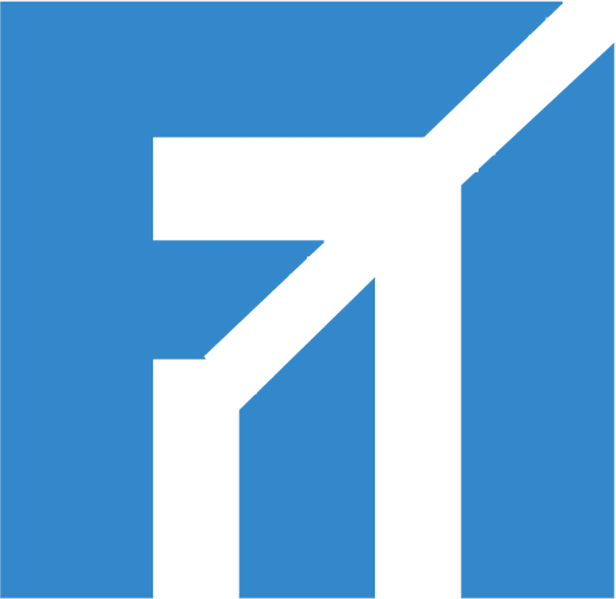
**“ALEXANDRU IOAN CUZA” UNIVERSITY OF IAȘI**

**FACULTY OF COMPUTER SCIENCE**



BACHELOR THESIS

**Application of K-means Clustering in Video Games**

Proposed by

**Andrei Ghiran**

**Session:** 06.2020

Scientific coordinator

**Prof.lect.dr. Moruz Alex**

**“ALEXANDRU IOAN CUZA” UNIVERSITY OF IAȘI**

**FACULTY OF COMPUTER SCIENCE**

**Application of K-means Clustering in Video Games**

**Andrei Ghiran**

**Session:** 06.2020

Scientific coordinator

**Prof.lect.dr. Moruz Alex**

SUMMARY

Introduction 3

1.Similar Applications 4

1.1. Deep learning agents 4

2.2. Computer vision-based agents 5

2. Implementation 7

2.1. AI adversary 7

2.1.1 The K-means Algorithm 7

2.1.2 The A-star Algorithm

3. User Manual

4. Conclusion and Future Goals

INTRODUCTION

From the day I started my studies at the faculty of computer science I knew I wanted to pursue a career in game developing. To be one step closer to my career of choice I decided that for my bachelor thesis I would combine my passion for game developing and the limitless applications of machine learning. Using the Unity engine I developed a simple game that showcases the K-means algorithm by using it to train an Artificial Intelligence adversary for the player.

The game has a simple premise, the player controls a blue square that can only move on a grid in the 4 cardinal directions and must navigate around walls and reach the goal, the AI adversary, represented by a red square, with the same movement constraints as the player and only moving at the same time as the player, must intercept and bump into them to end the game. The adversary doesn’t know the position of the player at all times, he has a vision radius in witch he can detect the player, acquire his position and save it in a file for later use. The K-means algorithm uses the saved player locations to form clusters and determine an advantageous position from where the adversary can intercept the player, the using the A-star algorithm the adversary will find a path to this more advantageous position and follow it.

**1.SIMILAR APPLICATIONS**

Information on the use of machine learning techniques in video games is mostly known known publicly through research projects as most gaming companies don’t publish specific information about their intellectual property. The most common use of clustering algorithms in the domain of video games is the classification of player behaviour in massive multiplayer online games (MMOs). These behaviours range from completing missions to exploring the game wold and even spending time with other payers. The classification is used by the developers to understand what most players enjoy doing the game and this knowledge influences the further development of the game. K-means clustering is also commonly used for procedural content generation.

The most popular application of machine learning in games is the use of deep learning agents that compete with professional human players. Another popular type of agent is the computer vision-based AI player. The most significant application of machine learning has been done on games such as Dota 2, the StarCraft series, Pong and Doom. Games that did not originate as video games such as Chess and Go have also been subject to machine learning research.

**1.1. Deep Learning Agents**

Deep learning focuses heavily on the use of artificial neural networks that learn to solve complex tasks. Deep learning uses multiple layers of artificial neural networks and other techniques to progressively extract information from an input. Due to this complex layered approach, deep learning models often require powerful machines to train and run on.

**Chess** is considered a difficult AI problem due to the computational complexity of its board space, its state space being 10^120 possible board states. Similar strategy games are often solved with some form of a Minimax Tree Search. These AI agents have been able to beat professional human players, such as the historic 1997 Deep Blue versus Garry Kasparov match. Since then, machine learning agents have only shown even greater success.

**Go** poses an even more difficult AI problem then chess. The state space of Go is approximately 10^170 possible board states, much greater then the state space of chess.

Google’s 2015 AlphaGo was the first AI agent to beat a professional Go player. It used a deep learning model to train the weights of a Monte Carlo tree search. The deep learning model consisted of 2 Artificial Neural Networks, one network to predict the probability of potential moves made by the opponent and the other network to predict the win chance of a given state. The combination of the deep learning model and the Monte Carlo tree search allowed the agent to explore the state tree much more efficiently than if it was using just the Monte Carlo tree search.

AlphaGo was initially trained on games against human players and later on games against itself. Later, in 2017, another implementation of AlphaGo, AlphaGoZero, was made public. This implementation was able to entirely train by playing against itself and it would train much faster then its predecessor. In the final stages of its training AlphaGoZero would develop new advanced strategies that, until then, were never used by a Go player.

**The StarCraft series** is a series of real-time strategy video games that have become very popular environments for AI research. Blizzard, the game’s developer, and DeepMind have collaborated to release a public StarCraft 2 environment for AI research.

Alphastar was the first AI agent to beat a professional StarCraft 2 player without any in-game advantages. Initially for its deep learning network the agent used a simplified zoomed out version of the gamestate as imput, but later it was updated to play using a camera like other human players. Up to the date of writing this document the code or architecture of the model have not been publicly released by the developers of Alphastar, but they have listed several machine learning techniques that they used, such as relational deep reinforcement learning and long short-term memory.

Initially Alphastar was trained with supervised learning, it watched replays of many human games in order to learn basic strategies, later it trained against different versions of itself an was improved trough reinforcement learning. The final version was successful but it was only trained to play on a specific map and matchup.

**Dota 2** is a multiplayer online battle arena (MOBA) game. Because of its complexity traditional AI agents have not been able to play at the same level as professional human players. The only widely published information on AI agents attempted on Dota 2 is OpenAI’s deep learning Five agent.

OpenAI Five utilized separate Long Shor-Term Memory networks to learn each hero in the game. It trained using a reinforcement learning technique known as Proximal Policy Learning. It trained against itself on a system containing 256 GPUs and 128,000 CPU cores accumulating 180 years of game experience each day. OpenAI Five's first public appearance occurred in 2017, it won a one-on-one game against a professional Dota 2 player known as Dendi. The following year the agent advanced so much that it could performe as a team and in a 2019 series of games beat the 2018 Dota 2 champion team.

**1.2. Computer vision-based agents**

Computer vision focuses on training agents to gain a high-level understanding on digital images and videos. Many computer vison techniques incorporate forms of machine learning and have been applied to various video games. This technique focuses on interpreting game events using visual data, in some cases AI agents, using model-free techniques, have learned to play games without any direct connection to internal game logic, solely using video data as input.

**Pong** is a table tennis sports game featuring simple two-dimensional graphics, manufactured by Atari and originally released in 1972. Andrej Karpathy, Tesla's director of artificial intelligence, demonstrated that a neural network with just one hidden layer is capable of being trained to play Pong based solely on screen data.

**Doom** (1993) is a first person shooter (FPS) game. A group of student researchers from Carnegie Mellon University used computer vision techniques to create an agent that could play the game using only image pixel input from the game. The students used convolutional neural network layers to interpret image data and output valid information to a recurrent neural network which was responsible for outputting game moves.

Other uses of machine learning in video games focus more on generating procedural content rather than AI agents that interact with the player. Procedural content generation is the process of generating data algorithmically rather than manually. Examples of procedurally generated content can be found in the weapons of Borderlands 2, the map layouts of Minecraft and the entire universes in No Man’s Sky. Common approaches to procedural generation include techniques that involve grammars, search-based algorithms, and logic programing. These approaches require a human to manually the range of content possible, but machine learning is theoretically capable of learning the features of possible content when given examples to train on, thus reducing the time spent on defining the range of desired content. Machine learning techniques used for content generation include Long Short-Term Memory, Recurrent Neural Networks, Generative Adversarial Networks and K-means clustering.

**Galactic Arms Race** is a space shooter video game that uses neuro-evolution powered procedural content generation to generate unique weapons for the player, based on his personal preferences. This game was a finalist in the 2010 Indie Game Challenge and its related research paper won the Best Paper Award at the 2009 IEEE Conference on Computational Intelligence and Games. The content was generated using a form of neuro-evolution called cgNEAT.

Machine learning is not a new addition to the domain of video games, but that does not mean there is no room for new and innovative application of its techniques. Over the years these techniques have continued to evolve and will continue to do so with the development of new video games and technologies.

**2. IMPLEMENTATION**

The project is developed with the Unity game engine, which is a cross-platform platform game engine developed by Unity Technologies. The engine can be used to create three-dimensional, two-dimensional, virtual reality and augmented reality games as well as simulations, animations and other experiences. The programing language associated with the unity engine is C#, all the scripts comprising the project are written in this programing language.

The project can be divided in three integral components: the AI adversary, the player characters and the environment.

**2.1. AI adversary**

In the editor the AI agent game object is comprised of a red square sprite, aRigidbody2D, two BoxCollider2D components and the PlayerDetector and Navigator scripts. One of the colliders together with the rigid body component are responsible for the correct interaction between the AI and the walls, they assure that the collision actually happens. The second collider has the is Trigger property checked and acts as the vision field of the AI.

The core components of the AI are the K-means clustering algorithm, found in the PlayerDetector.cs script, for determining an advantageous position and the A-star algorithm, found in the Navigator.cs script, for calculating a path to that position.

**2.1.1. The K-means clustering algorithm**

Clustering is used for partitioning a set of objects into groups, called clusters. Objects that are similar, according to a certain similarity measure, are places in the same cluster, while dissimilar objects are placed in different groups.

The K-means algorithm aims to partition n objects , in k () clusters , in witch every object belongs to the nearest cluster center. The most common algorithm uses an iterative refinement technique.

Given an initial set of k means , represented by centroids corresponding to each clusters, the algorithm alternates between two steps:

**Assignment:** The algorithm assigns each object to the cluster with the nearest mean, that with the minimum Euclidean distance between the object and the centroid corresponding to that cluster. If an object has the same minimum Euclidean distance to multiple centroids, it will be randomly assigned to one of the cluster that correspond to those centroids.

, where t represents the current iteration.

**Update:** Recalculate the means (the position of the centroids) for the new clusters.

The algorithm stops when, after 2 consecutive iterations the assignments no longer change. The algorithm does not guarantee to find the global optimum partitioning, the result may depend on the initial clusters. As the algorithm is usually fast, it is possible to run it multiple times and obtain different clusters.

The algorithm is most commonly used to assign objects to the nearest cluster by Euclidean distance. Using a different distance function may result in the algorithm never stopping.

Methods commonly used for initializing the clusters are **Forgy** and **Random Partition**. The Forgy method randomly choses k objects from the data set and uses these as the initial means. The Random Partition method first randomly partitions all the objects to one of the clusters and then proceeds to the update step, computing the initial means to be the centroids of the clusters randomly assigned points.

**Illustrated example of the K-means algorithm:**

Given a set of objects, represented by the squares in Figure 2.2.1.1**,** the k-means algorithm is used to partition them in k = 3 clusters.

**Step 1**: k initial means, represented by the colored circles are randomly placed within the data domain. (*Figure 2.2.1.1*);

**Step 2**: k clusters are created by assigning each object to the closest mean. (*Figure 2.1.1.2*);

**Step 3**: the cluster centroids become the new means (*Figure 2.2.1.3*)

**Step 4:** steps 2 and 3 are repeated until the cluster assignments don’t change anymore (*Figure 2.1.1.4*);

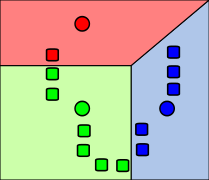
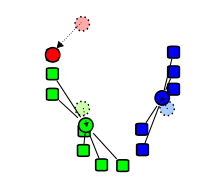
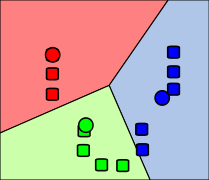
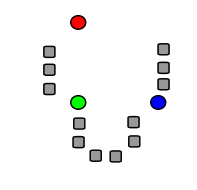
****

Figure 2.1.1.1

Figure 2.1.1.2

Figure 2.1.1.4

Figure 2.1.1.3

**Implementation of K-means clustering Algorithm**

In the project the k-means algorithm is used to partition the position where the AI adversary has encountered the player in previous sessions and calculate an advantageous position to were the AI can move to intercept the player. The code implementation is found in the PlayerDetector.cs script.

The coordinates of locations were the AI detects the player are stored in a text file named “points.txt” located in the project directory. To write and read from the file I used StreamWriter and StreamReader objects from the System.IO namespace. The coordinates are used to create Vector2 objects from the UnityEngine namespace, those store two-dimensional coordinates. To display points and cluster centers I used GameObjects also from the UnityEngine namespace.

For handling collections of coordinates and GameObjects I used the List object from System.Collections.Generic. Clusters being represented by a List object containing Lists of game objects.

To display a step by step execution of k-means I used the InvokeRepeating(string methodName, float time, float repeatRate) method from the UnityEngine. MonoBehaviour class. When called the method methodName will be invoked after time seconds and then repeatedly every repeatRate seconds. Calling CancelInvoke() will cancel all invoke calls.

The code has been structured in different methods that are called in order to do certain tasks. Next I will detail the methods and how they interact in order to implement the K-means algorithm.

* **OnTriggerEnter2D(Collider2D coll)** is called when the collider coll enters and stays inside the trigger collider attached to the AI, that acts like the vision field. The method checks if coll belongs to the player character, if that is the case the coordinates of the player character are added to the positions list;
* **initialize\_Cluster\_Centers\_and\_Points()** methoditerates trough the ols\_points list creates GameObjects with the coordinates of the points and finds the minimum and maximum values of the x and y. Then creates cl\_number cluster centers, assigns them a color, adds them to the cluster\_centers list and places them at random coordinates between the minimum and maximum x and y.
* **clusterize\_Points()** initializes the clusters list then iterates trough the points leaded from points.txt, computes the Euclidean distance between the point and the first cluster centerin the cluster\_centers list then iterates trough the rest of the list, computes the Euclidean distances and compares them. The point is added to the corresponding cluster in the clusters List and is assigned that clusters color.
* **move\_Cluster\_Centers()** iterates trough the clusters list, calculates the new coordinates for the cluster centers and counts the number of centers that have been moved. The new cluster center coordinates are computed by calculating the arithmetic means of the x an y coordinates for each cluster, the means become the new z and y coordinates of the cluster center. If the new center coordinates are different from the old ones the variable moved gets incremented by one;
* **SetNavGoalCoords()** calculates the centroid of the cluster centers, rounds the coordinates and adds or subtract 0.5 from x or y to make sure the coordinates land on the movement grid, and then moves the NavigationGoal game object at those coordonates;
* **K\_means()** calls **clusterize\_Points()**, **move\_Cluster\_Centers()** in a loop and count the number of iterations. The loop ends when the number of moved centroids is 0. After the loop a message is constructed containing the number of iterations, this message is displayed on the screen if the application is in Developer Mode. Lastly **SetNavGoalCoords()** is called;
* **Save()** writes the coordinates stored in the positions list in points.txt;
* **Load()** reads the player coordinates from points.txt and place them in the old\_points list;
* **make\_Colors()** generates 25 color objects by using 3 for loops nested one into another, with counters going from 1 to 0 in steps of 0.5, it creates a Color objects with those counters as RGB values and adds those objects to the colors list;
* **toggle\_display\_Clusters()** iterates the points and cluster centers lists and makes the corresponding game objects visible;
* **Start()** is called once at the start of the application, the method initializes the list objects used in the script, calls **Load()**, **make\_Colors()**, **initialize\_Cluster\_Centers\_and\_Points()**, initializes the moved variable with the number of clusters, so that the k-means algorithm execute at leas one iteration,and, if the application is in Play Mode, calls **K\_means()**;
* **Update()** is called at the end of every frame and it is used for handling keyboard inputs and checking to see if the cluster centers have not been moved and call the CancelInvoke() method;
* **OnTriggerStay2D(Collider2D coll)** similar to **OnTriggerEnter2D(Collider2D coll)** is called when another collider **coll** enters the trigger collider of the AI, but it is also called as long as coll is inside the AI trigger collider. The method checks if coll belongs to the player and moves the navigation goal game object to the players coordonates.
* **OnApplicationQuit()** is called when the application is closed, inside this method there is only a call of **Save()**.

**2.1.2 The A-star Algorithm**

The A-star(A\*) algorithm is a graph traversal and path finding algorithm. It was published by Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute in 1968. It can be seen as an extension of Edsger Dijkstra’s 1959. A\* guides its search using heuristics to achieve a better performance that its predecessor.

A\* is applied on weighted graph, starting at a specific starting node, its goal is to find a path to a given destination node, having the smallest cost, this cost can represent the distance traveled, the time traveled, etc. To find this path the algorithm creates a tree of paths with the start node as its root and extends those paths one at a time until it reaches thew destination node. A\* selects the path that minimizes where is the next node on the path, is the cost of the path from the start node to and is an additional heuristic that guides the algorithm by estimating the cost of the cheapest path from to the destination. As long as the heuristic function never overestimates the actual cost to get to the goal A\* is guaranteed to find the cheapest path from start to destination. A good heuristic is the straight-line distance between and the destination.

The algorithm most commonly uses a priority queue in which the nodes with the lowest have priority. At each step of A\* the top node is popped from the queue, the f and g values of its neighbors are updated and they are added to the queue. The algorithm until the destination node is popped from the queue or the queue is empty. To get the sequence of nodes that lead to the destination each node in the path keeps track of its predecessor, when A\* stops the destination node will point to the previous node, and so one, until a node’s predecessor will be the start node.

**3. USER MANUAL**

After starting the application you must chose one of two modes, Play Mode or Developer Mode, by pressing one of the corresponding buttons.

**Play Mode** is intended to exhibit how the game would behave in a normal play session. All the computing is done in the background. The player controls the blue square and the goal is to reach the green area by navigating the grey maze and avoiding the AI adversary, represented by the red square.

**Controls:**

* Use the arrow keys to move the player character (blue square). You can only move in a single direction at a time.
* Use the “Escape” key to close the application.

**Developer Mode** is intended to exhibit the internal mechanics of the game. When the session starts in developer mode the AI does not run the k-means algorithm right away and it will not do so until a specific key is pressed. In this mode there is a list of important keyboard inputs.

**Controls:**

* Use the arrow keys to move the player character (blue square). You can only move in a single direction at a time.
* Press “D” to display a graphical representation of useful information.
  + The white points represent all the positions recorded and used by the k-means algorithm;
  + The colored triangles represent the means of the clusters;
  + The dark red diamond represents the AI’s navigation goal;
  + The small red squares represent the path calculated by the A-star algorithm;
  + The grey, transparent square represents the AI vision field;
* Press “I” to execute one iteration of the K-means algorithm. The points, if displayed on the screen, will change color according to the cluster they have been assigned to and the means will move to the centroids of the corresponding clusters.
* Press “K” to start running the K-means algorithm, one iteration will be executed every half a second and when the algorithm stops a text with information regarding he number of iterations will appear at the bottom of the screen.
* Use the “Escape” key to close the application.

When the player enters the goal area or the AI touches the player character the game will stop and a message will appear on the screen displaying the result of the game.

**CONCLUSION**