian Ionuț

Bucharest, September 2021

Contents

List of Figures

List of Tables

Abstract

Our work focuses on finding a fast, dynamic, resource light approach for generating road networks which assemble as much as possible real road networks from different cities around the globe. The continuously growing world population which forces new cities to be built, the gaming industry which becomes more and more complex and dynamic, spreading to devices like VR headsets, the community which demands a complex environment and also new industries like self-driving cars which require a training environment for their systems, determined us to come with an approach that may resolve a part of the challenges mentioned above.

The thesis analyses a couple of approaches in procedural content generation regarding roads and maps, asses their pros and cons, lastly to come with our own generative approach after which we preset our own approaches.

Chapter 1

Introduction

1.1 Motivation

In the recent years, there could be observed a drastically advancement in the following three areas: game development, self-driving cars and urbanization.

For the game development area, along with the more powerful systems that were released (CPUs, GPUs, etc.), the demands of the gaming community for the upcoming games have also increased: people require better graphics, lots of content, catchy gameplay, etc.

For the self-driving cars area, more and more producers brought their prototypes on the market, competing with each-other regarding whose car is the best in terms of consumption, power, safety, price and of course, the precision of the autonomous system.

For the urbanization area, it's well known that the population of the Earth is continuously increasing, year by year, the need of new housing places increased as well, along with the desire of people to be closer and closer to urban areas rather than rural ones. Due to that, the current cities are suffering an intensive extension and even new cities have to be built from scratch (see the new city Qatar is building for the 2022 Football World Cup)

But what do the aforementioned areas have in common? Both gaming industry, self-driving industry and city building industry face the need of **maps**.

For the gaming industry, larger and more complex maps are demanded every year in order to satisfy the needs of the gaming community. But besides this, the game must still have a decent size, must run smoothly, must be entertaining and also must be released on the market in a not very long time. This way, manually creating game maps becomes an extremely time-consuming task. As a solution, procedural map generation techniques are used to generate the content dynamically as the player advances but in the same time giving him the original feel of

a pre-defined map. This strategy was already implemented in some newer games (see "No Man's Sky") but soon it will become an usual approach for developing games.

For the self-driving cars industry, the need to develop a safe and realistic autonomous system that replaces the human driver and performs at least as better as him represents the biggest challenge in developing such cars. Considering the car needs to experiment dozens of scenarios so it gets able to adapt in almost every real-life situation, exposing the car to such a number of scenarios would be extremely time consuming if done in real life but also it would be very dangerous because the car would still be "inexperienced" and it is prone to malfunctioning. To solve this inconvenience, lots of simulated scenarios (maps with different shapes and environment) are fed to the car system in order for it to learn. Generating that many scenarios manually is not very practical so an automated system that solves this task is required.

For the urbanization industry, projecting the plan of the new city part or even the whole city it's the most important step before actual building. These plans should consider a lot of parameters like proper road network in order to optimize the traveling speed and prevent traffic jams, the key buildings position like hospitals, super markets, schools based on the population density of each area, etc. Developing such scheme without the use of any automatic generator or validator for one's vision may be far from the optimum structure the city should have.

Our goal in this paper is to present the rough idea and a sample implementation for a software capable of solving tasks as the ones mentioned above. We will try to make the software adaptive, improvable so it can be adjusted to be used in different kinds of tasks.

1.2 Approach

Now that we set our goal, it's time to analyze how to implement such system. The approach I will go by makes use of the versatility of Artificial Intelligence (AI). More precisely, after using computer vision to model the raw dataset into some useful format. After that we will use a Generative Adversarial Network (GAN) trained with images of aerial map views from different cities, depicting the road system. Although, a GAN would only generate random images depicting roads. What we actually want to achieve is dynamically generating a roads system. For this, we will make use of Lindenmayer Systems. We will extract relevant feature from our processed dataset in order to create a randomized Lindenmayer System. Along with this approach we will try to train a Reinforcement Learning agent to do the same thing while

finally we may try to combine the two algorithms into a Generative Adversarial approach but instead of GANs we plan to make use of something more powerful: Generative Adversarial Imitation Learning.

1.3 Related Work

Dynamic map generation is not quite a new approach, being used especially in video games for a couple of years now. A lot of content from the recent top video games make use of what is called 'Procedural Content Generation' (PCG) to enhance their gameplay and make the game more interesting and enjoyable for the players. We are going to analyze some methods used so far in content generation, trying to focus primarily on map generation.

The first approach we will discuss is described in "*Lara-Cabrera R., Cotta C., Fernandez-Leiva A. (GAME-ON 2012) PROCEDURAL MAP GENERATION FOR A RTS GAME"*. This approach involves using a genetic algorithm for generating and evolving balanced maps (i.e., maps that give the same advantages / disadvantages for all players) for real-time strategy (RTS) games. In this case, the game used for practical applications is "Planet Wars", a game preferred as it was also featured in "Google AI Challenge, 2010". As the authors mention, their method is considered "an offline method that generates necessary content, using random seeds and deterministic generation and following a generate-and-test schema". Each individual of the algorithm has a total of 84 parameters. The algorithm will run for 100 generations, each generation having 40 individuals. It also uses a mutation rate of 70% and crossover rate of 75%. The selection of the individuals is done in a tournament way, from two individuals, the one with the highest score (fitness) is chosen to continue into the next generation. In order to compute the score of an individual, a genotype-to-phenotype transformation was used. After the normalization of the parameters, the map is finally generated. The convergence of the algorithm can be observed in Figure 1.1. However, the obtained map is just a random one. As mentioned above, the authors want to create maps as balanced as possible in order of having a game as fair for both players as possible. Evaluating how balanced a map is directly from the initial state is a difficult task so the authors approach was to let a bot play the game until the end as two players would and evaluate the balance coefficient based on the final state of the map. As the authors noted, this generation method may be improved by working on the symmetry of the generated maps and also the fact that the

balance coefficient is computed considering both players have exactly the same skill is not realistic and here is also a point that may be improved.

In conclusion, **genetic algorithms** can be a solution for the dynamic map generation task. Although, the game described in the earlier-mentioned approach is pure fictional, without any real-life equivalence. So, it is not very clear from the above description how well do genetic algorithms work for real life like maps. Also, we have to note that the map generation is done before the start of the game, no further generations occur.

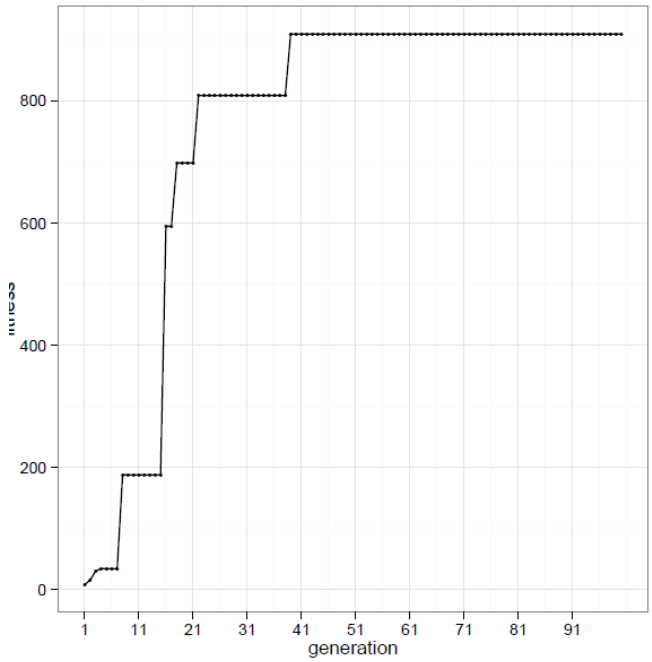

Figure 1.1

The second approach we will discuss is described in "*de Araújo L.J.P., Grichshenko A., Pinheiro R.L., Saraiva R.D., Gimaeva S. (2020) Map Generation and Balance in the Terra Mystica Board Game Using Particle Swarm and Local Search. In: Tan Y., Shi Y., Tuba M. (eds) Advances in Swarm Intelligence. ICSI 2020.*" This approach involves using "particle swarm optimization (PSO)" for generating balanced maps, fulfilling at the same time a given subset of requirements". In this case, the game used for practical applications is a modern and complex board game named "Terra Mystica". The first step is to define a set of requirements the final map must satisfy. The authors aforementioned have defined a set of four requirements for the "Terra Mystica", but for the sake of keeping the generality of this preview, we won't

mention them as they are game specific. Let them be REQ_1, REQ_2, …, REQ_n. For each such requirement, a function that counts the number of places where the respective requirement was broken is defined. Let them be f_1, f_2, … f_n. The goal is to find an algorithm that minimizes the sum of these functions, $F_{tot} = f_1 + f_2 + … + f_n$. Ideally, a map is perfect if F_{tot} is 0. Particle Swarm Optimization is a paradigm used for optimizing non-linear functions, inspired by the evident collective intelligence in several natural systems such as bird flock, bee swarm, among others. In this case, considering that we know exactly how many tiles are on the map and how many tiles of each kind as well, the author proposes the following approach: first, define the set of particles, each particle representing a tile of the map. Create a solution vector which maps each particle to a map tile. Finally, a standard PSO algorithm implementation is applied, each particle position being represented by a real numbers vectors and converted to an integer numbers vector before converting to a valid solution. The author tried multiple combinations for what he considered to be the most important parameters of PSO for this task: "inertia weight, cognitive scaling, social scaling". For each hyperparameter set, the execution of the algorithm was stopped after 5 minutes of running. Figure 1.2 shows the score and the convergence time to that score for each hyperparameter set.

In conclusion, **particle swarm optimization** seems to be a promising solution for the dynamic map generation task. Although, the game described in the earlier-mentioned approach is quite static, considering the known number of tiles and the number for each kind of tile, only their proper placement for a balanced game being the actual challenge. However, we have to note that the map generation is done before the start of the game, no further generations occur.

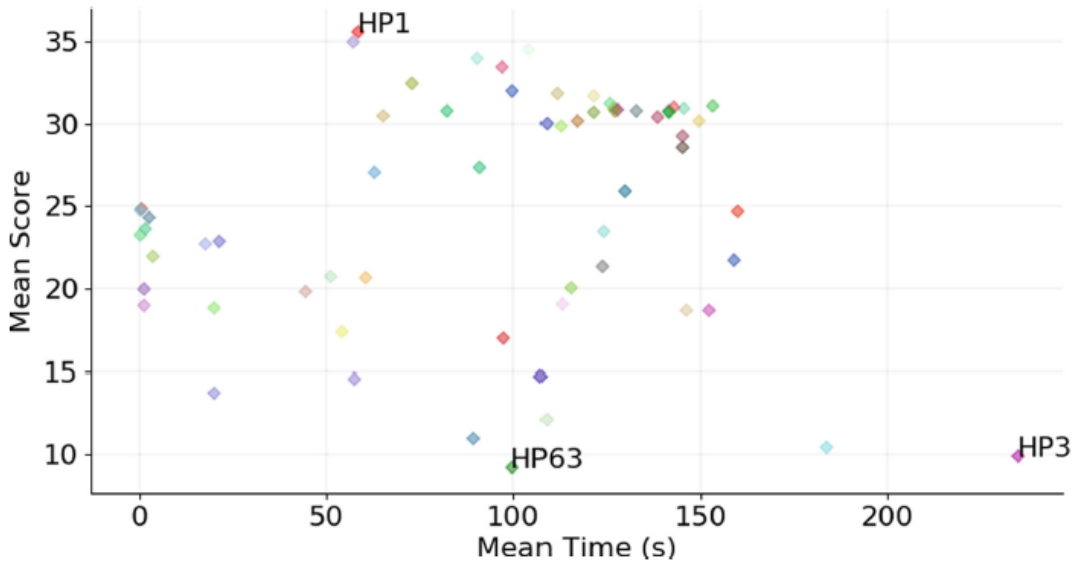

Figure 1.2

Another approach discussed in the same paper as the previous one is the Steepest ascent hill climbing with random restart (SAHC), a greedy metaheuristic less prone to getting stuck into local optimum than the classic hill climbing method. In terms of map generation, it was used in the following way: Start from a random tile and swap it with one of its neighbors. The score of the algorithm was taken at every second and the stop time was found empirically somewhere at 270 seconds. Figure 1.3 shows the score and the convergence time for the algorithm.

In conclusion, **steepest ascent hill climbing with random restart** seems to struggle for the dynamic map generation task. We should also mention that the game on which the algorithm was used it's not really complex in terms of map generation yet the results of this approach are not really impressive.

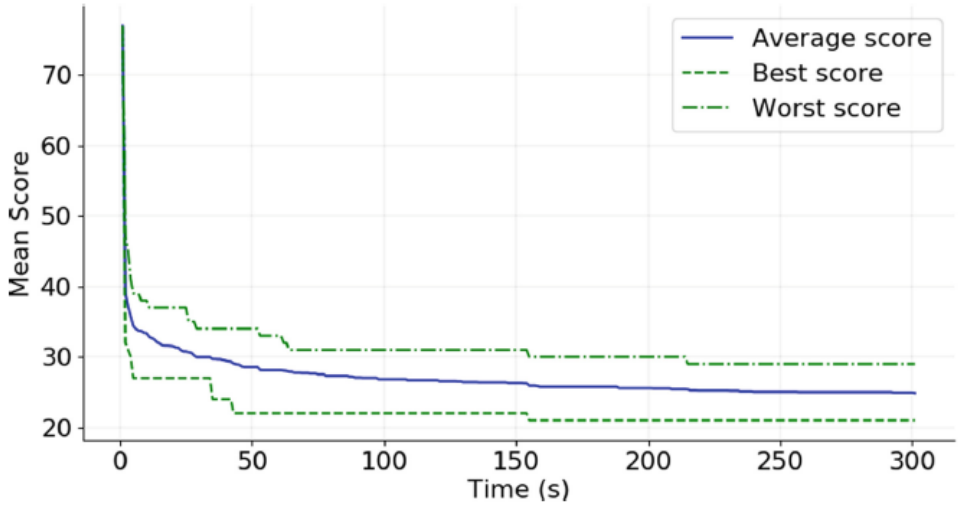

Figure 1.3

We are going to analyze a more statistical / mathematical approach, proposed by S. Snodgrass and S. Ontañón, *"Learning to Generate Video Game Maps Using Markov Models," in IEEE Transactions on Computational Intelligence and AI in Games, vol. 9, no. 4, pp. 410-422, Dec. 2017, doi: 10.1109/TCIAIG.2016.2623560.* As the title suggests, the authors are using Markov Models in order to generate video games maps. The authors are trying to come with a generation method that is widely applicable, not just for a specific game. They are trying

to achieve these using Multi-dimensional Markov Chains (MdMC), Hierarchical Multi-dimensional Markov Chains (HMdMC) and Markov Random Fields (MRF). As these are machine learning approaches, a training set containing well made maps by professional designers are required. Due to this, the quality of the dataset, in terms of size, content quality and game variety may heavily impact the final results. For this case, the models will be tested on three different games. The main advantage of statistical models over the classical "generate-and-test" approaches is the ability of generalization. In the case of MdMC, given a graph of states (no matter what these states represent), a probability distribution is defined for each state based on another set of states from the graph. Markov Random Fields (MRFs) model probabilistic relations between surrounding states in the states graph. First of all, the data (the maps, which in this case, 2D context is used) are represented as a matrix, where each element M(i, j) represents a type of tile on the position (i, j). This map is encoded into a smaller one by clustering areas of a given size into a single tile, based on different approaches. In order to train a Multi-dimensional Markov Chains model on a set of maps, a network architecture is chosen. Then, for each training level and for the given network architecture, the number of times each tile type follows each tile configuration is determined. Then, for each tile type following each configuration, a probability is computed based on the appearance frequency of this scenario (i.e., the number of times a tile type following a configuration of tiles has appeared, divided by the number of times that configuration appeared). Now, as the model is ready, the actual map generation may begin. Considering that we are in a 2D context, we have to generate a matrix. The model generates a tile at a time, this process being repeated until the map is done. To prevent the case where an unseen state has to be sampled, a simplified Katz backoff model is used. The HMdMC and MRF methods won't be described as they are improvements of the MdMC method described above.

In the Table 1.1 we can observe the performance of each model in terms of playable maps from a total of 1000 generations while in Table 1.2 we can observe the necessary time to generate a single map (in milliseconds).

Approach	*Super Mario Bros.*	*Loderunner*	*Kid Icarus*
MdMC	66.8%	1.5%	0.0%
M-HMdMC	**90.4%**	-	-
C-HMdMC	78.0%	**1.8%**	0.0%
MRF	21.9%	0.0%	**3.2%**

Table 1.1

Approach	*Super Mario Bros.*	*Loderunner*	*Kid Icarus*
MdMC	19.09	6.91	62.38
M-HMdMC	17.11	-	-
C-HMdMC	34.83	19.64	210.45
MRF	207684	8359.23	242939

Table 1.2

Note: M-HMdMC stands for Manual-HMdMC and C-HMdMC stands for Clustering-HMdMC.

In conclusion, as the authors also state, the **Markov Models** are quite limited, especially due to their lack of long-range dependency across a map. We can observe that they can't produce results for more complex games (Loderunner and Kid Icarus) and struggle even on simpler games as Mario. We may assume that there is no point in trying to extend them further to generating 3D maps or to use them for modeling real life like tasks.

We are going to analyze a quite bold approach for the dynamic map generation task: using Tree Search algorithms. This approach was proposed in "*Bhaumik, Debosmita & Khalifa, Ahmed & Green, Michael & Togelius, Julian. (2019). Tree Search vs Optimization Approaches for Map Generation"*. The authors tried to generate 2D game maps using just tree search algorithms. First of all, let's see how a tree search algorithm works in general: the algorithm starts with a base node called "root". Then, it expands more children nodes using different techniques Depth-First Search (DFS), Breadth-First Search (BFS), Greedy Best First Search (GBFS) and Monte Carlo Tree Search (MCTS). We will mention few words about how each algorithm works on a tree.

- BFS is usually implemented using a queue data structure. It starts with a given node (we will consider it to be the root in our case) and add it to the queue (which is initially empty). Then, at each step, the first node in queue is taken and all the neighbors of that node that were not already considered are pushed into the queue for later processing. In a visual way, the algorithm starts with the root (which is considered to have depth 0). Then each level is fully visited befor1e getting to the next level. BFS has linear complexity in terms of number of nodes.

- DFS is usually implemented using a stack data structure (the recursion stack most of times). It starts with a given node (we will consider it to be the root in our case) then it looks for the first neighbor of the node that was not visited yet. The same procedure is repeated recursively. When a node has no more available neighbors, the recursive call returns. In a visual way, the algorithm starts with the root (which is considered to have depth 0) then it traverses a chain of nodes with consecutive depths. DFS has linear complexity in terms of number of nodes.
- GBFS works the same as normal BFS with the particularity that the nodes are not expanded in a random manner. A priority queue and a cost function are used to determine at each step the node which has the greatest potential to yield a better solution.
- MCTS is a heuristic tree search algorithm that uses 4 operations (selecting, expanding, simulating / rollout, updating) to obtain the best solution from the solution set. In the selection phase, the node expected to yield the best solution is chosen and then a "winning path" is determined until it reaches a leaf. To evaluate the best node, the authors used the UCB1 function (Figure 1.4). In the expansion phase, from the leaf we are in, we create a new (or more) nodes, representing the future choices available. In the rollout phase, we create a final state node via sampling to which we try to get from the current node. In the updating state, the whole path till the root is recalculated based on the value attributed to the leaf node.

$$UCB1_i = \frac{V_i}{n_i} + c\sqrt{\frac{\ln N}{n_i}}$$

Figure 1.4

V_i = total accumulated reward for that node
n_i = total number of visits for the i node
N = total number of visits of the parent node
c = constant for balancing

Now we will show how to represent the problem so that we can apply the tree search algorithms on it. The representation also influences the generation speed and the style of the generated content. It's obvious that, due to using tree algorithms, the problem has to be modeled as a graph, each node and edge getting a meaning. The authors proposed three representations: Narrow, Turtle and Wide. Before discussing these representations, we mention that our space

consists of 2D maps and each map consists of tiles. For the Narrow representation, each node keeps the map and the tile that will be modified. Each edge from this node represents the changes the tile may suffer. The space of states is constrained by the number of tile kinds but is greatly increased by the fact that the node also keeps the tile for each map, so each node has pairs of (map, tile). For the Turtle representation, each node keeps the map and the position of the "turtle". Each edge represents a moving direction (only the 4 cardinal directions are used) and the modification of the tile. The space of states is constrained by the number of tile kinds + the 4 directions but is yet again greatly increased by the tile position kept in each node. For the Wide representation, each node holds just the map itself. The edges represent an encoding to the new map that can be obtained from the current state.

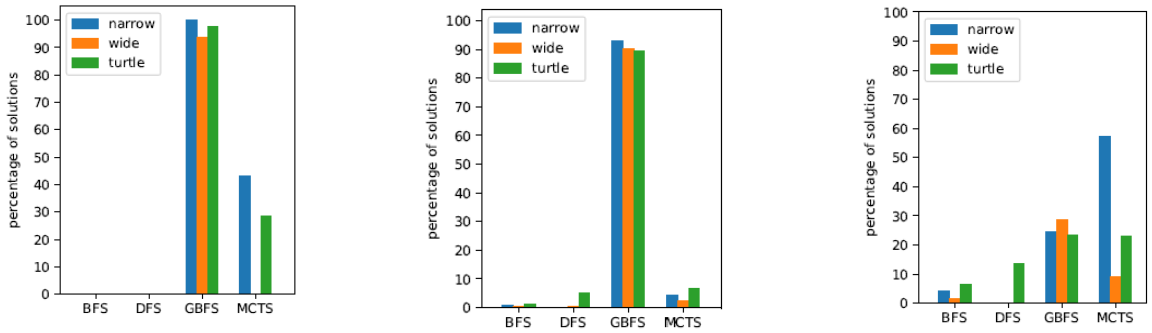

Figure 1.5

In Figure 1.5 we can see the playability rate for each game for each of the four algorithms and also for each tree representation. We can observe that the Greedy BFS yields the highest rate of playable maps for the first two games, no matter the tree representation used. Simple BFS and DFS struggle to yield any playable map, no matter the game or representation. For the MCTS, only the Narrow representation manages to give some results for the first and third game.

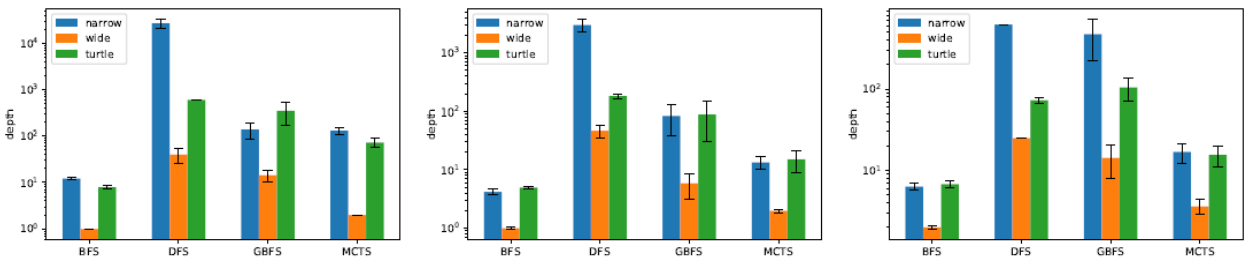

Figure 1.6

In Figure 1.6 we can observe the mean depth of the trees used to generate the maps. As expected, the DFS produces the deepest trees and also BFS produces the least deep trees.
On each configuration there are 2500 runs of 60sec or until a solution is found.

In conclusion, SOME **Tree Search algorithms** may be a decent approach for map generation task but they can perform only in static context, not dynamically. But in terms of results, time, resources and implementation difficulty they may be feasible.

Another approach we are going to analyze is the closest to the main theme of this paper, proposed by Ping K. and Dingli L. in "*Conditional Convolutional Generative Adversarial Networks Based Interactive Procedural Game Map Generation*". The authors describe the use of Generative Adversarial Networks and Convolutional Neural Networks into the task of dynamic map generation. The whole procedure makes use of Conditional Generative Adversarial Networks (cGAN, extension of simple GAN), Convolutional Neural Networks (CNN) and classical procedural methods. The main difference from a tradition GAN is that, cGAN can also add custom user input to the original input, offering higher customization possibilities. The map generation process described in this article works in the following way: firstly, a designer manually draws a rough sketch of the map. Then, this mask is fed to the cGAN that generates a labeled map, where each label encodes a map element (tree, hill, road, etc.). The obtained labeled map is further processed by a CNN to tie everything together (drawing paths between areas, giving additional meaning to the labels, like type of road, height of hills, etc) properly and obtain a final model that will serve as a mold for 2D or 3D rendering. Figure 1.7 depicts an illustration of the above-mentioned architecture.

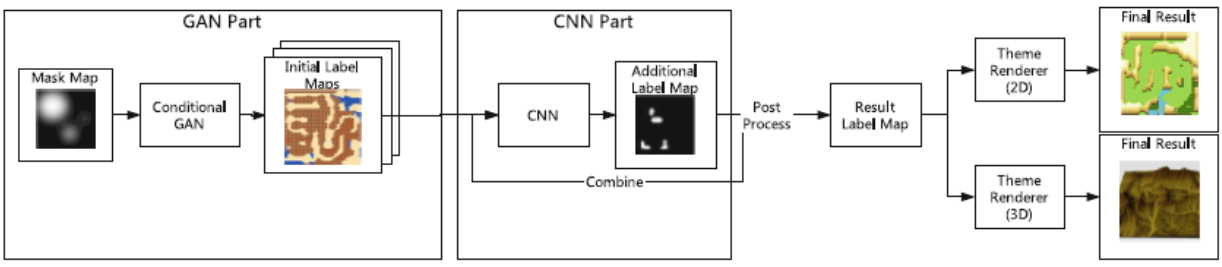

Figure 1.7

Further, we will analyze the network architectures proposed by the authors into achieving this task. The cGAN is composed from two deep neural networks: a generator (G) represented by

a -Net encoder-decoder and the discriminant (D). For training the generator, it is fed with the custom mask (X) as the training set and a labeled map (Y) as the expected result. So, the generator G receives a set of (X, Y) pairs. The generator will be used to generate labeled maps from custom masks, this was expressed as G(X) (the result of the generator for the given mask X). We want to use the discriminator D to differentiate between a genuine map and a generated one. Formally, given (X, Y) the genuine map and (X, G(X)) the generated map, D must decide which is which. As the author mentions, the two networks are trained to play the following min-max game. The performance of the network was measured using the "Intersection over Union" method, which is considered a good metric for measuring overlapping between two masks, in this case, assessing how well a generated map "overlaps" over the genuine one, for a given input. The generator was trained using the MeanSquaredError metric (MSE) because it determines how close two maps are from each other, combined with categorical cross-entropy, as it forces the maps to have the same style (label-wise). For the discriminator, binary cross-entropy was used, as it has to respond with binary values to the given data. The RMSprop was used as an optimizer function, being chosen experimentally. The greatest set-back of this approach is given by the fact that the main training source of the whole system is the hand-made drawn maps. Due to this, this method requires high human interaction. Firstly, the map mask has to be drawn by a specialized designer. Then the results of the generator upon the given mask must be monitored and evaluated by the same designer, if the map looks like his idea. If the map is not suitable, he must improve the mask and retake the whole generation process from the beginning.

In conclusion, deep neural networks, in this case represented by **Conditional Generative Adversarial Networks** and **Convolutional Neural Networks**, seem to be a great approach for the dynamic map generation task, the only real challenge being represented by gathering a sufficient quantity of good quality training data.

The last approach we will discuss remains in the sphere of Adversarial Neural Networks. Gisslén et all propose a solution called "*Adversarial Reinforcement Learning for Procedural Content Generation*" (ARLPCG). As the name suggests, this solution uses deep reinforcement learning to generate new content.

The approach itself consists of two RL agents: the generator and the solver, that are dependent on each other. The generator creates an environment which is then tested by the

solver. The generator receives feedback (rewards and observations) from the solver plus some auxiliary content which is set by the developers. This way, the generator learns to create different types of environments, of different kind, difficulty, complexity… etc. After a new environment is created by the generator, it "challenges" the solver to get an as good as possible result, making the solver more robust, more prone to solving new challenges never seen before and reduce the amount of hard-coding in order to solve the task. A major advantage of this approach is that, it produces two RL agents that can be used independently for other similar tasks and also the pair itself. A representation of the system can be observed in Figure 1.8:

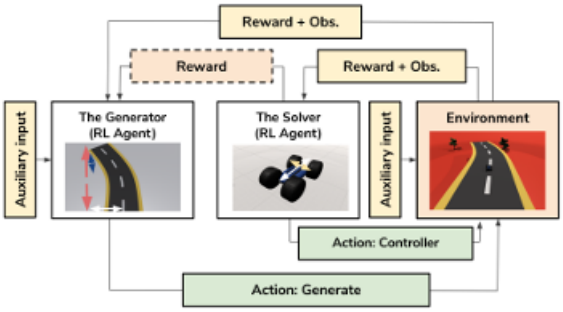

Figure 1.8

The generator will create the environment step by step, from scratch, instead of adjusting an existing random environment. The generator will create the new element / segment only after receiving feedback from the solver, this way assuring that the environment is playable at every moment of time. An auxiliary input is manually added in order to have better control over the generation process. This way it ensures a better environment generalization otherwise the two RL agents will converge to an optimum value, case in which the environments will be highly similar.

The solver is a RL agent that is penalized when not able to reach its destination (this way the generator is also warned that it should not produce impossible terrains) and is rewarded when able to get to the final goal.

The two agents are not trained in a min-max game way (i.e., one is not trying to over best the other). Instead, they work in a collaborative way, sometimes both may gain or lose, sometimes one may gain and the other one loses.

In conclusion, a system composed of a generator and a solver, both powered by RL agents along with some external interference (manually set auxiliary input) was able to successfully generate and traverse different types of environments. This was also possible due

to the fact that the two agents collaborated with each other instead of simply challenging each other. So, we can say that this approach is truly similar to what we want to achieve, in terms of dynamism and variety.

Chapter 2

The Dataset

2.1 Overview

To achieve our task, we will make use of semantic representation of the CITY-OSM dataset. The dataset consists of 1671 aerial images, weighting almost 45GB, of Berlin, Chicago, Paris, Potsdam, Tokyo and Zurich cities. The raw images were segmented into a much simpler representation consisting of roads (marked with the color 'blue' RGB(0, 0, 255)), buildings (marked with the color 'red' RGB(255, 0, 0)) and the rest of the background (marked with the color 'white', RGB(0, 0, 0)), not being of direct interest for our task. In its raw form, is very hard to work and extract useful features from the dataset. Given this, we will apply a number of transformations on it in order to make it as suitable as possible for our task.

2.2 Preprocessing

2.2.1 Resizing

The raw images are very big (i.e., 2611 x 2453 resolution => ~10 MB per image) and also, they don't have the same size. For this we will resize them to a convenient size. A resolution of 512 x 512 was chosen as it is small enough to work with but high enough to keep the important aspects of the image. We don't mind that the images will be a bit distorted from the original (transformation from rectangle to square) as it doesn't matter if a road is a bit wider / longer / skewed … etc.

Here we can observe a sample obtained after the resizing step:

Figure 2.1

2.2.2 Road extraction

For the moment we don't need any other element from the image but the roads. As the roads are composed of blue pixels (some may not be perfect blue), we will keep only the pixels that lay within a manually set color range. We obtain a binary image where the road is marked with black pixels while the white pixels mark the background (non-interest area).

Here we can observe a sample obtained after the road extraction step:

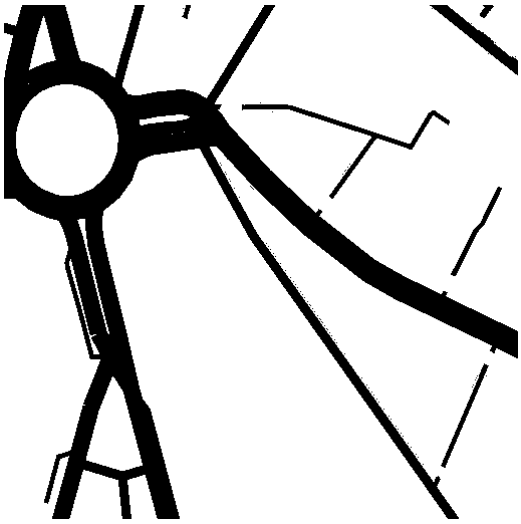

Figure 2.2

2.2.3 Closing

We will apply one iteration of closing morphological transform using a 3 x 3 rectangular kernel in order to get rid of the isolated pixels (i.e., noise) so they won't affect our feature extraction later. We obtain a binary image where the road is marked with black pixels while the white pixels mark the background (non-interest area).

Here we can observe a sample obtained after the closing step:

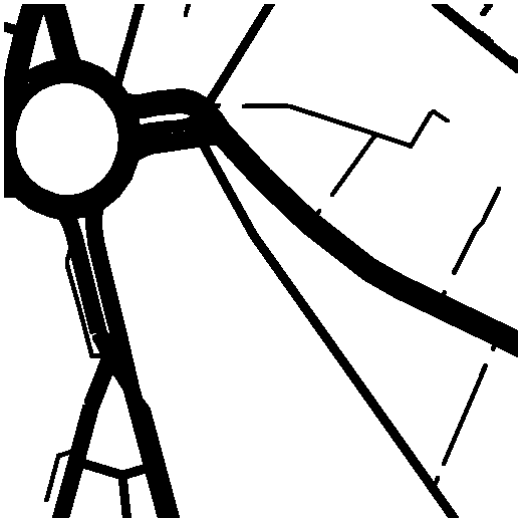

Figure 2.3

2.2.4 Erosion

We will apply two iterations of erosion morphological transform using a 3 x 3 rectangular kernel in order to fill in eventual white pixels from the road so the road is complete and well filled withing its edges. It also makes the road thicker, which will help us in the next step. We obtain a binary image where the road is marked with black pixels while the white pixels mark the background (non-interest area).

Here we can observe a sample obtained after the erosion step:

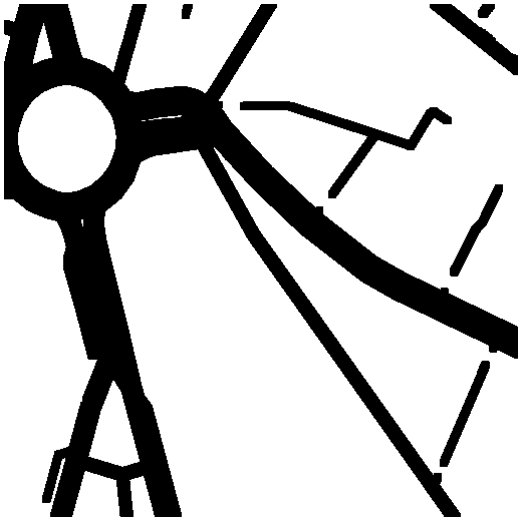

Figure 2.4

2.2.5 Skeletonize

Due to the fact that roads from an image or across images may differ in width, we will create a skeleton of the roads network where the width of the roads is almost equal. This is achieved using scikit image's *skeletonize* method. The method is an implementation of Zhang's algorithm [15] used for image content thinning. We obtain a binary image where the road is marked with white pixels while the black pixels mark the background (non-interest area).
Here we can observe a sample obtained after the skeletonize step:

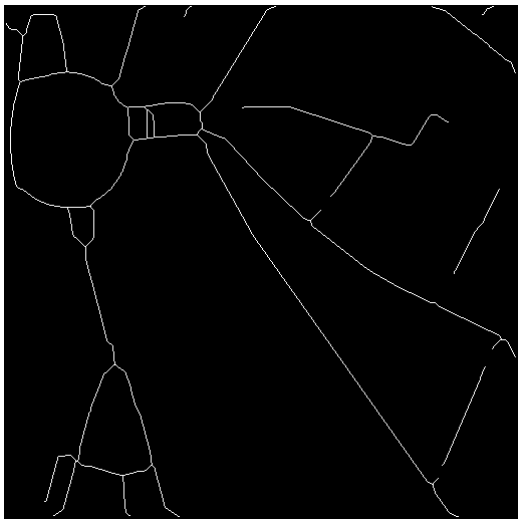

Figure 2.5

2.2.6 Binary Dilation

The final step consists of applying a binary dilation morphological transformation with a selim disk kernel. This way, we ensure that the roads obtained at the previous steps won't have any discontinuities at pixel level. We obtain a binary image where the road is marked with white pixels while the black pixels mark the background (non-interest area).
Here we can observe a sample obtained after the binary dilation, final step:

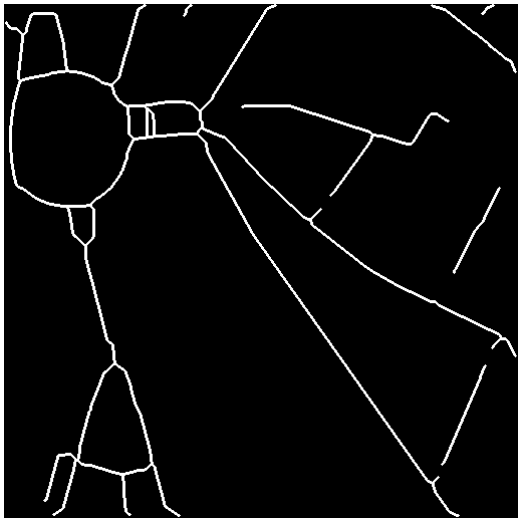

Figure 2.6

We can summarize the steps above with the following diagram:

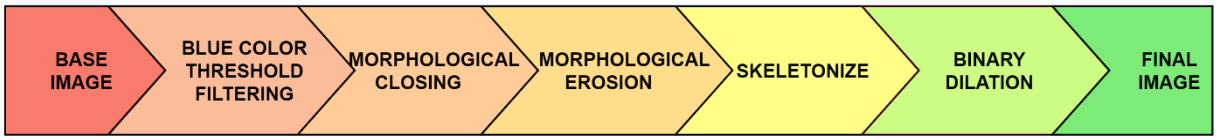

Figure 2.7

Chapter 3

Theoretical aspects

In this chapter we are going to present some theoretical aspects that were and will be used throughout our project.

3.1 Computer vision

Computer vision (abbr. CV) is a branch of Artificial Intelligence whose scope is to give machines the ability to "see", i.e., to let computer understand, process and generate visual content like images, videos, etc. Some common computer vision tasks would imply image altering (applying filters, distortions, skewing, rotating, etc.), object detection (detect certain objects in pictures or videos like animals, road signs, people's faces, etc..), model building (generate 3D elements, schemas, etc.), image segmentation (keeping only the point of interest from an image, labeling elements withing a picture, etc.). As our task involves working with images, its no surprise that we will use it in our project. We will use two important frameworks for handling computer vision tasks: OpenCV (Open-Source Computer Vision Library) and Scikit Image (skimage).

3.2 General Adversarial Networks (GANs)

General Adversarial Networks (abbr. GANs) is a deep learning approach used to tackle the task of content generation. GANs are an unsupervised learning method, learning to generate similar content to the input data they are fed with, in a trial-and-error manner. GANs are mainly composed from two major components: the "generator" and the "discriminator".

Generator's task is to create new content given an unlabeled dataset, trying to match the distribution of the input data, while discriminator's task is to decide whether a given sample was extracted from the existing data set (the generator has no information regarding the base data set) or it was generated by our generator, using.

The two models forcibly train each other in a min-max game approach: Let's consider both the generator and the discriminator being two players trying to win the following game: generator wins when he is able to produce a new sample which can't be distinguished by the discriminator as being "fake" while the discriminator wins when he correctly asserts a randomly given sample as being part of the original dataset or being created by the generator. In this manner, both models increase their accuracy in order to be more prone to win the "competition". We may assume that the accuracy of a GAN is equal to the number of failed decisions made by the discriminator while assessing data (i.e., the more samples which couldn't be classified as "fake" by the generator, the better the quality of the generator).

3.3 Reinforcement Learning

Reinforcement Learning (abbr. RL) is an area of machine learning which can be formally described in the following manner:

Given an "environment", defined at each moment by a "state", an "agent" along with a set of "actions" he is able to perform and a reward function that provides feedback to the agent based on the chosen action and the state of the environment, the agent has to learn the optimal sequence of moves that maximize the reward function. Reinforcement Learning is a goal-oriented method which doesn't require any prior training data besides the initial state of the environment. The agent will learn itself based on trial and error which action is the best to take for a given state of the environment. During the training process, the RL agent will develop a "policy". A policy is a function that maps a given state of the environment to the action considered best to take at that moment by the agent. The succession of states and actions through which the agent proceeds is called the "trajectory" of the agent.

In case the algorithm used by the agent to learn makes use of Deep Neural Networks, it is then regarded as Deep Reinforcement Learning.

The basic idea of Reinforcement Learning is represented in a simplified manner in the Figure 3.1:

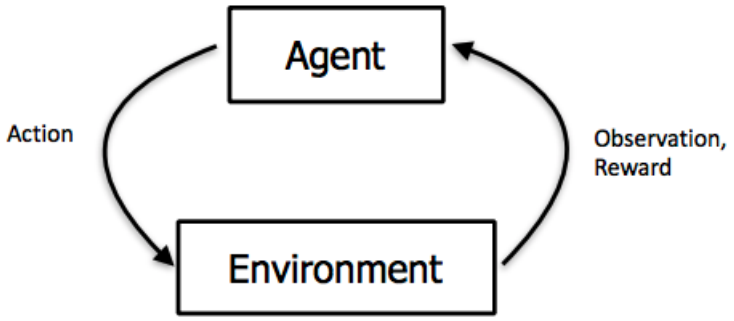

Figure 3.1

3.4 Lindenmayer Systems

Lindenmayer systems (abbr. L-systems), firstly introduced by the biologist Aristid Lindenmayer on behalf of development of filamentous organisms, assuming that the cells that compose the new filaments suffer changes based on their current state and some other information (input) received by the older, neighboring, cells. In computer science, Lindenmayers systems are abstractly shaped as grammars, which can be either finite or infinite, deterministic or non-deterministic. Applying such rules in a recursive manner yield branching-like structures.

3.5 Undirected graph:

An undirected graph can be defined as a pair of sets (V, E), where V is a finite, non-empty set, called “vertices” while E is a set of unordered pairs from V, called “edges”. For an undirected graph G, we note it as G = (V, E), meaning that the graph G is made of the set of vertices V and the set of edges E.

An edge is a binary, symmetrical relationship between two vertices of the V set. i.e., if the edge (x, y) ∈ V then (y, x) ∈ V too.

Ex. For the graph in the right we have:

V = {1, 2, 3, 4, 5, 6}

E = {[1, 3], [1, 6], [2, 5], [2, 6], [5, 6]} (not mentioning their symmetric too)

G = (V, E) = ({1, 2, 3, 4, 5, 6}, {[1, 3], [1, 6], [2, 5], [2, 6], [5, 6]})

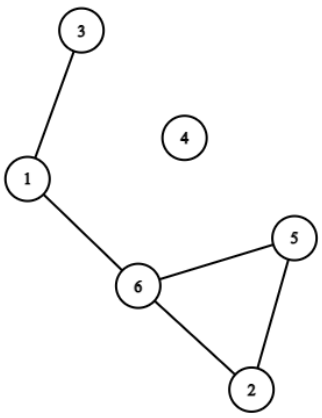

Figure 3.2

3.6 Breadth First Search (BFS):

The Breadth First Search (abbr. BFS) algorithm is a graph traversal algorithm. It starts from a specified node or a random one which is called “source node”. The algorithm then tries to visit the neighbors of the current node which were not already reached. The usual implementation of the algorithm makes use of a queue data structure (First In First Out) to keep track of the nodes that will be explored. For a graph G = (V, E), the algorithm runs in O(|V| + |E|) complexity in both time and memory, considering the graph is represented through the adjacency lists, where |V| denotes the number of nodes from the graph and |E| denotes the number of edges from the graph.

3.7 Manhattan distance

In a normed vector space, we define the Manhattan distance as $d_1(x, y) = \sum_{i=1}^{n} |x_i - y_i|$, where x and y are two points from our space and n is the number of dimensions of our space.

Chapter 4

Experiments and Results

4.1 Feature Extraction

After obtaining the skeleton roads images, we are going to start extracting some information from the images. Our goal is to convert the road network into an undirected graph.

4.1.1 Determine Nodes

Firstly, we will define the nodes (vertices). In order to achieve this, we came with an own heuristic approach, we will describe in the following lines.

Let's consider the skeleton image of the roads network as a binary image where a pixel labeled with 0 (black) represents background (a non-interest element) and a pixel labeled with 255 (white) represents road (an interest element). Firstly, it is obvious that our nodes should be elements placed on to the interest elements (road). From this simple condition, we will make our heuristic process only white pixels, ignoring the black one. Secondly, we can assume from the visual representation that one of the most interest points from a road network are the intersections. We will consider an intersection to be a point where at least 3 roads meet (see the images below).

Figure 4. 1

Our goal for now would be to determine the intersections of the roads network. Our approach for solving this task:

1. Consider each white pixel (has potential to be an intersection)
2. Draw a circle of radius R (R to be determined empirically) centered in the pixel we chose at step one.

3. Count the number of distinct roads (segments) intersected by our circle defined and step two.
4. If the number of intersected segments is greater than or equal to 3, we have an intersection so we place a graph node in the point considered at step one

(see Figure 4.2. The circle was depicted with green and the intersected segments with red)

Figure 4. 2

Another interest point from our roads network would be the end of the roads. To achieve this, we will apply the same reasoning as in the previous method, but now instead of wanting the circle to intersect 3 or more segments, we will want the circle to intersect exactly one segment, case in which we will place another graph node. (see Figure 4.3. The circle was depicted with green and the intersected segments with red)

Figure 4.3

Finally, there is one more class of interest points that are worth considering as a graph node: turning points (i.e. a point where the road does a turn to the left or right). For finding such points, we will still make use of the previous approach: we will look for points which circle drawn from intersect exactly two roads (segments) but as we want to have a turning, we will put the condition that the angle formed from two random points (one point taken randomly from each intersected segment) and the fixed point from step 1 as vertex, is less than a number of degrees D (D chosen empirically).

(see in Figure 4.4 a turn at around 90 degrees)

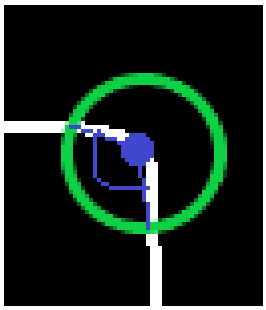

Figure 4.4

Applying the aforementioned strategy, we obtain the following:

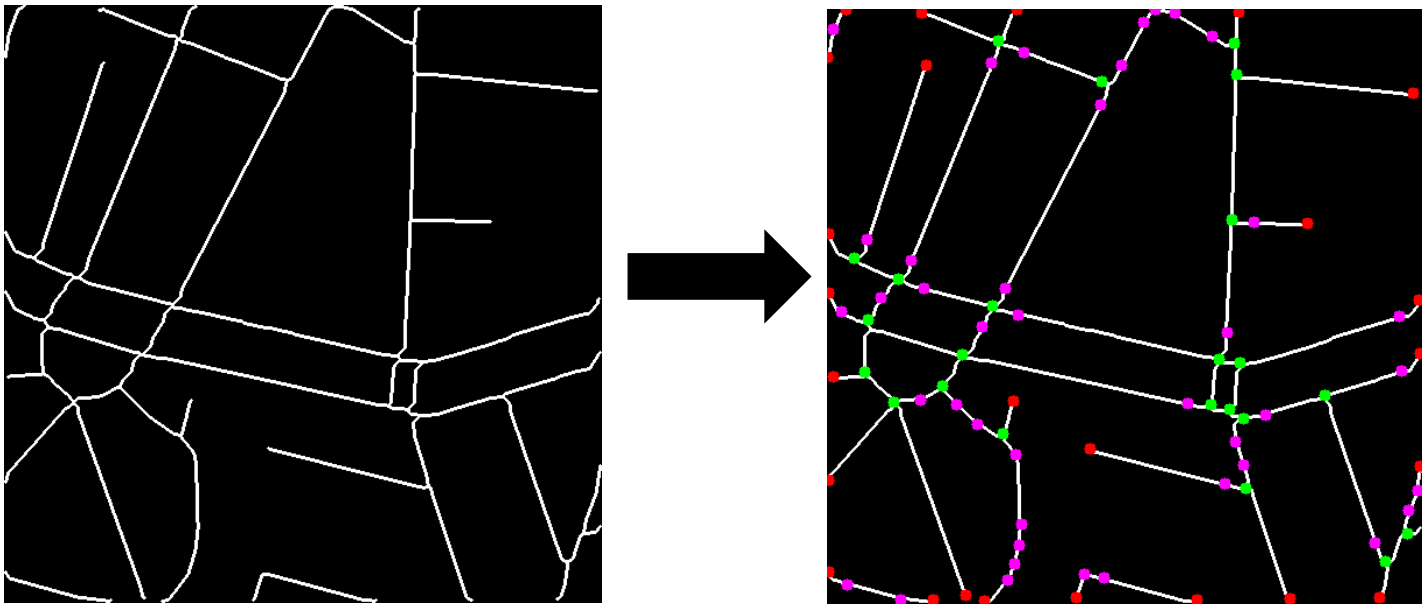

Figure 4.5

Note: The nodes placed in intersections are marked with green color, the nodes placed at road ends are marked with red color and the nodes placed at turning points of the road are marked with magenta color.

Additional implementation details:

- Some points are not perfectly aligned on an intersection, because during traversal and the radius of the drawn circle, some neighbor pixel may be chosen as the actual intersection, but at graph level we do not mind about this error.
- We put as condition that we can't have two graph nodes closer to each other at a distance smaller than 30 pixels (the value was chosen empirically and the metric used as distance was the Manhattan distance)
- The radius R of the circle drawn from a pixel to check for intersections was chosen empirically as 5

- The angle for considering a road turn must be (chosen empirically) less than or equal to 160 degrees and greater than 20 degrees
- Some misalignments may appear due to the fact that the roads are more than one pixel wide

4.1.2 Determine Edges

Having found the nodes, we can now proceed with determining the edges.

We will apply a BFS from each node, expanding only the white pixels (road) and ignoring the black ones (background). When we find another node while searching, we will draw an edge between the start node and the one we found. Because our nodes are actually a sole pixel and the roads are thicker than one pixel, we had to do a little trick to prevent a node to expand beneath another node instead of stopping there. We marked an "influence area" for each node as a circle of radius 5 so that, when we expand our queue, we don't need to find exactly the pixel depicting another node, but one of its influence points. Considering that the nodes were forced from the beginning to have a distance between their centers of at least 30 pixels, we ensured this way that there are no two distinct nodes whom interest areas may overlap.

Using the described approach, we generated a txt file for each image, with the following format:

m

X_1,Y_1

X_2,Y_2

…

X_m,Y_m

The first line contains an integer M, denoting the total number of edges. The following M lines have the format X,Y where X and Y represent the nodes. X,Y means that exists and edge between the nodes X and Y. The edges are unique, a pair X,Y appears only once. We ensured that if X,Y appears, the equivalent Y,X will not appear.

A node (i.e. a pixel) for our image is represented as a pair (r, c) with r, c $\in \{0, 1, 2 \dots, DIM - 1\}$, where DIM is the size in pixels of the image. In our case, DIM = 512 as our images have 512x512 shape. r represents the row index of the pixel while c represents the column index of

the pixel. To simplify the work with nodes and edges, we encoded each pixel with a unique integer given by the following function:

$$encode : AxA \rightarrow \mathbb{N}, where\ A = \{0, 1, 2 \dots, DIM - 1\}$$
$$encode(r, c) = r * DIM + c$$

Also, we define the reverse function that gives us coordinates pair from the encoded value:

$$decode : \mathbb{N} \rightarrow AxA, where\ A = \{0, 1, 2 \dots, DIM - 1\}$$
$$decode(x) = (\left\lfloor \frac{X}{DIM} \right\rfloor, X\ modulo\ DIM)$$

The node values in the text files are in encoded format for simplicity in parsing.

Here is an example of how the method works:

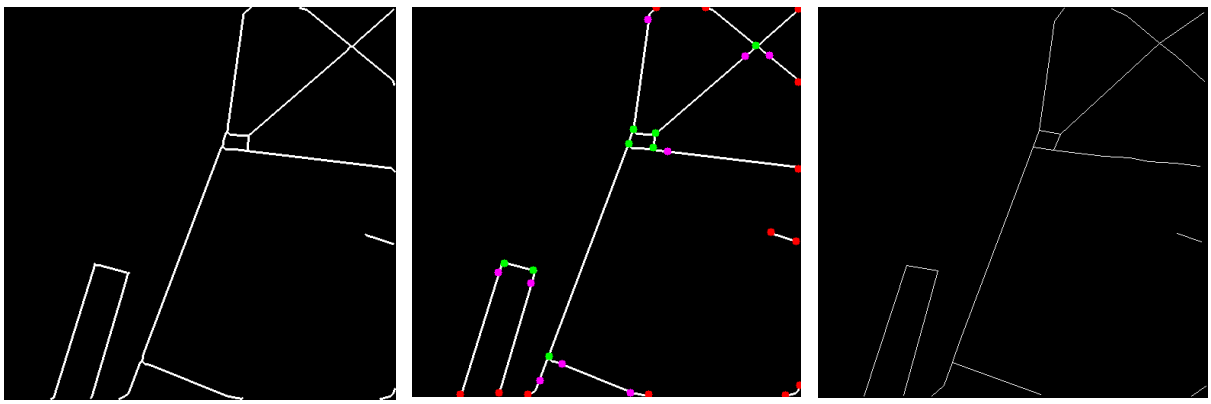

Figure 4. 6

4.2 Feature Extraction

In addition to the graph representation, we created a "feature file" for each image. For an image, we will consider as relevant the following elements:

- Road segments and their length
- Turning angles of the road
- Intersections and how many branches does an intersection have
- Road ends

These features are extracted at the same time with the nodes (the nodes are classified and intersection nodes, angle nodes and road end nodes, as mentioned above). The obtained file has one feature per line, features being coded in the following way:

- DIST_<value> meaning we have a continuous road segment with length equal to <value> (<value> will always be a whole number greater than 0)
- ANGLE_<value> meaning we have a road turn with <value> degrees (<value> is a whole number which will lay in range (-360, 360))
- INTER_<value> meaning we have an intersection with <value> branches (where <value> is a natural number in range [3, 7])
- END_True meaning we have a road end

Let's observe the distribution of these attributes over our image set:

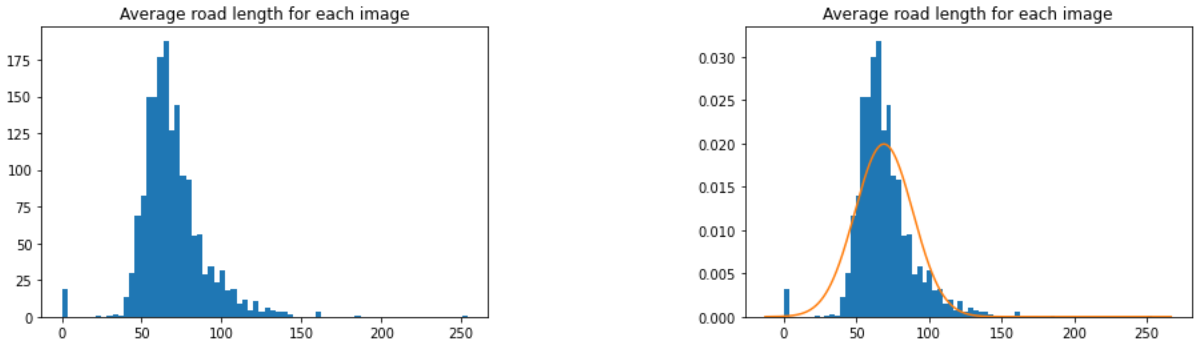

Figure 4.7

In the Figure 4.7 (left) we can observe the histogram of average road length over the set of images. In the Figure 4.7 (right) we also tried to apply a normal distribution over the set of distances to observe how well would it fit for future sampling.

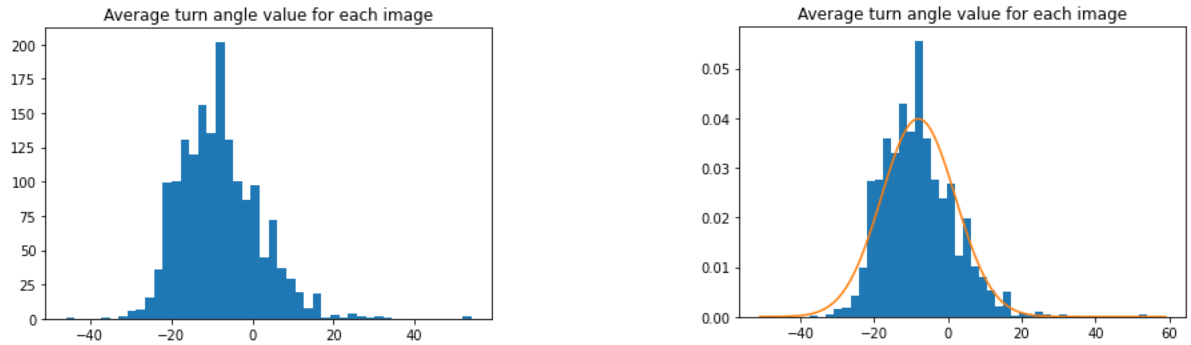

Figure 4.8

In the Figure 4.8 (left) we can observe the histogram of average turning angle over the set of images while in the Figure 4.8 (right) we tried to apply a normal distribution over the set of angles to observe how well would it fit for future sampling.

In the Figure 4.9 we can observe the number of intersections based on their number of branches along with a histogram that depicts their distribution. We observe that from 1671 sample images that we had 17727 intersections with 3 branches (an average of 10-11 such intersections for each image), 6804 intersections with 4 branches (an average of 2-3 such intersections for each image), 472 intersections with 5 branches (an average of 0.3 such intersections for each image so almost 1 in 3 images has one of these intersections), 77 intersections with 6 branches and 20 intersections with 7 branches.

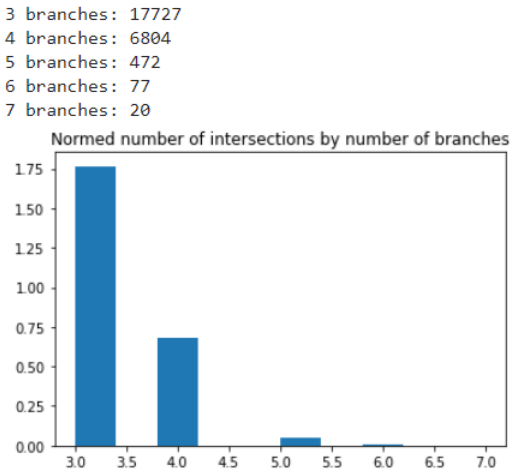

Figure 4.9

In the figure X we can observe the number of road ends over all the images. We can see that we have many images with exactly 16 road ends. Also we may say that most of the images have a number of road ends laying in the interval 8-16.

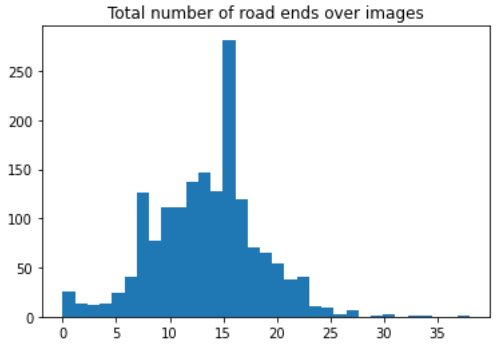

Figure 4.10

Based on the features extracted above, we will try to generate similar road networks.

4.3 Failed Attempts

Unfortunately, not everything was straight and smooth, we will present the problems, the difficulties, the challenges that appeared and some additional research that seemed helpful in overcoming those but it actually wasn't.

During the graph building stage, meaning from the raw huge images to the text document depicting the associated graph for each image, the only real challenges consisted of searching through the documentation of the used frameworks and finding the proper hyperparameters values within trial n error along with relatively big waiting times for all the image processing steps for the whole dataset.

On the other side, when coming to the part of finding a proper road generating method, a lot of research was involved, most of it without any palpable results. Following we will mention some papers we came upon to while searching for a solution, we will briefly explain what they propose, what we were hoping to extract from them and why we realized it's not a suitable approach for our task:

- NetGAN: Generating Graphs via Random Walks

This approach involves feeding graphs to a GAN architecture and then create graph samples. This approach would have been very useful considering the implementation takes just simple undirected graphs given through the list of edges, exactly as the data we created. The main problem is that NetGAN can only generate independent samples while we need a continuous network generated dynamically. Although an interesting approach, it doesn't really suit our needs.

- Misc-GAN: A Multi-scale Generative Model for Graphs

This approach involves feeding graphs to a GAN architecture which tries to extract a distribution for the given graph dataset at different levels of granularity. The network learns this distribution and then tries to transfer it to the samples to generate. As for the previous approach, this method is not suitable for generating continuous road networks, only individual samples, which is not of much use for our goal.

- GraphGAN: Graph Representation Learning with Generative Adversarial Nets

GraphGAN was designed to work for datasets where we can define the set of nodes and then predict / generate links based on existing training set. In our case, we know nothing about the nodes or the edges so this approach won't be of much help.

- Variational Graph Auto-Encoders (VGAE).

The same as for GraphGAN, VGAE was designed to work for datasets where we can define the set of nodes and then predict / generate links based on existing training set. Because we have both dynamic nodes and edges, we cant use this approach in our task.

Generative approach

Finally, we came with the following road generation strategy:

1. Based on the extracted features, we compute for each image the following information: total number of road ends, total number of angles, smallest angle, biggest angle, average angle, total number of segments, the sum of the lengths of the segments, average length, smallest length, biggest length, total number of intersections, number of intersections of each kind (with 3 branches, with 4 branches…etc.).
2. With the computed information we cluster the set of images into 16 clusters, based on the similarity of the above-mentioned information, using KMeans algorithm.
3. Every time we want to generate a roads network, we choose a cluster randomly and use its specific features for further generation.
4. When we start the generation process, at each step we pseudo-randomly choose what element will be generated next (A straight road of certain distance, a road turn of certain angle, an intersection with certain number of branches or a road end)
    a) There are some element precedence rules in order for the road to make sense:
        - After a straight road, we will generate a turn, an intersection or a road end
        - After an angle we force a new straight road (so we can observer how the road turns)
        - After an intersection, for each of its branches, we may generate either a straight road or a turn
        - After a road end we don't generate anything further
    b) When we have more options, we choose the next element kind based on a weighted random (the probability of an element to be chosen is equal to the total number of elements of its kind from the cluster, divided by the total number of elements of every kind)

i.e., Let T = total number of elements in the cluster. Let D = total number of straight roads in the cluster, A = total number of angles (road turns) in the cluster, I = total number of intersections in the cluster and E = total number of road ends in the cluster.

The next element is chosen with the probabilities $\frac{D}{T}, \frac{A}{T}, \frac{I}{T}, \frac{E}{T}$

c) The actual value for the chosen element (road length, turning angle degree, number of branches for the intersections, etc.) is sampled from a normal distribution over the data in the cluster.

i.e. The average value for each image in the cluster is extracted, then we compute the mean and variance on these list of values which are then used to create a normal distribution random variable.

d) Apply the three steps above recursively until some stopping condition is satisfied (i.e., the recursion depth is equal to an empirically chosen value X)

We basically applied a randomized Lindenmeyer System.

After the first try, we got results similar to this:

Figure 4. 11

We can see that the obtained network is a total mess, mostly due to self-intersecting roads.

After constraining the newly generated elements to prevent self-intersection, we obtain results similar to this:

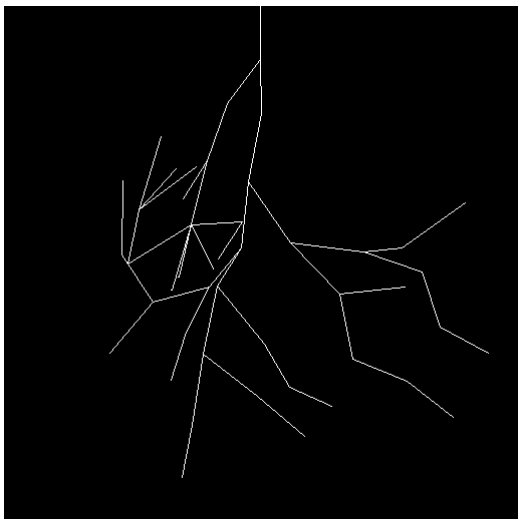

Figure 4.12

Definitely an improvement but not very realistic yet. We can observe that the roads are concentrated in a small area and also some roads are too close to each other.

After some additional constraints regarding the angles between the branches of an intersection, we have much better results like:

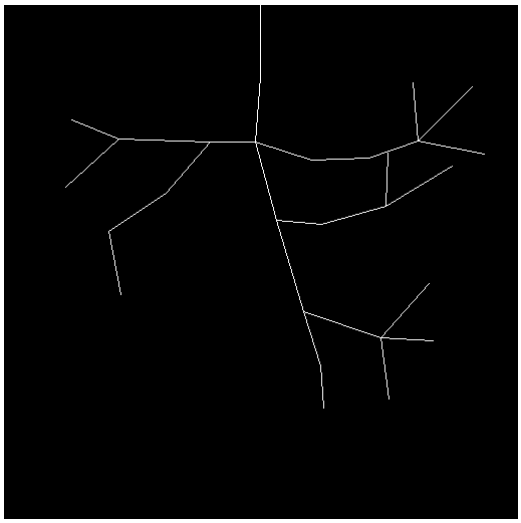

Figure 4.13

We managed to generate decent road network for one patch of 512x512. But on this approach, we used just one starting point from which we began the generation procedure. We also generalized this to any number of starting points from which we will start generating road. Here we can see a sample with three starting points:

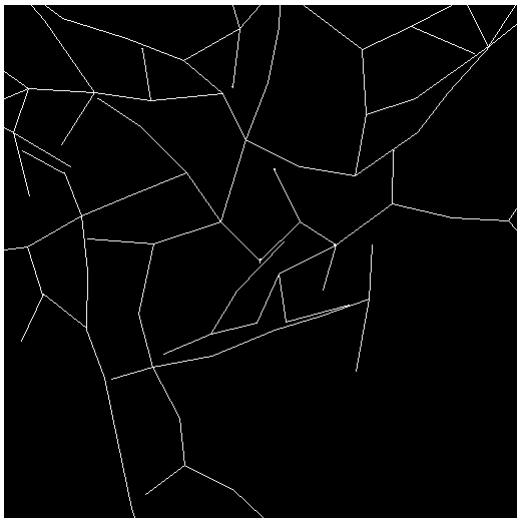

Figure 4.14

We went even further and generated an entire 3x3 grid. The roads between the grid tiles are interconnected (note: the white lines between the tiles are not generated, they are manually added to view the tiles separated):

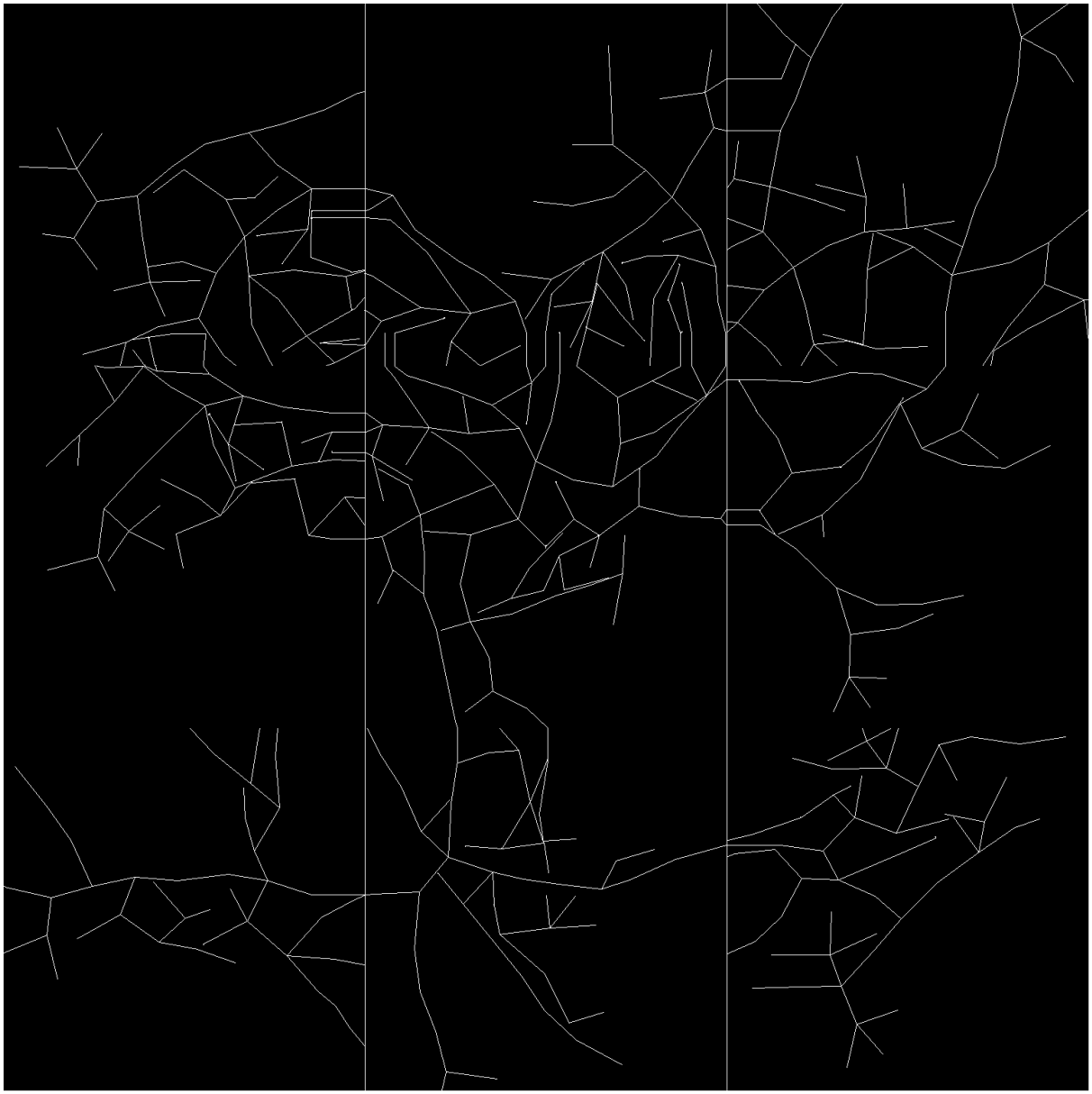

Figure 4.15

To achieve this, we used the following strategy:

- We created an empty 3x3 grid (9 tiles)
- Each tile has:
    - Starting points: the points from where we will generate road
    - Ending points: the points that lay on the edge of the tile
    - Current edges: the edges that define the roads for that tile

- After generating the first tile, we transform the end points of the current tile into starting points for the neighbor tiles
- We repeat the previous step until we completed entire grid

Even more, let's consider the scenario where we have a moving agent that starts from a certain point and then walks along the generated roads. We made possible for it to walk without ending, using the following strategy: depending on what direction he goes, we delete an entire row or column of tiles and generate new one, bound to the already existing ones.

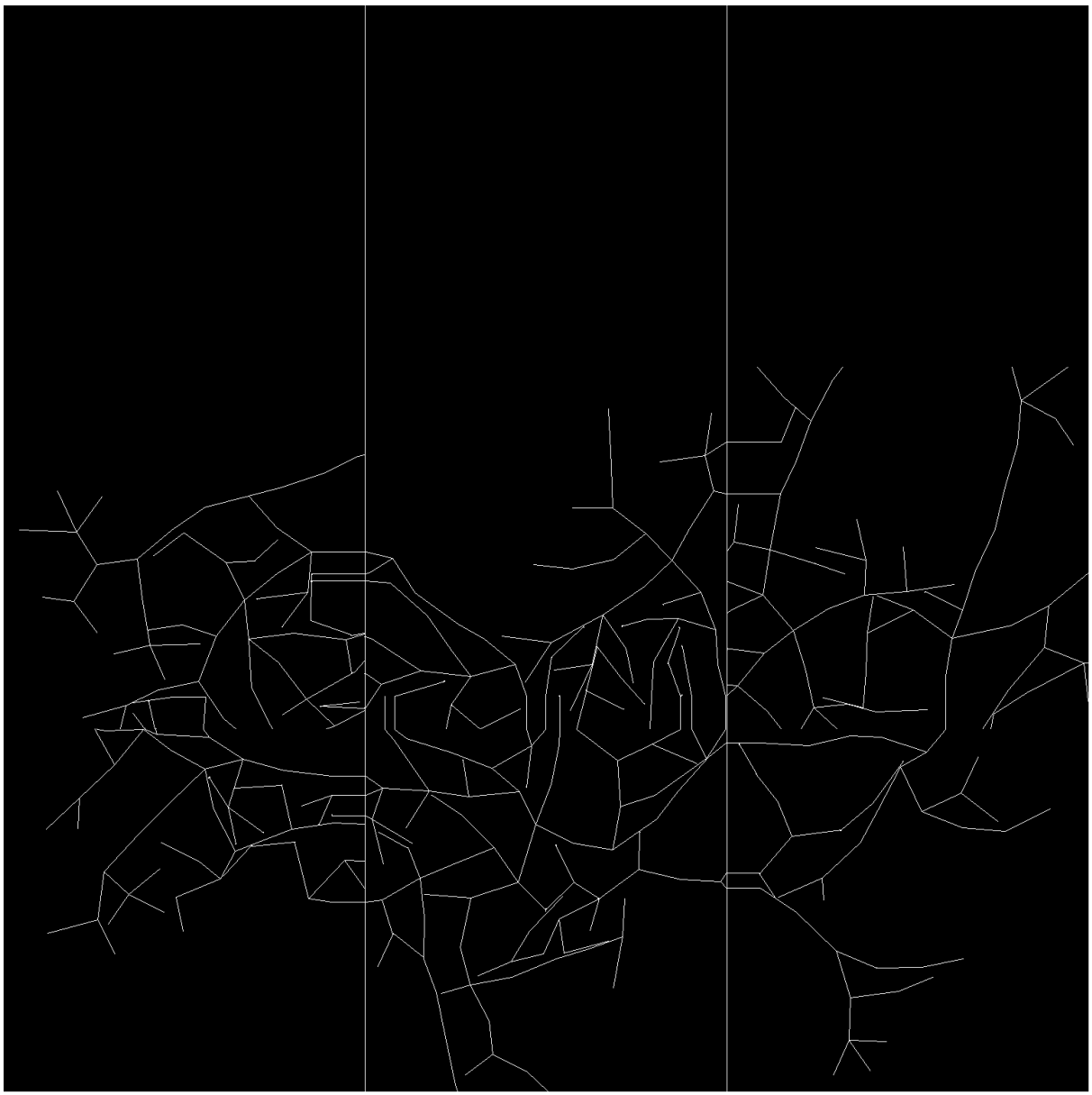

Figure 4.16

We can observe a shift to bottom (old bottom disappears, old middle becomes new bottom, old top becomes new middle and the new middle is generated empty). Now we will show the newly generated road on the new tiles.

Figure 4.17

We observe that only the top right corner was generated. That's because it was the only new tile that was neighboring edge points from another tile.

Using this approach, we can dynamically generate roads infinitely.

To speed up the generation process, we used a multi-process approach, each tile being processed independently. We present in the table below the total run time for generating sequentially and in parallel different numbers of tiles (from 1 to 9).

Note: A multi-threaded approach, although correct in theory, couldn't be applied due to python language's GIL (Global Interpreter Lock).

Number of tiles		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Sequential time (sec)	*Run #1*	1.75	3.60	8.32	9.33	13.81	19.86	21.69	16.24	18.18
	Run #2	1.66	4.87	6.76	11.20	15.83	11.77	17.30	17.77	23.40
	Run #3	2.43	3.75	8.37	11.34	9.41	12.67	20.65	22.77	26.69
	Run #4	2.71	4.62	8.70	11.15	13.07	17.83	14.97	24.45	25.73
	Run #5	1.60	5.34	7.28	9.02	11.95	16.86	16.46	25.83	23.10
	Average	2.03	4.43	7.88	10.40	12.81	15.79	18.21	21.41	23.42
Multi-threaded time (sec)	*Run #1*	2.12	4.65	5.49	10.66	7.14	16.52	10.43	13.93	14.65
	Run #2	2.00	2.21	4.79	9.61	11.78	9.04	14.99	18.84	10.28
	Run #3	2.58	4.87	6.92	5.83	12.61	13.61	12.04	9.35	11.48
	Run #4	2.74	2.92	6.51	8.75	9.22	8.63	9.83	16.38	20.81
	Run #5	1.30	3.44	7.96	6.24	8.73	11.35	17.62	11.46	15.85
	Average	2.14	3.61	6.33	8.21	9.89	11.83	12.98	13.99	14.61

Table 4. 1

Observations:

- The time required to generate a tile (512 x 512) is somewhere around 2 seconds.
- The time required to generate a tile and the whole grid varies according to the complexity of the road network (number of features, especially intersections).
- The average required time for generation is lower for the multi-process approach than for the sequential approach.
- The more tiles we have to generate at a moment of time, the greater the difference between the average time for sequential approach and the parallel approach.

The second approach we will be trying for this task is based on Deep Reinforcement Learning. We will train an agent to generate such road networks.

The agent

We will use a Double Deep Q-Network (abbr. DDQN) powered agent, implementation of the "tensorflow agents" framework along with Adam optimizer and element wise squared loss as the loss function.

The set of actions that the agent may perform is similar to what we used in the previous approach with Lindenmayer Systems: the agent will have to choose at each step between creating a straight road segment of a certain length (DIST_<value>, $value \in [0, 210)$), creating an angle of a certain number of degrees (ANGLE_<value>, $value \in [-90, 90]$), creating an intersection with a certain number of branches (INTER_<value>, $value \in \{3, 4, 5, 6, 7\}$) and creating a road end (END). The actions are coded in the following way:

Let a be in action, $a \in [0, 44], where$:

- If $a \in [0, 20]$ then a is of type DIST with $value = a * 10 + rand_{between(0,9)}$
- If $a \in [21, 38]$ then a is of type ANGLE with $value = \left((a - 21) * 10 + rand_{between(0,9)}\right) - 90$
- If $a \in [39, 43]$ then a is of type INTER with $value = (a - 39) + 3$
- If $a = 44$ then a is of type END

The environment

The environment is defined by a state made out of 23 parameters:

- Number of: segments, angles, road ends, intersections (and also for each kind separately), marginal nodes, road self-intersections, nodes out of the field
- Min, Max, Average value of angle degrees and segments length (+ total sum of segments' lengths)
- Length of last segment, degrees of last angle, branches of last intersection and last action taken

We won't explicitly describe the reward functions but instead we will explain the rules that governate the environment:

We reward generation of an object adjusted with the value it was generated with. i.e.,

- We encourage segments of medium value in the detriment of very long or short ones.
- We penalize very close angles (less than 30 degrees).
- We penalize new segments if they get out of bounds or self-intersect the existing road.
- We encourage road ends a bit less than other elements in order for the road to not end that fast.
- We encourage the creation of a new segment after an angle.
- We encourage the creation of a new angle after a new intersection.
- We penalize generating same type of element in row.

We can observe the results of the first trials in Figure 4.18 (left and right):

Figure 4.18

The generated road networks are extremely simple, meaning the environment needs better reward function and observations also the agent may need another set of hyper parameters. The average reward amount can be observed in figure X.

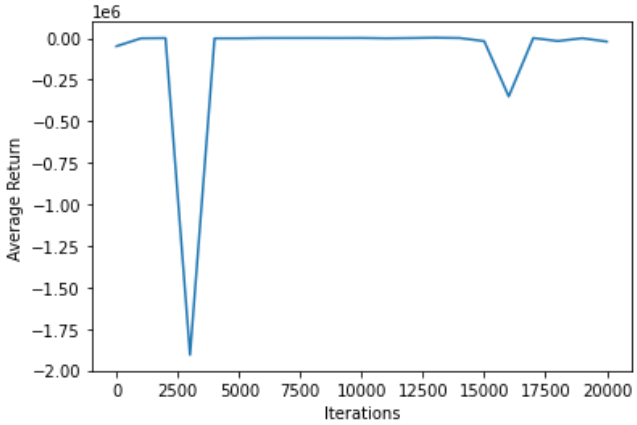

Figure 4.19

Chapter 5

Conclusions

5.1 Outlook

Throughout our work we proposed two new ways of generating dynamic road networks maps. The motivation of our work lays into the growth of industries like gaming, self-driving vehicles and city building for the continuously growing population. We consider that our work presented in this paper may help, in this, or in other form to overcome the requirements for the aforementioned tasks. Another motivational fact was the lack of such material (as we outlined in the "Related Work" chapter) and even if some exist, they are very hard to use, to improve, to change, to adapt for ones needs.

The approach presented shows us that using Lindenmayer Systems it is possible to achieve the task of dynamically generating road networks as it can be seen in figures X and Y. It's true that the generated maps don't resemble the dataset maps very well but even though, the generated maps are integrating the same elements that define the real road networks maps and can be used for practical applications like generating video games tracks for example.

On the other hand, an approach based on Reinforcement Learning has the potential to yield good results although it is very time and computation consuming in terms of training and finding optimal parameters in terms of reward function, learning rate, and other hyper parameters used during the training process.

The two approaches combined with the new dataset we created through computer vision may lead to even better and sophisticated approaches like Generative Adversarial Immitation Learning.

5.2 Improvements and future work

This procedure is open to a variety of further improvements and new features:

1. First of all, the realism of the generated road networks may be improved to assemble more and more a real-life road network.
2. Another improvement would consist of generating additional elements to the road, like buildings, green spaces, etc. and even more, add more specific buildings like hospitals, schools, malls, stadiums, etc. This feature would add additional constraints to the generation process making the final result more realistic.
3. Some other improvement would be to define an orientation for each road and also different number of lanes for each road. At the moment we support only generic roads, but creating different types of roads would be a nice feature.
4. Another nice feature would be to pre-define some elements of road and then to let the algorithm generate the roads taking in consideration those already existing elements. This would make the generation process more interactive and more convenient regarding the needs of the user.
5. We may use our reinforcement learning agent as a generator in a Generative Adversarial Imitation Learning (abbr. GAIL) system. This approach may yield better results as the RL agent will also receive feedback from a discriminator, enhancing its performance.
6. Finally, some better rendering options may be used to offer different angles, perspectives, 3D variant (implying adding height to roads as well), etc. This can be achieved with tools like Unity.

We can observe that the project is still prone to many improvements. It is also very versatile considering the features that may be added, leading to a wider range of usage.

Bibliography:

[1] Ian Millington, “AI for Games, 3rd edition”, CRC Press, 2019

[2] Adrian Kaehler & Gary Bradski, “Learning OpenCV 3”, O'Reilly Media, Inc., 2016

[3] Radford, Alec & Metz, Luke & Chintala, Soumith. (2015). Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.

[4] Goodfellow, Ian. (2016). NIPS 2016 Tutorial: Generative Adversarial Networks.

[5] Xiaolong Wang, Abhinav Gupta, Generative Image Modeling using Style and Structure Adversarial Networks”, 2016

[6] de Araújo L.J.P., Grichshenko A., Pinheiro R.L., Saraiva R.D., Gimaeva S. (2020) “Map Generation and Balance in the Terra Mystica Board Game Using Particle Swarm and Local Search”. In: Tan Y., Shi Y., Tuba M. (eds) Advances in Swarm Intelligence. ICSI 2020

[7] Lara-Cabrera R., Cotta C., Fernández-Leiva A.J. (2013) “A Procedural Balanced Map Generator with Self-adaptive Complexity for the Real-Time Strategy Game Planet Wars.” In: Esparcia-Alcázar A.I. (eds) Applications of Evolutionary Computation. EvoApplications 2013

[8] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta and A. A. Bharath, "Generative Adversarial Networks: An Overview," in IEEE Signal Processing Magazine, vol. 35, no. 1, pp. 53-65, Jan. 2018

[9] Ping K., Dingli L. (2020) “Conditional Convolutional Generative Adversarial Networks Based Interactive Procedural Game Map Generation.” In: Arai K., Kapoor S., Bhatia R. (eds) Advances in Information and Communication. FICC 2020

[10] Lara-Cabrera, Raul & Cotta, Carlos & Fernández-Leiva, Antonio. (2012). Procedural Map Generation for a RTS Game. 13th International Conference on Intelligent Games and Simulation, GAME-ON 2012.

[11] Carli, Daniel & Bevilacqua, Fernando & Pozzer, Cesar & d'Ornellas, Marcos. (2011). “A Survey of Procedural Content Generation Techniques Suitable to Game Development”. Brazilian Symposium on Games and Digital Entertainment, SBGAMES. 26-35. 10.1109/SBGAMES.2011.15.

[12] S. Snodgrass and S. Ontañón, "Learning to Generate Video Game Maps Using Markov Models," in IEEE Transactions on Computational Intelligence and AI in Games, vol. 9, no. 4, pp. 410-422, Dec. 2017

[13] D. Perez, J. Togelius, S. Samothrakis, P. Rohlfshagen and S. M. Lucas, "Automated Map Generation for the Physical Traveling Salesman Problem," in IEEE Transactions on Evolutionary Computation, vol. 18, no. 5, pp. 708-720, Oct. 2014

[14] Kaiser, Pascal & Wegner, Jan & Lucchi, Aurelien & Jaggi, Martin & Hofmann, Thomas & Schindler, Konrad. (2017). Learning Aerial Image Segmentation From Online Maps. IEEE Transactions on Geoscience and Remote Sensing. PP. 1-15. 10.1109/TGRS.2017.2719738.

[15] T. Y. Zhang and C. Y. Suen. 1984. A fast parallel algorithm for thinning digital patterns. Commun. ACM 27, 3 (March 1984)

[16] Bojchevski, Aleksandar & Shchur, Oleksandr & Zuegner, Daniel & Günnemann, Stephan. (2018). NetGAN: Generating Graphs via Random Walks.

[17] Zhou, Dawei & Zheng, Lecheng & Xu, Jiejun & He, Jingrui. (2019). Misc-GAN: A Multi-scale Generative Model for Graphs. Frontiers in Big Data. 2. 3. 10.3389/fdata.2019.00003.

[18] Grasl, Thomas & Economou, Athanassios. (2011). GRAPE: Using graph grammars to implement shape grammars...21-28.

[19] Wang, Hongwei & Wang, Jia & Wang, Jialin & Zhao, Miao & Zhang, Weinan & Zhang, Fuzheng & Xie, Xing & Guo, Minyi. (2017). GraphGAN: Graph Representation Learning with Generative Adversarial Nets. IEEE Transactions on Knowledge and Data Engineering. PP. 10.1109/TKDE.2019.2961882.

[20] Kipf, Thomas & Welling, Max. (2016). Variational Graph Auto-Encoders.

[21] Linus Gisslén, Andy Eakins, Camilo Gordillo, Joakim Bergdahl, & Konrad Tollmar. (2021). Adversarial Reinforcement Learning for Procedural Content Generation.

[22] S. Risi and J. Togelius, “Increasing generality in machine learning through procedural content generation,” Nature Machine Intelligence, pp. 1–9, 2020.

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, & Martin Riedmiller. (2013). Playing Atari with Deep Reinforcement Learning.

[24] Jonathan Ho, & Stefano Ermon. (2016). Generative Adversarial Imitation Learning.

[25] A. Khalifa, P. Bontrager, S. Earle, and J. Togelius, “Pcgrl: Procedural content generation via reinforcement learning,” arXiv preprint arXiv:2001.09212, 2020.

[25] M. L. Littman and C. Szepesvári, “A generalized reinforcement-learning model: Convergence and applications,” in ICML, vol. 96. Citeseer, 1996, pp. 310–318.

[26] P. B. Silva and A. Coelho, “Procedural modeling of urban environments for digital games development,” in Proceedings of the 7th International Conference on Advances in Computer Entertainment Technology, ser. ACE ’10. New York, NY, USA: ACM, 2010.