# MAPF Project

Yonatan Shelach

Tom Seligman

Jessica Leibovich

# Table of contents

# Background

## Challenge representation

1. Definitions:
   - $S = \{s_1, s_2, \dots, s_{N_s}\} \subset \mathbb{R}^2$ a final set of $N_s$ source points
   - $D = \{d_1, d_2, \dots, d_{N_d}\} \subset \mathbb{R}^2$ a final set of $N_d$ destination points
   - $\mathbb{T} = \{0,1,2, \dots\}$ discretization of the time
   - A path $p = p(s,d)$ between a source point $s \in S$ and a destination point $d \in D$ is a function $p: \mathbb{T} \to \mathbb{R}^2$ such that
     - $p(0) = s$
     - There exists $t_s \in \mathbb{T}$ such that $p(t) = s$ for all $t \leq t_s$
     - There exists $t_d \in \mathbb{T}$ such that $p(t) = d$ for all $t \geq t_d$
     - The velocity is fixed, i.e., for $t_s < t < t_d$ we have $\|p(t+1) - p(t)\|_2 = 1$
     - There exists a fixed time $t_m \in \mathbb{T}$ such that $t_s \leq t_m$

     For such a path we say that it starts at time $t_s$.

   - Let $W$ be our warehouse with $N_s$ source points and $N_d$ destination points. The locations in the warehouse are represented by a matrix $B^{m \times n}$ such that $m \geq \max\{N_s, N_d\}$.
   - An obstacle $o \in \mathbb{R}^2$ is a coordinate in the warehouse (i.e., an entry in the matrix), in which an agent is not allowed to pass through.
   - Let $A = \{A_1, A_2, \dots, A_{N_a}\}$ be our agents that can go in a fixed pace in the warehouse.
   - Let $R = \{R_1, R_2, \dots, R_{N_a}\} \subseteq S \times D \times \mathbb{T}$ be our routing requests. Where the last coordinate represent the destination time of the routing request.

2. Challenge:

   With a given warehouse $W$ and routing requests $R$ the challenge is to find a plan of $|R|$ paths for $|R|$ agents that will satisfy the routing requests while the following holds:

   1. There are no collisions between paths of different agents. (no two agents are in the same position at the same time and no swapping positions between two agents).
   2. There is no path in the plan which goes through an obstacle's position.
   3. For every path of an agent there is exactly one matching routing request.

   Such a plan will be called a solution.

   The goal is to find a solution in minimal computational time.

The purpose of our project is to solve this problem, using the Transformer model.

As part of building the training data for our transformer model, we managed to use the LNS2 algorithm in a way that allowed us to create high-quality data for training that improves the results of our model.

## Transformer background

Transformer is an NLP model that rely on the new idea that is called "attention", which was first introduced in the paper "Attention Is All You Need".

**Attention model intuition:**

The assumption behind the attention model is that each element of the output is determined mainly by a subset of elements in the input, and we recognize those elements as part of the same context.

For example, if our input is the image below, then if a pixel of the picture is relevant for a specific feature then most likely the pixels similar to it are also relevant. In this picture if we want to determine whether or not the image contains a lens, then the fact that the pixels of the lens are similar to each other can help us identify whether or not there is a lens by focusing only on those pixels that we consider as having the same context.

The idea of the attention model is to assess which elements in the input have the same context and to make our network to calculate each element of the output based on those elements of the input, unlike other models that are looking at the entire input.

In practice, attention is a weight function that gives for each element in the input and each element in the output, a value for how likely it is that those elements belong to the same context.

The way we assess if two elements belong to the same context is by a function called similarity which is tuned as part of the learning process.

# Architecture

What we have accomplished in this part is to create the offline route database that will be used by a preprocessing pipeline for our transformer. Also, we managed to encode routes which are represented by an array of coordinates to a short string of numbers. Those encoded routes were put into the tokenizer. We implemented a classic transformer based on the paper "Attention Is All You Need".

At the end of this section, we present the problem with our approach and in the next part we suggest another approach, which is based on the paper "Subdimensional Expansion Using Attention-Based Learning For Multi-Agent Path Finding".

## Offline route database

The code for this section is located in $mapftransformer/database/routes.py$.

The offline database contains routes ordered by routing requests for fast filtering. The routes are generated using the mid-point algorithm which was developed by the previous semester's team. This database allows us to limit our solution's domain for faster calculation time and allows the transition from routing requests to paths in our sequence-to-sequence model.

The database is divided into folders by the y's coordinate in the source and the destination. So, based on that, a route from (9,2) to (0,4) will be in the folder named "src_2_dst_4". Inside each folder, there are the CSV files. Each file contains routes of a specific length and can be identified by the name of the file e.g., the file "path_length_13.csv" represents the routes by length 13. Each route is a list of coordinates inside the defined warehouse that goes from the source to the destination.

In this way, pulling out the relevant routes for a specific request is made easily and quickly. The database creation process needs to be applied offline to ease the online use of the transformer.

## Tokenizer and route encoding

The code for this section is located in $mapftransformer/preprocess/preprocess.py$.

In the context of NLP, a word in our model is a route, hence a sentence is a combination of routes. The goal is to find a translation for a combination of routes to a valid plan, as we defined the solution.

A valid route $r$ is a set of points $\{(y_i, x_i)\}_{i=0}^{n}$ where $n+1$ is the arrival time to the destination. Each route encoded to $(y_0, y_n)\_\langle\text{direction encoding}\rangle$ where the direction encoding works as the following logic:

- Every direction is encoded to an integer:
  - 0 – for staying in the same position
  - 1 – for going up
  - 2 – for going down
  - 3 – for going left
  - 4 – for going right
- In addition to that and to reduce the length of the encoding, the number of consecutive moves that are in the same direction were encoded in a two-digit form after each change of direction. For example, a section in the route with one stay in place, two steps right, and four steps down was encoded to 00140204.

To summarize, a route from $(x_0, y_0)$ to $(x_f, y_f)$ with the series of directions $d_1, d_2, \dots, d_n$ and the corresponding number of steps of each direction $c_1, c_2, \dots, c_n$, where each $c_i$ is in two digits form was encoded to:

$$(y_0, y_f)\_d_1c_1d_2c_2 \dots d_nc_n$$

With that, we can decode routes in any warehouse with a size of no more than 100 in each dimension. Bigger warehouses require a longer form representation of the number of steps in each direction (more than two digits).

The tokenizer we used reads the encodings of all the routes from the database we have generated and gives each route a distinct integer between 1 and the number of routes in the database.

## Loss function

The code for this section is located in $mapftransformer/transformer/loss.py$

The goal of our loss function is to ensure that it gives 0 if and only if there are no collisions in the plan of routes that we get from the transformer output and all the routing requests have been fulfilled perfectly, therefore the loss function was defined as:

$$\ell(P,R) = F_{collisions}(P) + w_\ell \cdot F_{misses}(P,R)$$

Where:

- $P$ is a plan of routes
- $R$ is a set of routing requests
- $F_{collisions}$ is a function that gives the number of collisions between a given set of routes. The collisions are from two types: vertex conflict – two routes that possess the same coordinates in the warehouse at the same time, and second type is swapping conflict – two routes that swap coordinates between them at the same time. For example, if the first route $r_1$ in time $t$ was in the coordinates $(x_1, y_1)$ and the second route $r_2$ was in time t in the coordinates $(x_2, y_2)$ and in time $t+1$ route $r_1$ was in coordinates $(x_2, y_2)$ and route $r_2$ at the same time was in $(x_1, y_1)$ we have a swapping conflict between the routes $r_1, r_2$.
- $F_{misses}$ is a function that counts how many routing requests were not fulfilled in the given set of routing requests according to the given plan.
- $w_\ell$ is a positive real number that indicates the weight of the outputs of each function.

Now we can say that if there are no collisions in a plan $P$, we get that $F_{collisions}(P) = 0$. Moreover, if all the routing requests of the set $R$ are fulfilled by a plan $P$ we get that $F_{misses}(P,R) = 0$ and therefore $\ell(P,R) = 0$, and it also works in the opposite direction because the outputs of those functions are non-negative and $w_\ell > 0$.

The goal of the training process is to minimize this function to 0. When the output of the loss function approaches zero, there are fewer collisions and the prediction answers the routing requests more precisely.

## The transformer

The code for this section is located in $mapftransformer/transformer$.

The architecture as can be seen in the figure below uses stacked self-attention and point-wise, fully connected layers for both the encoder and decoder. At each step the model is

auto-regressive, consuming the previously generated symbols as additional input when generating the next.

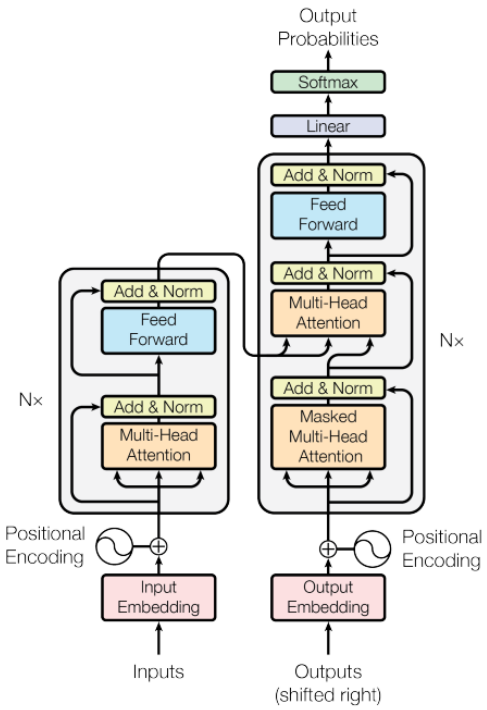
Now we will explain the following components of the transformer:

**Multi-head attention.** Instead of performing a single attention function with $d_{model}$ -dimensional keys, values and queries, we project the queries linearly: keys and values $h$ times with different, learned linear projections to $d_k$, $d_k$ and $d_v$ dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding $d_v$-dimensional output values. These are concatenated and once again projected and resulting in the final values.

Our attention function is based on binary similarity function which we called a coexistence function which for two paths returns 1 if they collide, otherwise returns 0.

**Encoder**. This unit is composed of a stack of $N$ identical layers which are divided into two sub-layers. The first one is the multi-head self-attention mechanism and the other is a position-wise fully connected feed-forward network. A residual connection is connected around each sub-layer and followed by a normalization layer.

**Decoder**. This unit is also composed of a stack of $N$ identical layers which contains the same layers as in the encoder with an additional third sub-layer that performs multi-head attention with masking to prevent dependency on the subsequent positions. In this sub-layer, similarly to the other sub-layers, a residual connection is applied followed by a normalization layer.




**Embedding**. An embedding is a mapping of a discrete-categorial variable to a vector of continuous numbers. Usually, this context is relevant for word embedding.

**Positional encoding.** Because the model contains no recurrence and no convolution the relative/absolute position needs to be taken into consideration, so this information is added to the encoding. This stage is important in NLP problems because in this way the model will treat the input as a sentence instead of a bag of words.

### The architecture problem

Our approach didn't work with the classic transformer model, because our data is not labeled, and therefore cannot be passed to the decoder of the transformer correctly. Even if our data was labeled, in our approach, the idea of labeling a single routing request to the correct route is impractical because it depends on the context of other routing requests in the same request, hence for different requests, a single request can be translated to different routes.

## Description of the LNS2 algorithm

The LNS2 algorithm is an algorithm that solves the Multi-Agent Path Finding problem in a novel way. The way this algorithm works is by guessing a set of paths that contains collisions, and then repeatedly selecting a subset of colliding agents and replans their paths to reduce the number of collisions until the paths become collision-free.

The LNS2 algorithm receives as input:

- a file named $warehouse.map$ which describes the warehouse.

The file includes parameters such as height and width and each position in the warehouse is represented by a $'.'$ or $'@'$ were $'.'$ represents a passable position and $'@'$ represents an obstacle.

- A file named $batch\_\{number\}.scen$ where {number} is the number of the routing requests batch. We run the LNS2 multiple times with different routing requests batches. Each batch consists of $K + 1$ lines where $K$ is the number of requests in a batch. Each request is represented by 9 numbers:
    1. Bucket number – designed to improve the performance of the LNS2 but we set all buckets to zero as this improvement is minuscule when considering the number of requests, we set in a batch.
    2. Name of the warehouse map file.
    3, 4. Dimensions of the warehouse.
    5,6,7,8. The coordinates of the starting position and the target position for the agent.

**Notice:**

Unlike in our MAPF problem, the LNS2 algorithm receives routing requests that don't include a constraint on the arrival time.

The output of the LNS2 algorithm is a file called $batch\_\{number\}$ corresponding to the input file with the same number. This file includes a lot of statistical information produced by the algorithm, and a solution for the routing requests in the LNS2 format which represents each step in the path with one position number, following the formula:

$$position = (warehouse\ width) \times (x\ coordinate) + (y\ coordinate)$$

## Using LNS2 to create our training data

The training data for our transformer model is consistent of batches of $K$ requests where each request is described by five numbers:

$x_{start}, y_{start}$ – the coordinates of a starting position for an agent where $y_{start}$ is always at the beginning of the warehouse.

$x_{end}, y_{end}$ – the coordinates of a target position for an agent where $y_{end}$ is always at the end of the warehouse.

$arrival\_time$ – The time in which the agent is required to arrive from position $(x_{start}, y_{start})$ to position $(x_{end}, y_{end})$.

We wanted to make sure that the value we set for $arrival\_time$ is reasonable when considering the starting and target positions.

We achieved so, this by setting the starting and target positions first in a random way and then finding a solution for the request without the time constraint using the LNS2 algorithm.

Then, we used the lengths of the paths in the LNS2 solution as the arrival time for our routing requests. Those, ensure that we possess a new batch of routing requests, in our format (with arrival time constraint), that has a solution (the same as the LNS2 solution).

Pros and cons of our approach:

The pros:

1. It supplies a sanity check to our model: if our model cant find a solution on our training data then we know the model is incorrect since we know there is a solution.
2. It gives a correct order of magnitude of the target time which allows beneficial training.

The cons:

1. It gives biased training data which can damage the accuracy of our model since our model is not exposed to an input that does not have a solution.

<u>The flow of the code for creating the training data is:</u>

1. Creating many batches of input files ($batch\_\{number\}.secen$) consisting of routing requests without time in the LNS2 format, produced with random values for: $x_{start}, x_{end}$ and the required values by our model for: $y_{start}, y_{end}$.

2. Running the LNS2 algorithm on all our batch files. We used multi-threading which significantly reduced the running time for this stage.

3. Reading and converting the solution in the LNS2 output files into solutions in our format which represents each step in the path by its coordinates.

4. Validating the solutions from the previews step – including validation of the paths and that there are no collisions in the given solution.

5. Creating the routing requests batches in our format based on the valid solutions that ensure that our routing requests batches are solvable.

6. The final output file with the training data is a CSV file called: $training\_data.csv$

Steps 1 and 2 are done by the file $lns\_io.py$.

Step 3 is done by the file $solution\_validator.py$.

Steps 3,5 and 6 are done by the file $training\_io.py$.

The parameters of the warehouse, and for the creation of the training data are all set and programmable in the file: $config.py$.

# Our suggested paper – "Attention-Based Learning For Multi-Agent Path Finding"

## Introduction

In this part we present a summary of a different approach to handle our MAPF problem.

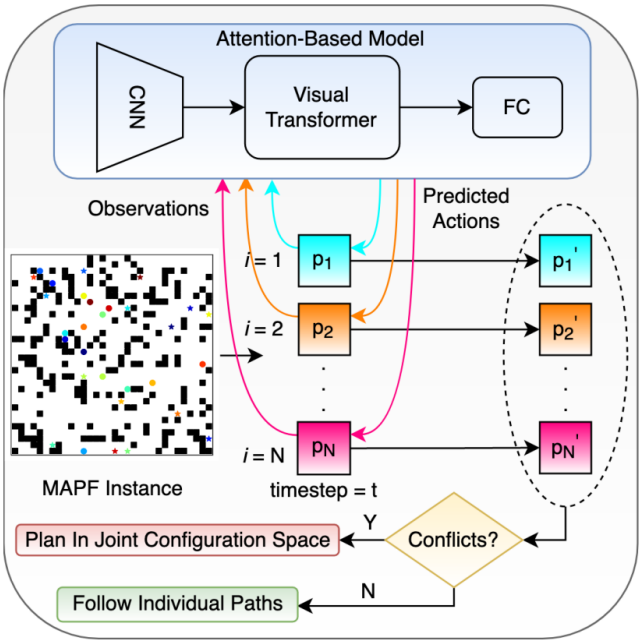
Multi-Agent Path Finding aims to find paths for a team of agents from their start point to their goal location. At first, the potential conflicts between agents are ignored; Then, agents either follow their plans or are coupled together for planning to resolve conflicts between them.

At every time step, each agent shares its observations with the attention-based model. The model predicts the action for each agent individually by attending to the map's structure and other agents' information.

The agents follow the predicted actions if there are no conflicts. Otherwise, the agents in conflict are coupled together by planning in their joint configuration space. The report shows that the attention-based model takes an agent's observation at each time step as input and outputs the predicted action of that agent. This model can pay attention to the structure of the map and other agents' information such as start, and goals to avoid potential conflicts.

The report explains about MAPF planner called LM* that begins by using the attention-based model to plan an individual path for each agent, and then couples agents together when needed by planning in their joint configuration spaces to resolve conflicts.

Transformers can help each agent to pay attention to the structure of the map and the subset of other agents that they must interact with to avoid conflicts.




## Problem description

The index set $I = \{1, 2, \dots, N\}$ denotes a set of $N$ agents. All agents move in a workspace represented as a finite graph $G = (V, E)$, where the vertex set $V$ represents the possible locations for agents and the edge set $E = V \times V$ denotes the set of all the possible actions that can move an agent between any two vertices in $V$. An edge between $u, v \in V$ is denoted as $(u, v) \in E$ and the cost of an edge $e \in E$ is a non-negative real number $\text{cost}(e)$. The $\pi^i(v_1^i, v_\ell^i)$ denote a path for agent $i$ that connects vertices $v_1^i$ and $v_\ell^i$ in the graph $G$. The $g^i\left(\pi^i(v_1^i, v_\ell^i)\right)$ denote the cost associated with the path. This path cost is the sum of the costs of all the edges present in the path. All agents share a global clock. Each action, either wait or move, requires one unit of time for any agent.

Any two agents $i, j \in I$ are claimed to be in conflict if one of the following two cases happens. The first case is a "vertex conflict" where two agents occupy the same vertex at the same time. The second case is an "edge conflict" where two agents travel through the same edge from opposite directions between times $t$ and $t + 1$ for some $t$.

The Multi-Agent Path Finding (MAPF) problem aims to compute conflict-free paths for all agents while the sum of path costs reaches the minimum. During the training phase, the model uses data generated by solving various MAPF instances using an expert MAPF planner. The model begins with several convolutional layers to extract the low-level features. The resulting output feature map then passes through a Visual Transformer (VT). The VT first uses a tokenizer to group pixels (of the feature map) into a small number of visual tokens; with each token representing a semantic concept in the image, Transformers are then applied to model relationships between these tokens. The attended visual tokens are then used as input to fully connected layers, which then output the predicted actions for agents.

In the report the learning-assisted M* (LM*) is presented. They consider graph $G$ to be a

four-connected grid in which agents are allowed to move in one of the four cardinal directions or to wait in place at each time step. Moving into an obstacle is considered to be an invalid move, and if an agent selects to move into an obstacle during testing, it instead waits in place for that time step. In practice, after training, agents rarely choose an invalid move, which indicates that they effectively learn the set of valid actions at each location.
The observation of an agent consists of ten channels, where each channel comprises a $32 \times 32$ size matrix. Obstacles: Channel 1 contains a binary matrix that represents the free space and obstacles in the grid graph. Specifically, entries corresponding to free spaces have value zeros while entries of obstacles have value ones. Channel 2 and 3 consist of two matrices describing the agent $i$'s start and goal respectively. In these matrices, the entry corresponding to the agent $i$'s start (or goal) has value one while all other entries have value zeros. We then generate another matrix (Channel 4) in which each entry has a value of the minimum cost-to-go from the corresponding location to the agent i's goal. These values are computed by running the Dijkstra search backward while ignoring any other agents. We scale this matrix so that all values lie between 0 and 1.
Channel 5 and 6 of $O_t^i$ are two binary matrices that represent the start and goal locations of all other agents, where the entries corresponding to any other agents' starts (or goals) have value ones while all other entries have value zeros. We also compute the cost-to-go for each agent $j$, sum up these cost values for each location and normalize. Visual Transformer comprises a static tokenizer and a transformer. Classification tokens and positional embeddings are used to enable attention-based learning within the transformer.
Channel 5, 6, 7 together provide the model with the context about where the other agents in the world are headed and what route they might take. In this work they use three binary matrices to provide the future position of other agents, one per time step. The Model Architecture comprises of 3 main components: (1) a convolutional neural network (CNN), (2) followed by a Visual Transformer to learn and relate higher-order semantic concepts, and (3) is fully connected layers for action classification. The output feature map from these convolutions is then passed through a Visual Transformer. VT uses a static tokenizer module to convert the feature map into a compact set of visual tokens.
The visual tokens are attended to using a transformer comprising 16 encoder layers each containing multi-headed attention modules with 16 attention heads and a multi-layer perceptron with a dimension of 512. During the training, they minimize the cross-entropy loss.
The results gave high training and validation accuracy indicating that the model is capable to select an optimal action for each agent during the planning process.

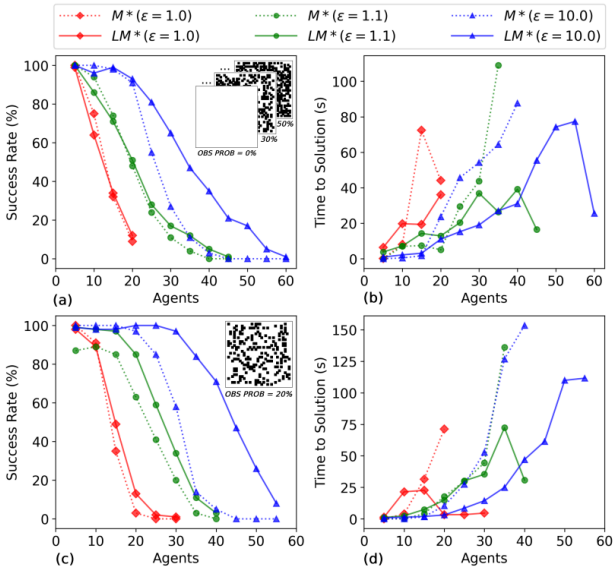
## Data generation

The training data is composed of $10K$ random maps of size $32 \times 32$. The maps include obstacles that are placed randomly. Each cell is being marked as an obstacle with a probability that is randomly selected from $\{0\%, 10\%, 20\%, 30\%, 40\%, 50\%\}$. The MAPF instances are generated according the maps with $N = 2,3, \ldots ,50$, where $N$ is the number of agents. Thus, in total they have generated $490K$ MAPF instances ($10K$ for each number of

agents). The start and goals are selected randomly for each agent and each test instance. To create labeled training data – observation of an agent at a certain time step and an action that should be selected by the agent at that time step, the ODrM* algorithm was used with an inflation of 1.1 with a time limit. It was able to solve approximately $295K$ instances. For each solved MAPF instance, the solution is a joint path $\pi = \{v_1, \dots, v_\ell\}$, where each $v_k = \{v_k^1, \dots, v_k^N\}, k = 1,2, \dots, \ell$ is a joint vertex that contains the locations of all agents. The generation of the training data is applied by selecting 30% of the joint vertices $v_k$ from $\pi$. Then, selecting 30% of the agents and their corresponding individual vertices $v_k^i$ from each $v_k$ and generating the observation $O_t^i$ of the agent at that time step and the action $A_t^i$ the agent takes in order to move to $v_{k+1}^i$. The generated dataset has size of 23 million that splits into the train set (90%) and test set (10%).

## Conclusion

This paper presents the Learning-assisted M* (LM*) MAPF planner which is composed of an attention-based learning part and the M* algorithm. The LM* begins with the attention-base model to plan for each agent and then couples agents together to plan in their joint configuration space to resolve conflicts. The results shows that LM* is capable to circumvent conflicts which yields higher success rates and shorter run time. The participants in the paper suggested to investigate this kind of fusion (attention-based learning and MAPF planners) with other MAPF planner, such as CBS. The results of this paper shows that the problem can be solve in a short run time. For example, a map of size $32 \times 32$, with 30 agents can be solved with a good success rate, in under 30 seconds.




## Our notes from the paper

This paper suggested a whole new different approach. The arriving time for each agent in our problem does not appear in this paper, and it must be handled. Our suggestion is to handle it in the phase of the cost computation channel, inside the observation, which is computed by Dijkstra algorithm. There are may be more gaps between the problems. Also,

the generation of the dataset will be a lot of work and would consume large amount of memory. The paper comes with a code (GitHub) and implemented in python (using the PyTorch framework), so it may be handy.

The main difference between the new approach to our approach is that they don't rely on a database of paths and instead the new model generate the paths. Another difference is that their model uses labeled training data in contrast to our approach.

## What can we take from this approach

In the new model they use the transformer to choose the next move for all the agents in a certain time step from a set of legal moves, in a way that reduces the chance for collisions. The success of this model gives us hope that we can manipulate it to choose paths from our database with reduced chance for collisions.