

**Hide &Seek**

Seminararbeit

*Programmieren 3*

von

Zhen Fan

Studiengang: Wirtschaftsinformatik

Matrikelnummer: 2246328

Zhicheng Xiang

Studiengang: Wirtschaftsinformatik

Matrikelnummer: 2255670

Junyi Zhu

Studiengang: Informationswirtschaft

Matrikelnummer: 2261592

Institut für Angewandte Informatik und Formale

Beschreibungsverfahren (AIFB)

KIT-Fakultät für Wirtschaftswissenschaften

Betreuer: Andreas Fritsch

Zweiter Betreuer: Martin Forell

Dritter Betreuer: Demian Frister

Eingereicht am: 28. Februar 2022

**Abstract**

This project is a program for the 'Hide and Seek' game by Vector

The program implements the traversal of an uncertain labyrinth and, at the end of the traversal, the robot, as Seeker, relies on the information obtained for logical path planning and recognition of another robot, based on the traversal information.

The labyrinth traversal relies on a depth-first algorithm, the path planning relies on the labyrinth traversal results (implemented by inflection points) and the recognition of the goal relies on Yolov5.

The project implemented the basic logic of the game in the final stage, fulfilling the requirements for target recognition.

# Table of contents

[Table of contents II](#_Toc96952095)

[List of figures III](#_Toc96952096)

[1. Introduction 1](#_Toc96952097)

[2. Object detection 1](#_Toc96952098)

[2.1 “FindContours” in OpenCV 1](#_Toc96952099)

[2.2 Yolov5 2](#_Toc96952100)

[2.2.1 Model training 2](#_Toc96952101)

[2.2.2 Use of model 2](#_Toc96952102)

[3. Traversing the Labyrinth 3](#_Toc96952103)

[3.1 Introduction of Labyrinth 3](#_Toc96952104)

[3.2 Concept for traversing 3](#_Toc96952105)

[3.3 Traversal Logic 4](#_Toc96952106)

[3.3.1 Right-hand rule 4](#_Toc96952107)

[3.3.2 Turning Point 5](#_Toc96952108)

[3.3.3 DFS 5](#_Toc96952109)

[4. Hide and Seek 6](#_Toc96952110)

[4.1 Seeker 6](#_Toc96952111)

[4.2 Hider 6](#_Toc96952112)

[5. Implementation 7](#_Toc96952113)

[6. Problem 9](#_Toc96952114)

[7. Conclusion 11](#_Toc96952115)

[References 12](#_Toc96952116)

[Erklärung 13](#_Toc96952117)

# List of figures

[Figure 1: Contour of Vector I 1](file:///C:\Users\Zhen_\Desktop\Seminararbeit_hide_and_seek2.docx#_Toc96952141)

[Figure 2 :Contour of Vector II 1](file:///C:\Users\Zhen_\Desktop\Seminararbeit_hide_and_seek2.docx#_Toc96952142)

[Figure 3: Contour of Vector III 1](file:///C:\Users\Zhen_\Desktop\Seminararbeit_hide_and_seek2.docx#_Toc96952143)

[Figure 4:Yolov5-Logo 2](file:///C:\Users\Zhen_\Desktop\Seminararbeit_hide_and_seek2.docx#_Toc96952144)

[Figure 5:Vector with recognition score in the camera 2](file:///C:\Users\Zhen_\Desktop\Seminararbeit_hide_and_seek2.docx#_Toc96952145)

[Figure 6:Absolute directions 4](file:///C:\Users\Zhen_\Desktop\Seminararbeit_hide_and_seek2.docx#_Toc96952146)

[Figure 7:Labyrinths virtualised with Pygame 4](file:///C:\Users\Zhen_\Desktop\Seminararbeit_hide_and_seek2.docx#_Toc96952147)

[Figure 8:Problems under the right-hand rule 5](file:///C:\Users\Zhen_\Desktop\Seminararbeit_hide_and_seek2.docx#_Toc96952148)

[Figure 9:Turning Points Scenarios 5](file:///C:\Users\Zhen_\Desktop\Seminararbeit_hide_and_seek2.docx#_Toc96952149)

[Figure 10:"Main" class flow chart 7](file:///C:\Users\Zhen_\Desktop\Seminararbeit_hide_and_seek2.docx#_Toc96952150)

[Figure 11:UML diagram of the “Node" class 7](file:///C:\Users\Zhen_\Desktop\Seminararbeit_hide_and_seek2.docx#_Toc96952151)

[Figure 12:Collision caused by distance sensor 10](file:///C:\Users\Zhen_\Desktop\Seminararbeit_hide_and_seek2.docx#_Toc96952152)

# Introduction

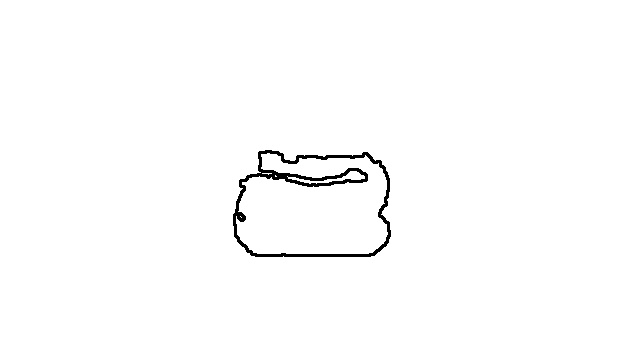
This project is based on a game of hide-and-seek between two robots. The robots in the game are Anki Vector.

The game involves an irregular labyrinth where one robot acts as seeker and the other as hider. The seeker needs to find the hider and the hider is hiding somewhere in the labyrinth.

The anki vector robot has a camera that can move in any given direction (coordinates) and its own api provides information about its position (coordinates, orientation, etc.). It also has a distance sensor that senses the distance of obstacles in front of it.

The project can be subdivided into three parts. The first part is the recognition of another Vector, where the knowledge of target recognition is involved. The second part is the traversal of an uncertain labyrinth, where the traditional algorithm of depth-first is mainly adopted. The third part is a hide-and-seek logic based on the labyrinth information available, which is used to play the game efficiently based on the information obtained.

# Object detection

This chapter describes the implementation of a Vector using a camera to scan and detect another Vector in the game of hide-and-seek. In this we have used two methods in succession and have finally chosen Yolov5 to implement this functionality.

## 2.1 “FindContours” in OpenCV

OpenCV (Open-Source Computer Vision Library: [http://opencv.org](http://opencv.org/)) is an open-source library that includes several hundreds of computer vision algorithms.

We choose the findContours method it provides to get all the contours on the image and extract the Vector's contours individually.

Figure : Contour of Vector I

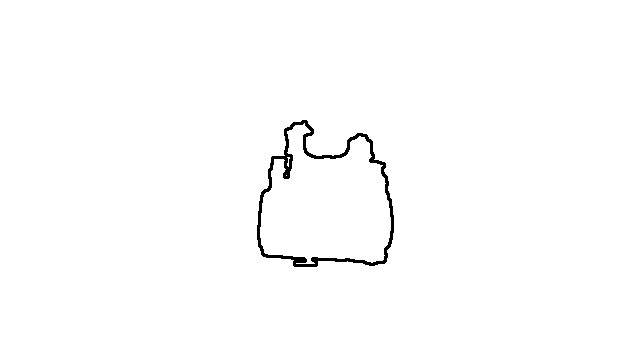
After a certain amount of contours have been extracted, the matchShapes() method is taken to match them and determine if another Vector has been detected based on the size of the returned value.

Figure :Contour of Vector II

However, this method is slow and inaccurate, and as shown in Figure 3, the Vector's silhouette is blurred as the distance increases, leading to false positives.

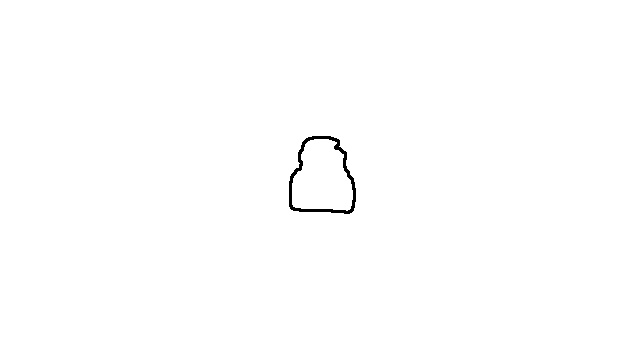
Unlike other simple shaped objects, each side of the Vector has a different profile from different angles, requiring a large number of profiles to be taken and not covering all cases.

Figure 3: Contour of Vector III

Furthermore, the speed is greatly affected by the need to traverse all the stored silhouettes for each recognition.

After a certain amount of testing, the method was found to be inadequate and was abandoned

## 2.2 Yolov5

Figure :Yolov5-Logo

To make the results more accurate, Yolov5 was used, in which we provided Yolov5 with information about the vector's model for it to learn, so that the Vector could eventually identify another Vector accurately and efficiently.

YOLOv5 is a family of object detection architectures and models pretrained on the COCO dataset, and represents Ultralytics open-source research into future vision AI methods, incorporating lessons learned and best practices evolved over thousands of hours of research and development.[[1]](#footnote-2)

### 2.2.1 Model training

In order for Vector to accurately detect Vector, it needs to be provided with a large amount of model information. We therefore took a large number of photos of Vector that could have been captured by the camera during the game.

Then we divided the images into three categories: val, train and test, and annotated them. Afterwards, the train.py provided by yolov5 used these images to train a model that could be used. A total of 100 rounds of training were conducted and the best models were used. (The best result means that the model will match the best when it is trained to match the images in val first)

Figure 5:Vector with recognition score in the camera

### 2.2.2 Use of model

Once we have a usable model, we need to apply it to the hide-and-seek game, here with the help of the camera api provided by the Vetcor.

With robot.camera.latest\_image, the latest image seen by the camera is temporarily stored and the saved image address is transferred to the detect\_img method. detect\_img method is based on Yolov5's

detect.py with slight modifications, it returns the result image and the score of the recognition. The result image allows us to visualise which images are judged to be likely targets, and the recognition score is the similarity to the target.

When the score exceeds the criteria, the target is detected.

Because the images provided by the training model were taken by the Vector's camera, which is not particularly sharp, we set a relatively high criterion to exclude those objects that were misidentified.

# 3. Traversing the Labyrinth

For seekers, the logic of finding the hider in the game of hide-and-seek depends on their own information of the labyrinth. How to effectively traverse the labyrinth is a prerequisite for a smooth game.

Traversing the labyrinth requires the vector to use the collected information as much as possible in the process of traversing, and to handle the data effectively, so as to model the labyrinth..

## 3.1 Introduction of Labyrinth

The labyrinth in this game consists of boxes of fixed size (200 mm \* 300mm \* 50mm). The space between the boxes constitutes the road, the box itself exists as an obstacle, and the road of the labyrinth is forced to be a straight road due to the functional limitations of the vector and to avoid excessively complex operations.

In addition, the labyrinth of the game is not fixed, and the Vector has no knowledge of the entire labyrinth before playing the game.

## 3.2 Concept for traversing

The vector's api provides basic information that can be used, including the coordinates and angles of the vector.

* The coordinate space is relative to Vector, where Vector’s origin is the point on the ground between Vector’s two front wheels. The X axis is Vector’s forward direction, the Y axis is to Vector’s left, and the Z axis is up.[[2]](#footnote-3)

Since the Vector's actions are basically in the plane, the Z coordinate is not needed. As a whole, only the x,y coordinates are used as reference objects.

* Obstacles in the labyrinth are objects that the vector needs to avoid in the process of traveling. The time-of-flight distance sensor provided in the api of the Vector itself is responsible for this behavior.

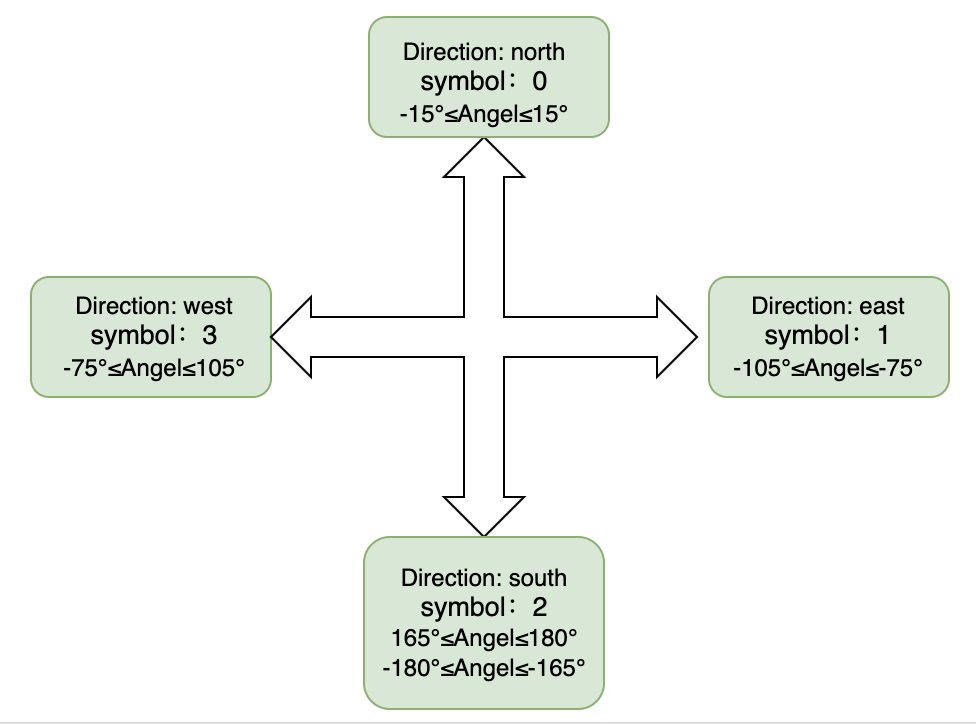
This sensor has a usable range of about 30 mm to 1200 mm (max useful range closer to 300mm for Vector) with a field of view of 25 degrees.[[3]](#footnote-4)

Figure :Absolute directions

* Defining absolute directions is possible by enforcing straight paths in the labrinty. In this game, we define the direction as follows:

0 for North, 1 for East, 2 for South, 3 for West.

The Vector can get this direction through its own degree attribute, as shown in the following figure.

## 3.3 Traversal Logic

With the two absolute reference objects of coordinates and direction, the problem of traversing the labyrinth is transformed into how the vector can effectively move in the labyrinth , collect and handle data.

### 3.3.1 Right-hand rule

Figure :Labyrinths virtualised with Pygame

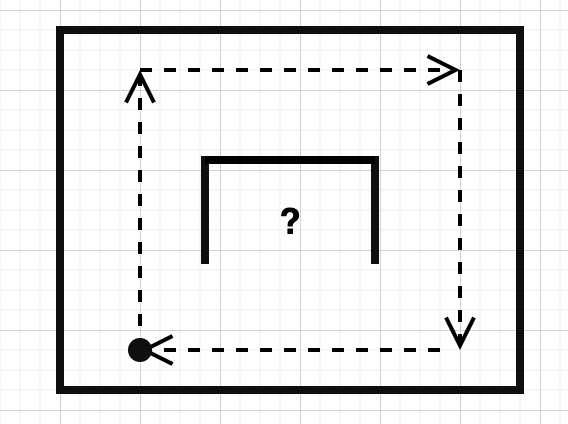
At the beginning, we adopted the right-hand rule as the method for traversing the labyrinth..

The right-hand rule means that whenever the Vector encounters an obstacle during the journey, it will choose to turn 90 degrees to the right and continue to move forward until it returns to the starting point.

In this process, pygame, as a simple virtualization tool, will draw the graphics of the entire labyrinth according to the change of coordinates.

This strategy was eventually abandoned for the following main reasons:

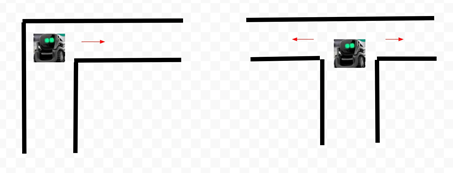
* In pygame, each coordinate is generated by a simple transformation of the coordinate points originally provided by the vector, which inevitably results in a large error. The accumulated errors are magnified during moving, and the resulting drawn labyrinth patterns are so inaccurate as to be almost unusable.

For example, if the Vector is given two coordinate points (101, 100), (99, 99), this will occupy both squares, but their actual distance is actually small.

* The right-hand rule does not recognise isolated obstacles. This means that if the obstacle is not connected to the labyrinth boundary, the vector cannot meet it during its movement.

### 3.3.2 Turning Point

Figure 8:Problems under the right-hand rule

After abandoning the right-hand rule, we found a simpler and more efficient way of traversing the labyrinth after some experimentation.

A turning point is a point inside the labyrinth where there is a way forward in both the x-axis direction (1, 3) and the y-axis direction (2, 4). When the Vector is at a turning point, it can see everything in its path, which naturally includes another vector it needs to find.

Figure :Turning Points Scenarios

In order to find the turning points in the labyrinth, the vector travels in such a way that every few seconds (adjusted according to the size of the labyrinth and the speed at which the vector travels, default 2s) a detection of the remaining three directions is carried out, which relies mainly on the distance sensors mentioned above. A threshold value is also set here, and if the detected distance’s value exceeds the threshold value, the program will define the detected direction as passable.

The second approach is, if the vector encounters an obstacle during its travel, a detection as described above in the first approach is likewise performed.

Together, these two approaches constitute the method for defining and finding the turning point.

### 3.3.3 DFS

In order to solve the problem of not being able to identify isolated obstacles for the first time due to the use of the right-hand rule, we have used a depth-first traversal method in this approach

First, there exists a stack. This stack acts as a depth-first tool, making a record of the current traversal and is also the basis for backtracking by vector.

There are three cases in which vector performs backtracking:

1. when it encounters a dead end (if vector detects an obstacle in front and finds that there is no passable direction to its left or right), the stack throws out the latest added turning point and vector returns to the latest turning point. The passable directions at that point are updated at the same time, and the direction from which it came is updated to not available.

. 2. This backtracking does not end immediately; after returning to the previous turning point and updating the information, the directions’ availability is checked again, and if there is another direction available at that turning point (if there are multiple passable directions, the Vector moves according to the priority of the directions, which is 0->1->2->3) , the backtracking will end and the vector will move in that direction, otherwise it will continue backtracking.

3. If the vector detects a return to a turning point in the stack during the move, it will also perform the same operation as described in 2.

After each of the above backtracking executions, a check, if the traversal can end, is handled. The condition is: There are no passable directions for all the turning points in the stack. The basis for this condition to take effect is that, firstly, the points out of the backtracking state itself no longer have a direction that can be continued. If the backtracking were to continue at this point, all points in the stack would be checked and moved in turn, so to avoid meaningless moves, the check can be made directly before the move.

# 4. Hide and Seek

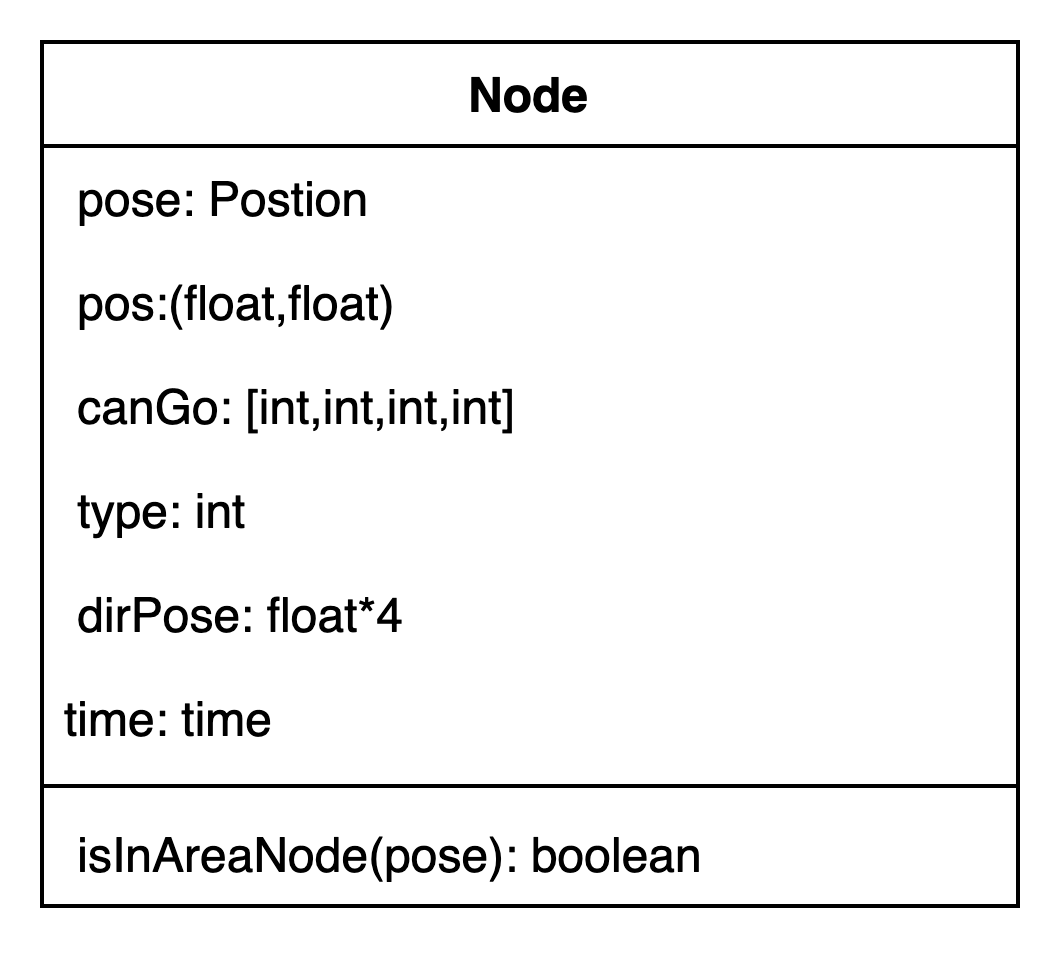
The prerequisite for implementing the logic of the hide-and-seek game is fulfilled once the set of turning points of the labyrinth has been obtained and the recognition of the other vector has been achieved.

## 4.1 Seeker

Seeker’s main task is to find the other vector hidden in the labyrinth as quickly as possible. The main information can be used to it is the set of turning points obtained in chapter 3. Since at each turning point all objects in the passable direction of that turning point can be seen, it is only necessary to perform a traversal of the set of turning points and to detect the passable direction of each turning point. If another vector is found, the game is over, otherwise move to the next turning point.

## 4.2 Hider

The logic of the hider is relatively simple; it takes the form of moving randomly through the labyrinth to avoid the seeker's search, i.e. it makes a turn in a random direction when it encounters an obstacle



# 5. Implementation

In this section we will briefly describe the exact implementation details of the code to match the hide-and-seek ideas we described in the previous chapters.

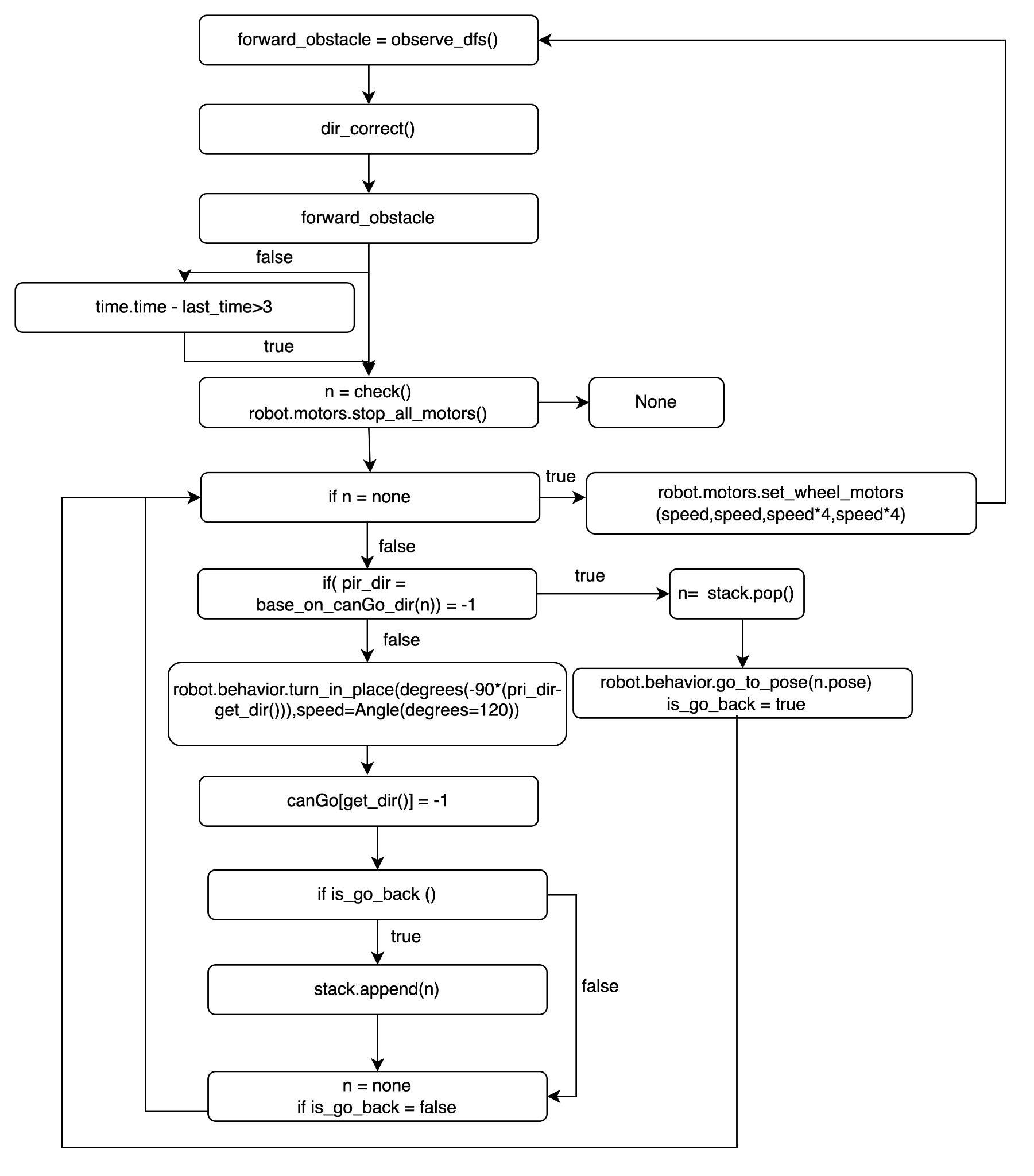


Figure :"Main" class flow chart

Figure 11:UML diagram of the “Node" class

First, we define the node class in the class.The node class contains information about the location of the node, its type and the time it was generated. The canGo list stores the directions in which the current node can move forward. isInAreaNode(pose) method determines whether the location of the input lies within the node range.

The main logic of the project is in main.py.

As can be seen in Figure 10, the move () method traverses the labyrinth in a loop,

At the beginning of the loop, the program first makes a judgement about whether there is an obstacle ahead (observe\_dfs()). Afterwards, the Vector will adjust the direction of advance(dir\_correct()).

If the Vector finds an obstacle or if the time interval exceeds 3 seconds, the Vector's engine stops immediately (robot.motors.stop\_all\_motors()). After this, the program makes a determination of the turning point.

The check () method checks whether the node at which the Vector is currently positioned is a turning point and returns it if it is, or returns None if it is not.

If check () returns an turning point, then the Vector will move on to the next action.

In the action phase, base\_on\_canGo\_dir(n) checks the canGo value of the turning point, which relies on the directional priorities described in Chapter 3.

If the method finds that there is a direction in which that turning point can go, i.e. the return value is not -1, the Vector will turn in that direction and set the canGo value for that direction to -1.

(robot.behavior.turn\_in\_place(degrees(-90\*(pri\_dir-get\_dir())))

If the value of the is\_go\_back variable is true, this turning point will be again added to the stack, this is to avoid that there is still a passable direction for that turning point after the turning point has been thrown.

At this point, the action phase ends (n=None,is\_go\_back =false) and continues back to the beginning of the loop.

If the method finds that there is no direction at that turning point in which it can go, i.e. the return value is -1. then it will enter the backtracking phase.

In the backtracking phase, the next forward destination is the last turning point on the stack (n = stack.pop() robot.behavior.go\_to\_pose(n.pose), is\_go\_back is set to true

After the Vector has moved to this turning point, it returns to the beginning of the action phase and repeats the previous steps until it finds a direction to go, at which point it ends the loop by setting n to None.

In addition, the flow of the check () method needs to be described again here.

The check () method, after stopping the Vector's engine, will first generate a node n whose constructor method will automatically record information about the current Vector's position.

Afterwards, the isNodeExist(n) method takes this node as an argument and determines whether the node is a turning point that already exists in the stack.

If the node is not in the stack, a determination is made as to whether the pathway exists in all three directions of the Vector (except the direction from which it came).

If the node is already on the stack, the end traversal condition is determined, as explained in Chapter 3.

After the loop that traverses the labyrinth has ended, the seek () method is performed. The hide () method, on the other hand, uses a separate thread that waits for keyboard input to start. The logic of these two methods is relatively simple and is explained in Chapter 4.

In addition to this, the detect () method, the method that identifies the object, also starts a separate thread. This method mainly calls the run method in predict\_func.py.

# 

# 6. Problem

Throughout the execution of the project, a series of problems arose, some of which were solved by certain means and some were left behind. This capital covers each of these issues and the responses to their resolution.

1. The distance sensor has a delay in determining obstacles.

During the Vector's journey, the distance sensor returns uninterrupted information due to the need to determine the presence of obstacles at all times, although this process is often misjudged or not judged due to delays and outliers.

To solve the problem of latency, there is a short hibernation after each judgment.

To address the issue of outliers, the distance value required to determine whether an obstacle is present is the average of the distance values collected over a short period of time.

1. Vector cannot move in an absolutely straight line and will inevitably produce errors.

During movement, due to the limitations of the vector itself, it is not possible to walk in the same absolute straight line as required in the simulation in the virtual world. After accumulating a certain amount of error, the vector's direction will be deflected by an angle, making the overall movement incorrect.

This problem was finally solved by correcting the angle at all times. The vector always checks the current angle and the current direction of travel, and when the angle represented by the current direction of travel deviates too much from the actual angle (i.e. exceeds a defined threshold), the vector corrects the angle so that it can travel as straight as possible.

1. The yolo algorithm has a delay in recognising the Vector.

Although the yolo algorithm recognises objects quickly, it is also unable to do so in real time due to the limitations of the Vector camera. In addition, the speed limitations of the algorithm and I/O delay inevitably result in a certain delay, which can cause errors in judgement.

Therefore, to solve this problem, the Vector only makes a judgement every 0.5 seconds, a short period of time that does not have a significant impact on the hide-and-seek results, but it does act as a good buffer against the stress of the algorithm.

1. The go\_to\_pose method provided by the Vector api sometimes fails to work.

The Vector’s walking depends on its own go\_to\_pose method. This methode tells Vector to drive to the specified pose and orientation. In navigating to the requested pose, Vector will use path planning.[[4]](#footnote-5)

However, the Vector's own path planning is not always accurate and sometimes uncontrollably reports errors, which affects the stability of the Vector's travel.

Since this method is provided by the Vetcor’s own api, there is no good solution. During the traversal of the labyrinth, we try to make the Vector walk in a straight line as much as possible, so that the coordinates provided do not cause too much deviation when navigating.

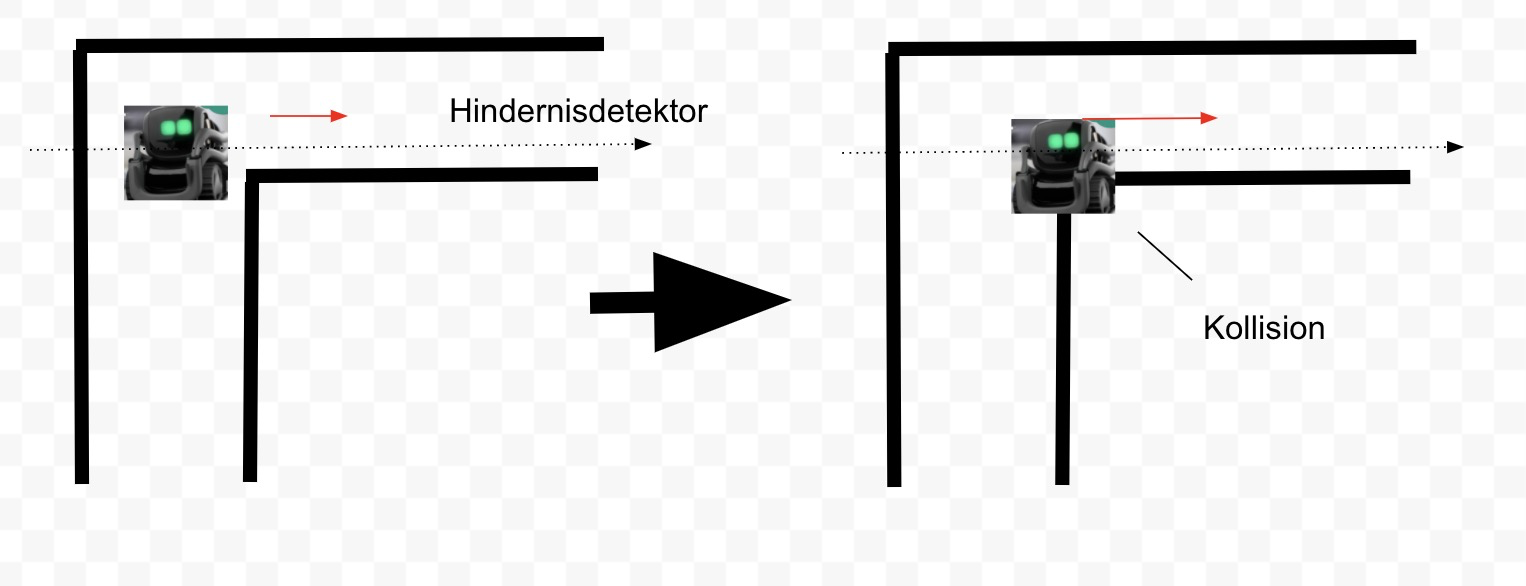
1. The distance sensor’s sector field of view is too small.This directly results in errors when judging obstacles.

Figure 12:Collision caused by distance sensor

Vector’s time-of-flight distance sensor has a usable range of about 30 mm to 1200 mm (max useful range closer to 300mm for Vector) with a field of view of 25 degrees.[[5]](#footnote-6)However, in Experiment it was found that the monitoring area of the vector is much smaller than the size described in the vector’s API. The width of the monitoring area is smaller than the width of the vector itself, which can lead to collisions with the labyrinth when the vector is turning.

This problem is caused by the Vector's own functional limitations. We are therefore unable to improve this.

# 7. Conclusion

In this project we implemented a simple game of hide-and-seek using the vector robot api, Yolov5 which involves deep learning and image recognition, and a depth-first algorithm from a traditional algorithm.

The program success rate of the project is not very high due to the problems described in Chapter 6, but this is a problem at the physical level that cannot be solved by writing logic for the time being.

This project focused on the logic of traversing the labyrinth and getting the turning points, weakening the logic of seeking and hiding, but also in the end basically achieving the originally expected goals and results.

# References

[Vector SDK 0.6.0](https://developer.anki.com/vector/docs/generated/anki_vector.behavior.html)

[Github-Yolov5](https://github.com/ultralytics/yolov5)

[Yolov5-Document](https://docs.ultralytics.com/)

# Erklärung

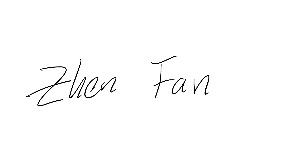
*Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel*

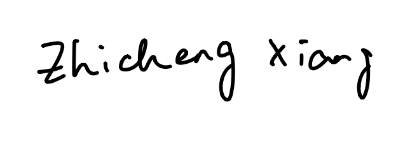
*vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten*

*anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT*

*zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu*

*haben.*

Karlsruhe, den 28. Feburar 2022 Zhen Fan

 Zhicheng Xiang



Junyi Zhu

1. https://github.com/ultralytics/yolov5 [↑](#footnote-ref-2)
2. <https://developer.anki.com/vector/docs/generated/anki_vector.util.html#anki_vector.util.Pose> [↑](#footnote-ref-3)
3. <https://developer.anki.com/vector/docs/generated/anki_vector.proximity.html> [↑](#footnote-ref-4)
4. <https://developer.anki.com/vector/docs/generated/anki_vector.behavior.html> [↑](#footnote-ref-5)
5. <https://developer.anki.com/vector/docs/generated/anki_vector.proximity.html> [↑](#footnote-ref-6)