BACHELOR

Space segmentation into semantic navigation areas from 2D gridmap made by AGV

van Galen, Dirk J.H.

Award date:
2021

Eindhoven University of Technology
Department of Mechanical Engineering

Space segmentation into semantic navigation areas from 2D gridmap made by AGV

Eindhoven, September 7, 2021

Dirk van Galen

Supervisor TU/e: Cesar Lopez Martinez
René van de Molengraft

Contents

1 Introduction

Over the last years, Dutch healthcare institutions have faced a shortage of workforce [1]. Due to the higher pressure on the healthcare system, more work needs to be done by too few employees. One way to solve this problem is to help with basic, time-consuming tasks such as moving containers or beds. These simple tasks can be done by robots.

The Robotics Lab is developing an AGV (automated guided vehicle) able to assist in hospitals by taking over logistical operations such as moving containers and hospital beds. One of the features of the robot is its capability to autonomously navigate in designated areas, which are predefined in a map. These areas represent virtual roads the robot can move in. These predefined areas are either "corridors" or "crossings". In a corridor, the robot is free to move forward and backward while having perception of what is happening in front of it. In crossings, the robot has limited perception since entities can approach it from beside, which it has no perception of. When the robot is arriving at a crossing, it slows down, for example. Thus, the behaviour of the robots is different in either a corridor or a crossing.

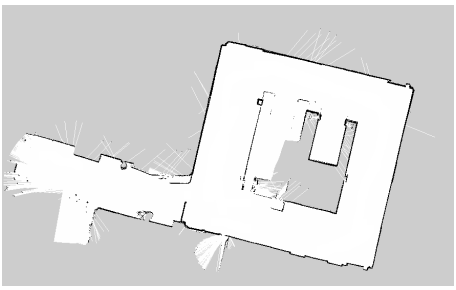
At the moment, these navigation areas are specified by user input in a grid map scan made by the robot. This makes setting up the robot in new unknown areas time-consuming. It is therefore necessary to automate this process, to present a working robot to hospitals in the near future, where the scan of the map is automatically annotated with the right navigation areas. This makes it possible to demonstrate the robot in new areas quickly and effectively, and thus make a more convincing presentation of the possibilities of the robot.

For humans, it is easy to recognize rooms, doors or walls by looking at the overall structure of the gridmap. A human being can recognize areas by applying previously learned things about floorplans and real-life situations. The combination of how occupied cells (pixels in the gridmap) are aligned to form walls, which in turn form corridors and crossings, is understandable for most humans. The algorithm is provided only with a grid map containing free and occupied cells. For an algorithm, the process of defining each required area is not straightforward, since it has no ability to look for certain semantic information such as doors and corners.
The current setup transforms this grid map, with little semantic information, to a more rich, useful map with the right areas assigned, with human input. An algorithm should deem any human input unnecessary to automate the process of extracting the required semantic information from the basic grid map.

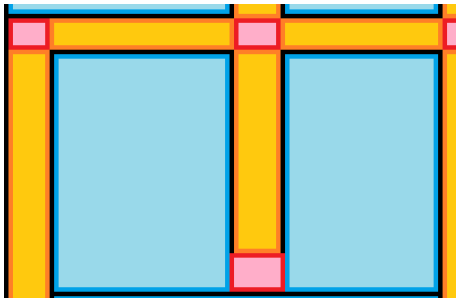
In this report, the development of the tool which will annotate the gridmap with the right areas is described (in the report the focus was on finding the crossings, since the other areas are defined as corridors). The method which was chosen relies on finding free space which the robot can move to.

2 Problem statement

Setting up the navigation areas for the robot in new, unknown areas is time-consuming and requires user input.
In Figure 2.1a an example of a generated gridmap is shown. In Figure 2.1b a simplified partition of the gridmap is shown. In this figure, black cells represent a wall, red cells a crossing, yellow cells a corridor and blue cells a room.



(a) Example of a grid map generated by the robot



(b) Different navigation areas

Figure 2.1: A real example of a generated gridmap and a filled-in representation of an area

As can be seen in Figure 2.1a, limited data is provided. A cell is either occupied (black), free (white) or unknown (gray). While for a human being it is easy to recognize doorways and hallways, the lack of semantic information makes it a complicated task for an algorithm to divide each gridmap into the right navigation areas.

Requirements

A tool that divides the grid map must work according to certain requirements in order to say it is either successful or not.

The tool must:

1. Recognize free space
2. Enrich the given gridmap by showing and defining semantic navigation areas.
3. Be robust; it must work for any gridmap.
4. Take into account the robot's footprint.
5. Take into account the robot's minimal turning radius
6. Work within 10 minutes.
7. Require minimal user input.

Navigation areas

The navigation areas are defined as follows, listed from most to least important;

1. Corridor: Area in which the robot has a clear perception of what is happening in front of it, no need to expect sudden different entities from the side.
2. Crossing: Area in which the robot has limited perception of possible events happening, for example, another robot approaching from the side. This can thus also be a corner.
3. Room: Destination of the robot. In this area the robot uses a different navigation algorithm than the navigation algorithms used in crossings and corridors.

3 Method selection

To divide the gridmap into different segments a method must be derived. This method must ensure that the right area is given the right label. This can be done in different ways. In this report the most viable methods are explained and compared. It is important that the chosen method is robust, does not require any user input and the method must also work quickly. If the algorithm would take too long, say 10 minutes, it limits the user to put the robot to work quickly. The other requirements are also shown in the requirement list.

Literature Research

On the internet and in the available databases there is information available about methods to divide, recognize or label different areas in provided pictures. The most relevant available methods and processes are described here.

Room labeling method

Many methods available are meant to label every room in an area with one unique label. For example, Room Segmentation: Survey, Implementation, and Analysis, by Bormann et al [2] explains and compares four different methods of dividing a 2D grid map into different rooms. These methods are displayed in Figure 3.1

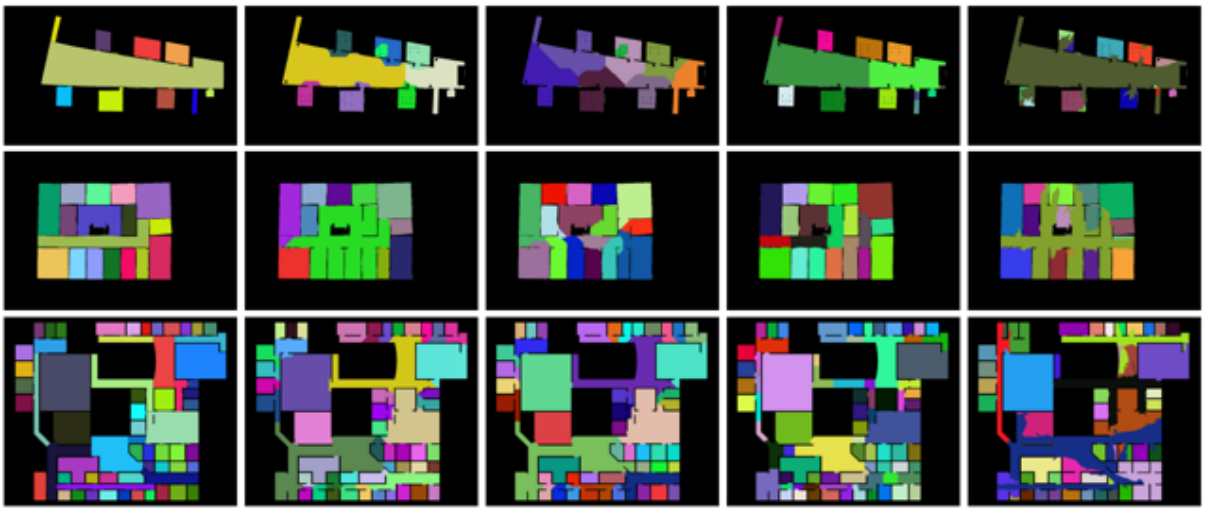


Figure 3.1: Four different segmentation methods, the first column depicts the ground truth room segmentation from human labeling, the second column shows morphological segmentation, the third column yields the distance-based segmentation, column 4 is the Voronoi graph-based segmentation, and column 5 shows the feature-based room segmentation. [2]

The advantage of these methods is that they are tested and robust. Most of the methods succeed in labeling different rooms correctly. A problem in these methods is however that while the algorithms are useful in labeling different rooms, they lack possibilities to recognize different rooms. Thus they will not be able to recognize either a corridor or a crossing. They will merely recognize that a certain room is in fact a room, while for the problem at hand it is necessary to define also the “function” of the area. The task requires the algorithm to recognize different areas and label them accordingly.

Feature-based room segmentation method
A method described by Oscar Martinez Mozos seems promising since this method distinguishes between doors, corridors and rooms [3]. This method combines different possible shapes to find possible rooms, doorways and corridors. Where the distinction is made it is possible to create a similar method to distinguish between corridors and crossings. The method is based on taking a point in a 2D Gridmap and simulating a 360° laser-scan, also known as raycasting. The resulting scan is then compared to training data by the AdaBoost algorithm (a boosting algorithm) [4] in which previously taken scans in known places are shown. An example of this is shown in Figure 3.2. A scan taken in respectively a room, doorway and corridor is shown in Figure 3.3. Please note, used definitions for areas in the paper by Martinez Mozos are different from the definitions used in this report.

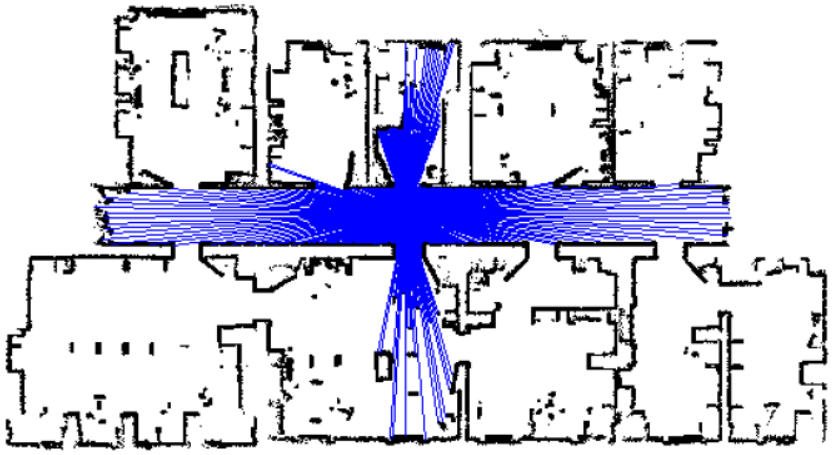


Figure 3.2: Range scan obtained by a mobile robot in a corridor. The image also shows the complete map of the environment where the scan was taken. The scan covers the complete 360° field of view of the robot. [3]

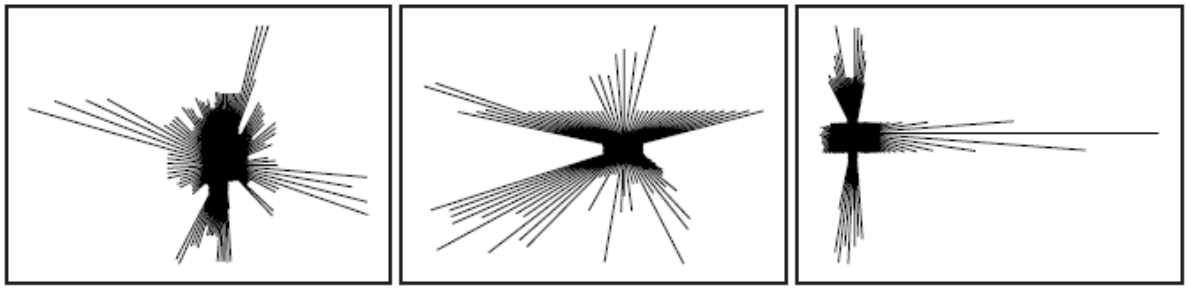


Figure 3.3: Example scans recorded in a room, a doorway, and a corridor. [3]

This method (also slightly covered by Bormann et al [2]) provides promising results based on the relevant features. Once enough training data is provided the solution should give reliable results. However, this is also where problems can occur. The method is dependant on training data and unreliable when a situation is shown which is not similar to the training data, or insufficient training data is available.

Conclusion
After considering known methods and speaking with experts on the topic the focus was laid on finding or creating a method that focuses on the free space that the robot encounters when driving around, instead of finding a method that relies on trying to recognize spaces from an image, without knowing what it actually represents. Moreover, it is difficult to find enough training data, or to find reliable and representative training data. Following this realization the proposed method is explained in the remainder of this report.

4 Proposed method

To reliably determine which space should be labeled “crossing” it is robust to work with the free space available. It can be derived that whenever the robot encounters free space which is not straight in front of it, it must be in a crossing at that moment. If it were to only find free space in front of it, with no free space to its sides, it must be in a corridor. When a situation occurs where it finds free space in front, and to one side, it is in a T-junction, which will also be labeled “crossing”. In a corner, the space will also be labeled “crossing”. To visualize this, Figure 4.1 shows the situation with the robot in a corridor. Figure 4.2 shows the robot while approaching a crossing.

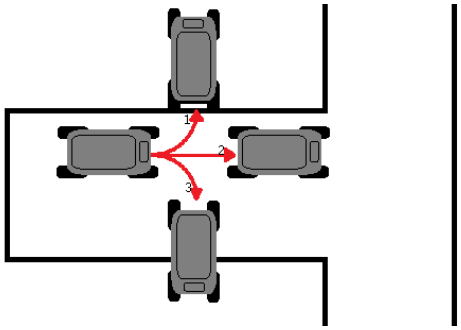


Figure 4.1: Case 1; Visualisation of the free space method while in Corridor

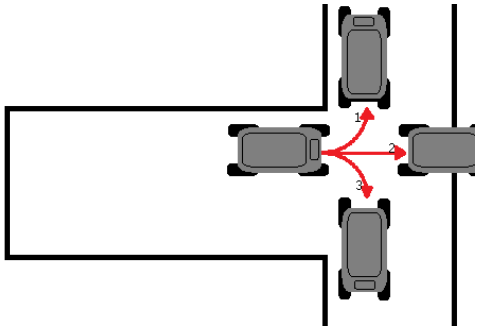


Figure 4.2: Case 2; Visualisation of the free space method while approaching a crossing

In Case 1 it is clearly visible that the turns 1 and 3 will result in the robot hitting a wall. Driving straight ahead (“turn 2”) will not result in any collision. Thus it is most likely that the robot is in a corridor at this moment. Case 2 displays what happens when approaching a crossing. Turns 1 and 3 result in possible turns. Turn 2 results in the robot hitting a wall. Based on turn 1 and 3 the robot can know that it is approaching a crossing.
In an algorithm more and different turns can be taken, instead of only turns of 90°. Furthermore, smaller steps can be taken, where the algorithm can take account of the size (footprint) and the turning radius of the robot. This method could be very reliable when the right parameters are set. Difficulties could arise when it is not clear for the algorithm in what way it should approach a certain map.

The method relies on checking if a certain position is possible for the robot to move to (free space). A branch could be formed where the robot moves towards free space. When it only encounters free space in front of it, it will move forwards to this space. Once it encounters free space after taking a turn, it will save this position as a possible position to move to when its current branch dies out. A representation of these branches is showed in Figure 4.3

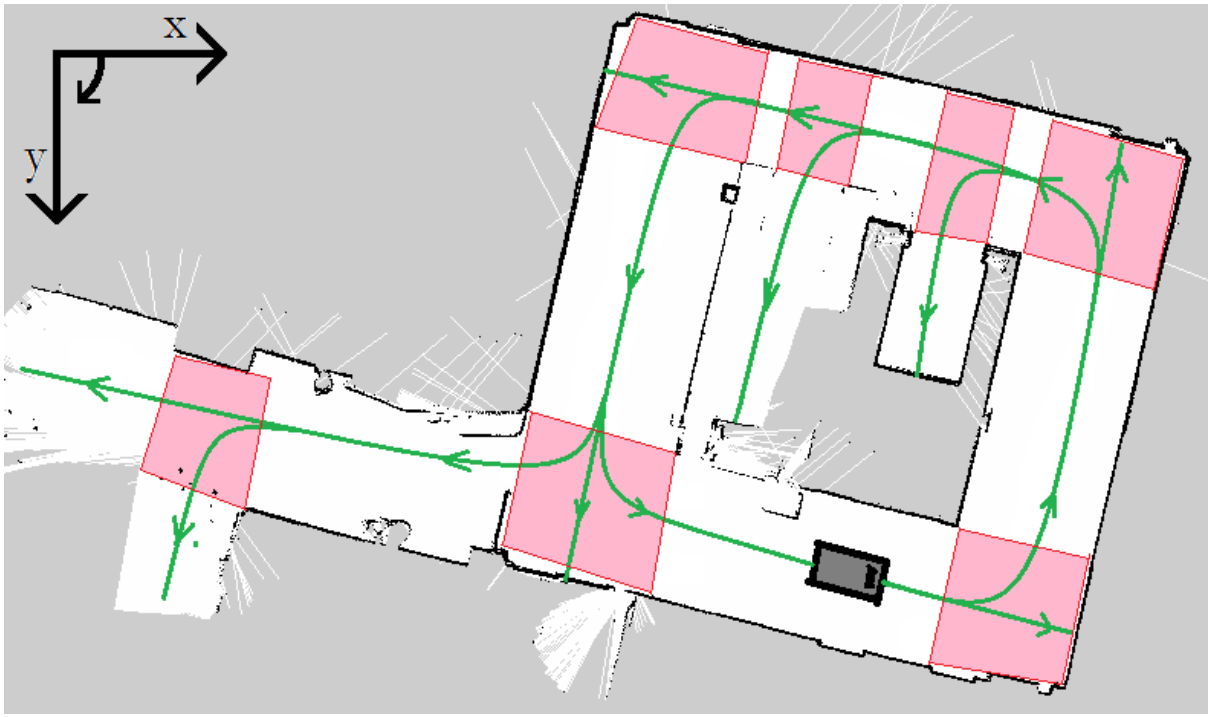


Figure 4.3: Representation of the branches (in green). In red are the areas which are "corners". Only successful turns are shown, turns that resulted in hitting a wall are not shown.

5 Implementation

To implement the algorithm, the choice was made to use Python. Python is open-source with a lot of information available online [5]. Python also has many packages dealing with image processing available which could contribute to the algorithm. OpenCV [6] is one of these packages. It has many available functions, for example `pointPolygonTest()` [7], which is a function that examines if a certain point is inside a polygon.

Algorithm

In this section the main functionality is explained. To start the algorithm, a starting position must be taken. Afterwards, the dominant direction, which is the longest available direction and its corresponding angle, is found. From here, the first branch is formed. The algorithm will follow a certain branch as long as possible, until it hits a wall. When free space is found beside the robot, a new branch is formed. In this process, all visited spaces will be labeled with either “corridor” or “crossing”.

To elaborate on the algorithm a flowchart is used, where each block is labeled. The flowchart is in essence which steps are followed in the algorithm. The flowchart is shown in Figure 5.1. The flowchart consists of different blocks which all have their own function in the algorithm. The blocks are also referred to in the pseudo-code and the most important blocks/steps are elaborated on further on in the report.

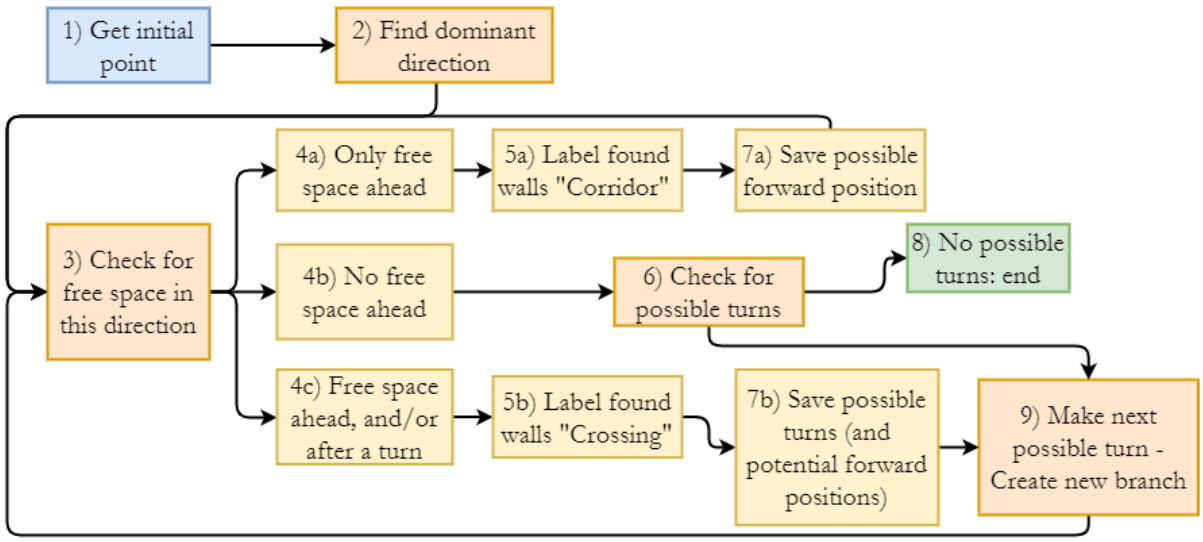


Figure 5.1: Flowchart explaining the concept of the algorithm

The flowchart in Figure 5.1 is also explained in the following pseudo-code:

```
## Block 1 (initial point):
start_pos = input([x,y])
## Block 2 (dominant direction):
start_angle = dominantdirection(start_pos,map)

r_pos = [start_position,start_angle]

crossing_walls = []               #List containing all pixes marked as crossing
pos_available = [r_pos]           #List containing all possible positions, including
    startposition

## Block 6 (check possible turns):
while len(pos_available)>0  #Loop will continue as long as there are possible positions
    for the robot

  ## Block 9 (make next possible turn, starting with startposition):
  r_pos = pos_available.pop(0)  #Save first possible position and remove it from the
    available positions list

  ## Block 3 (check for free space straightforward and to the sides):
  for turn in range(-130,130,10):
    robot_test_pos = turning(r_pos,turn) #Turn the robot over a given angle
    robot_test = getcorners(robot_test_pos)

    ## Block 4 (check when finding free space what direction it is in:)
    onwall = findwall(robot_test,map,contourspace,1) #Find all walls surrounding the
    current contour

    if robot_test not on wall and not encountering wall:
      possible_turn.append(turn)

  ## Block 5&7 (label crossings & add new possible new positions to list):
  if possible_turn:
    if turn <= -30:
      r_new_left_pos = turning(r_pos,turn)
      pos_available.append(r_new_left_pos)

    elif -30 < turn <= 30:
      r_new_straight = turning(r_pos,turn)
      pos_available.append(r_new_straight)
    elif turn > 30:
      r_new_right_pos = turning(r_pos,turn)
      pos_available.append(r_new_right_pos)
```

Key variables and functions

In this subsection the main variables, functions and structures are explained in more detail, to elaborate on the pseudo-code displayed in Subsection 5 Algorithm.

Gridmap
The gridmap is loaded in the algorithm. This gridmap is then converted into a grayscale image which is made up of an array containing the value for each pixel at every coordinate (function: `convert2array()`). This image can then be handled as if it has a x and a y-axis. Angles are evaluated with respect to the horizontal x-axis. These axes are also shown in Figure 4.3.

Variable `r_pos`
To start the algorithm, a position for the robot is taken. This variable `r_pos` contains the coordinates of the centre of the robot and the angle it's rotated over with respect to the x-axis. To start, a x and a y-coordinate is needed for the start position. The algorithm will find the dominant direction (which is the longest direction in which it can travel) and the corresponding angle. This is done in Block 2 of the flowchart. The coordinate of the middle of the robot and the angle together make up `r_pos`. A visualisation of this "raycasting" is shown in Figure 5.2

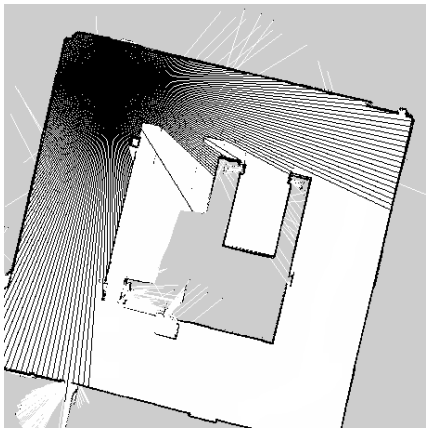


Figure 5.2: Visualisation of how the dominant direction is found using raycasting

The code for function `dominantdirection()` is shown in Appendix A.1.1
With only `r_pos` the contours of the robot can be drawn. For this, the function `getcorners()` is used. This function uses `r_pos` and trigonometric functions to determine where the four corners of the robot are. The code for the function `getcorners()` is shown in Appendix A.1.2

Function `findwall()`
To examine whether the given robot is on a wall, the function `findwall()` is used. This function works by comparing if the contour of the robot contains any non-white pixels. First, the contour is drawn in the contourspace, which refreshes after each iteration. An example of the contourspace is shown in Figure 5.4. Once the contour is drawn, the area surrounding the robot is checked for non-white pixels, meaning either a wall or unknown space. If a non-white pixel is detected in the given gridmap, the position of this pixel is checked with the OpenCV function `pointPolygonTest()` to see if the point is inside the contour. If this is the case, it is added to the list `wallsincontour`, which contains all the non-white pixels in the contour. The code for the function is shown in Appendix A.1.3. The function `findwall()` is also elaborated on in the flowchart shown in Figure 5.5

Function `turning()`
To accommodate for the turning radius of the robot, a function is used to calculate the next position of the robot. This function works by turning the robot over its turning radius and a certain given angle ("`turn`" in the pseudo-code). Once it reaches the correct angle, it travels straightforward until it's traveled

a fixed distance (`curvelength`). This process of following the turning radius and then traveling until having moved a predefined distance is shown in Figure 5.3

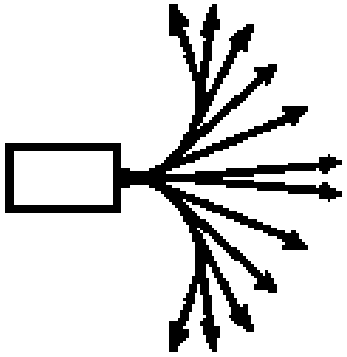


Figure 5.3: Visualisation of the turning process

The fixed distance `curvelength` can be altered by changing the factor over which the curve length must be multiplied. Because of this, it is possible to make it travel a shorter distance when traveling forward for example.

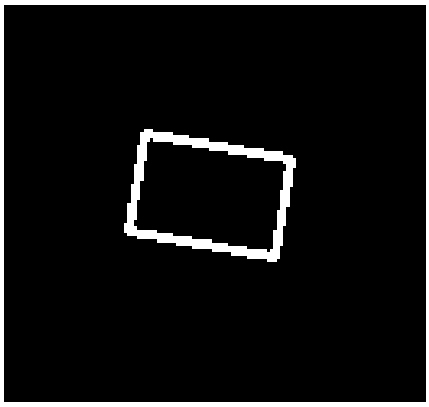


Figure 5.4: Example of the contourspace during one iteration

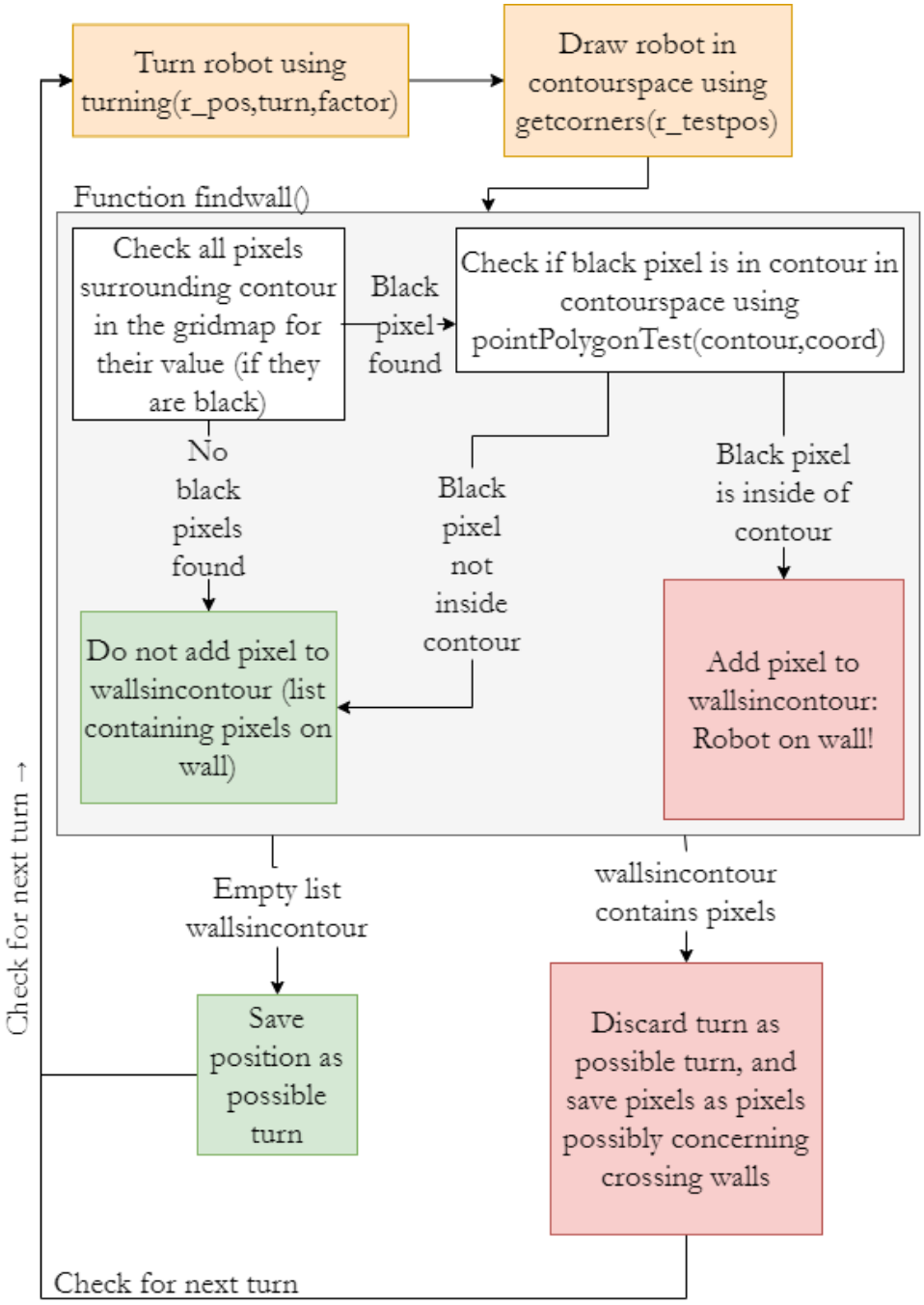


Figure 5.5: Flowchart showing the function findwall and how it is checked for every turn

Finding free space

Key functionality of this algorithm is to find available, free, space. Its basis is found in the fact that if there is free space to the sides of the robot, the robot must be in a crossing at that moment. To accommodate "traveling" through the gridmap and finding all available free space, the algorithm will check left turns, movement straight forward and right turns.

After checking multiple different turning angles via the turning process described before, the algorithm has possibly found multiple possible turns. It will then take the average possible left turn, the average possible right turn and the average of the possible forward moves. This way, it will find any possible route. After testing it was found to be best to define average possible forward moves as moves with a turning angle in between -30° and 30°. Meaning, any move between these angles is considered straightforward.

This helps with limiting the number of false positives in finding corridors (moreover these false positives in the "Displaying results" subsection). A visualization of the process of finding the average possible turn is shown in Figure 5.6. Contour "3" shows that the average possible turn is taken as a next possible position. As one can see, some of the turns it makes to the left are discarded as they are on a wall. The process of finding which turns are possible and which turns are not is shown in the flowchart in Figure 5.5.

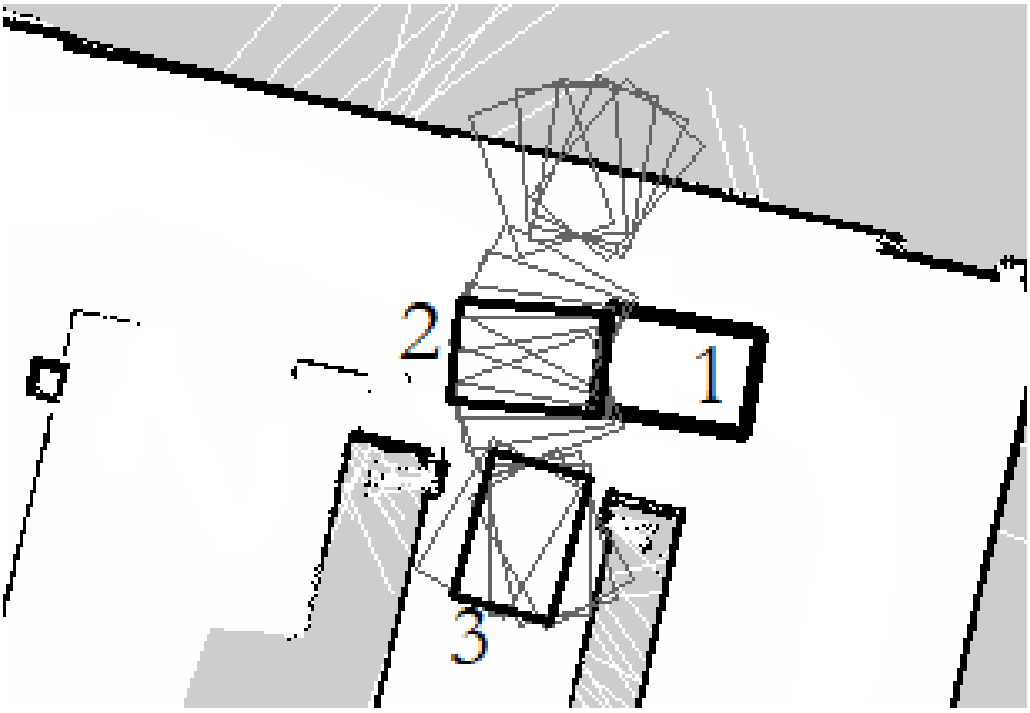


Figure 5.6: Average turns. [1] is the starting position. [2] is the average possible move forward. [3] displays the average possible left turn. Contours in gray are all the checked contours

Displaying results

While running the algorithm and moving the robot through the gridmap, the walls which are detected as crossing walls are saved after each iteration in `list: crossing_walls`. To show where all the walls are that make up the crossings, the pixels which are detected inside a crossing can be displayed. The problem with this is that currently, the algorithm makes mistakes. Once a robot ends up in a certain position in a corridor it detects that there is a possible turn, while in fact it is in a corridor, because of the width of the corridor. This is displayed in Figure 5.7. Contour 1 displays the robot in an actual crossing, where it makes a correct turn (and also labeling the detected walls "crossing walls") to contour 2. The robot displayed by contour 2 then makes a wrongful turn to contour 3. Because a turn was made, the detected walls are labeled "crossing walls", even though the robot was inside of a corridor.

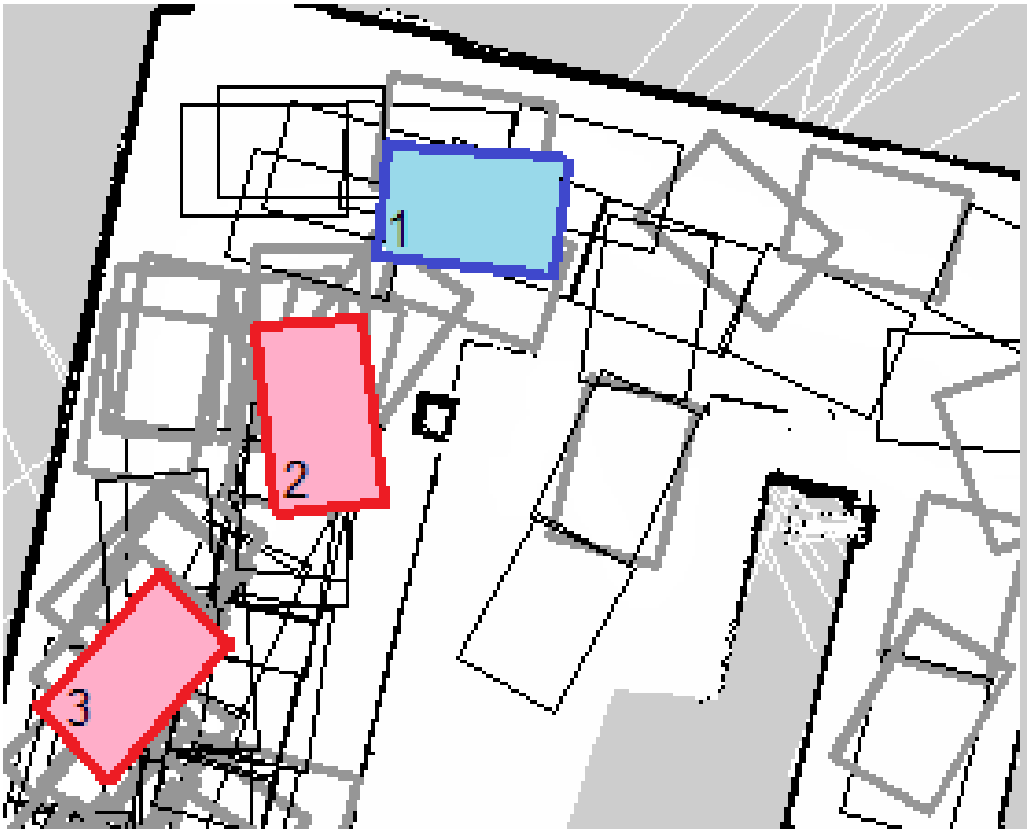


Figure 5.7: Example of how the robot is able to make a turn in a corridor, and then giving wrong information about the surrounding walls.

To counter this phenomenon, the algorithm works around the gridmap multiple times, and every time it checks for crossing walls. Once the algorithm has ran for a certain amount of time (dependant on given variables such as given gridmap, robotsize with respect to gridmap), almost all wall pixels are recognized by the algorithm. Then a weighing can be added to all found pixels. Meaning; when a pixel is detected as "crossing wall" more times, it will get a higher ranking. To do this, all pixels in `crossing_walls` are checked for how often they occur (how often they are being recognized as a crossing wall). They are stored in a `dictionary: rank_coord` which contains the coordinate as a key and the number of occurrences as a value.

```
snippet of rank_coord:

 (747, 293): 493, #Likely to be actual crossing wall
 (747, 294): 504,
 (748, 293): 508,
 (749, 293): 506,
 (660, 275): 78,  #Probably a false positive crossing wall
 (661, 274): 74,
 (661, 275): 76,
 (662, 273): 70
```

The code in Appendix A.1.4 shows how the visualisation is created.

6 Results

How the results are created is explained in the subsection "Displaying results" in Section 5. The resulting visualisations are the end result of the algorithm, and show how the crossings are found by the algorithm.

In Figure 6.1 the visualisation resulting from 1132 iterations is shown. The figure shows a gridmap which is the result of a scan made by the robot. This figure is a large mess of different contours, but it allows for the viewer to see what paths the algorithm has taken to move in the gridmap and which areas it has passed most often. Grey contours represent contours made after a turn. These grey contours are however hard to see in between of all the black contours (contours where the robot traveled forward in the algorithm). In Figure 6.2 the walls which are detected as crossing walls at least once are shown. As is visible from the figure, almost all walls are detected at least once as a crossing wall. This figure has no weighing attached to it. From this figure it can be concluded that many false positives are detected, since most of the walls coloured white are actually corridor walls. Figure 6.3 shows the crossing walls with a weighing, and displays how corners are detected by the algorithm much more often. This shows how the algorithm, over time, detects crossing walls more often.

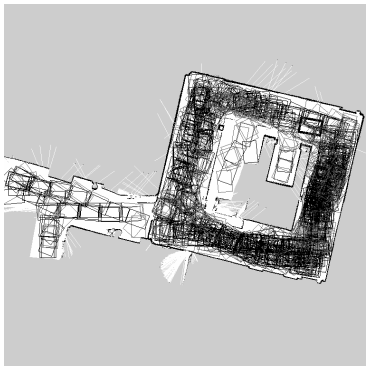


Figure 6.1: The result of 1132 iterations, showing how each wall is detected multiple times

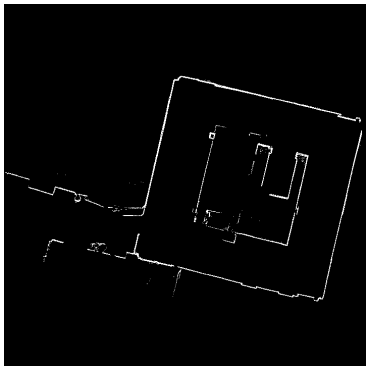


Figure 6.2: All detected crossing pixels are in white, without weighing

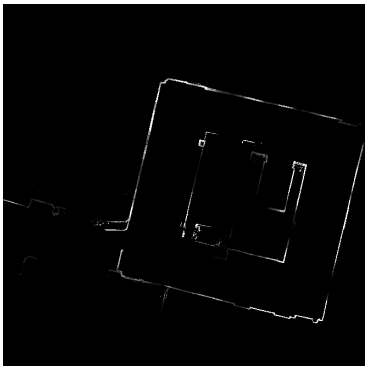


Figure 6.3: Crossing walls visualisation with ranking based on number of occurrences

As can be seen in Figure 6.3, the algorithm provides information on the semantics of the position of crossing walls. For a different gridmap the robot will follow different paths. A different gridmap is shown in Figure 6.4, with the corresponding visualisations. This is a computer-generated gridmap, not one generated by the robot.

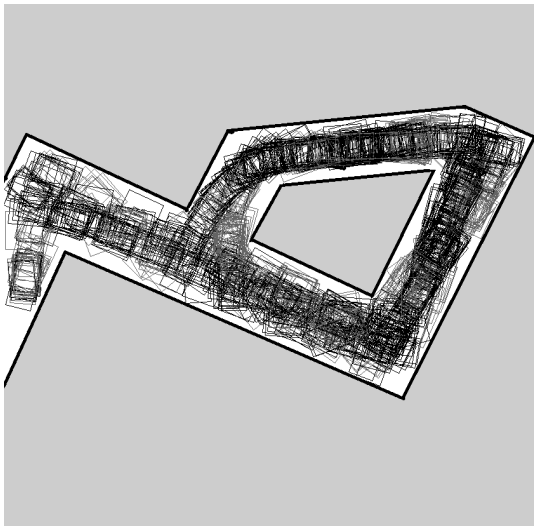


Figure 6.4: The result of 937 iterations, showing how each wall is detected multiple times.

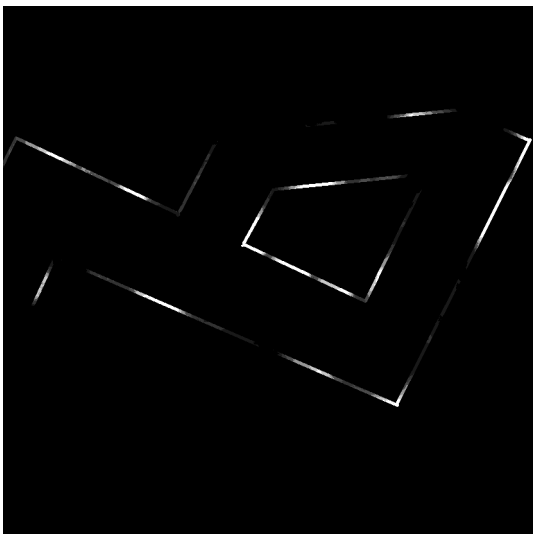


Figure 6.5: Crossing walls visualisation with ranking based on number of occurrences.

In a gridmap with relatively thin corridors, the algorithm will have fewer false positive crossing detections, because the case in Figure 5.7 (case with wrong turn) will happen less often. To simulate this a gridmap with thin corridors was created. The results of running the algorithm in such a map are shown in Figure 6.6. This result shows in the crossing walls visualisation much more clearly where the walls surrounding the crossings are compared to gridmaps with relatively large corridors.

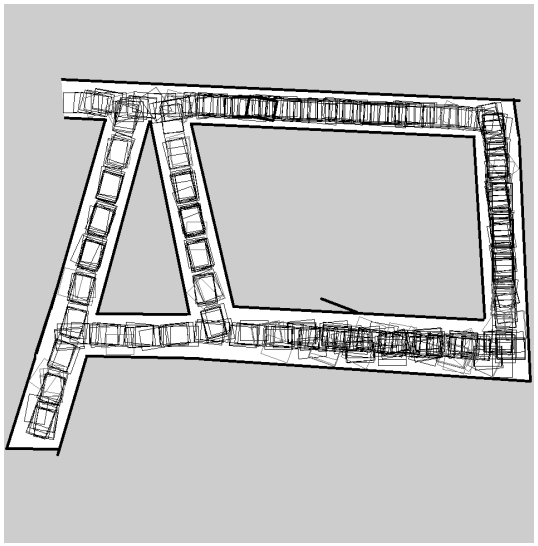


Figure 6.6: The result of 258 iterations, showing how each wall is detected multiple times.

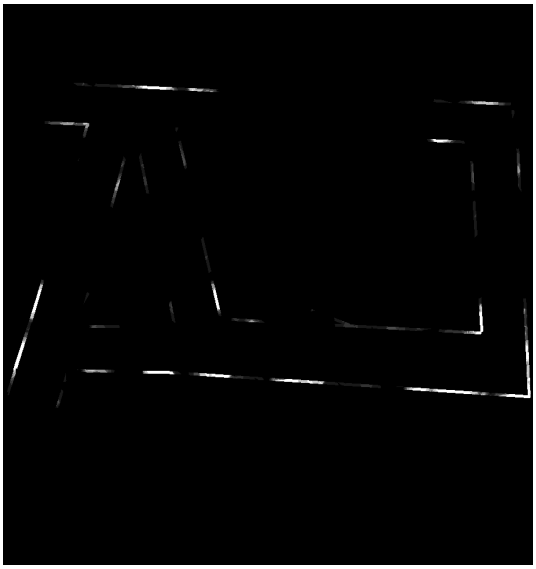


Figure 6.7: Crossing walls visualisation with ranking based on the number of occurrences.

The visualisations shown in Figures 6.3, 6.5 and 6.7 are the final results of the algorithm on three different maps. In the results it is clear that most free space is detected, and with it, the crossing walls. Whenever a robot is able to turn around a corner the turn is made. The algorithm is effective in finding all spaces it can go, and thus seeing where free space is. The corners are detected but it also displays the shortcomings of the algorithm. The combination of the number of false positives and the method of defining the navigation areas is not viable. This will be explained in more detail in Section 7 Discussions.

7 Discussion

To assess whether the algorithm is successful in helping with the recognition of the different navigation areas, it is necessary to check if and to what extend the requirements are met. In the Introduction multiple requirements are stated.

Assessing requirements

As can be seen in the Section 6 multiple figures display the results of the algorithm. Especially Figures 6.1, 6.4 and 6.6 are of interest, since these Figures display the steps and paths taken. When looking at these figures, it is clear that the algorithm succeeds in finding free space. The first requirement is met, since free space is recognized and detected.

The first problems come with the second requirement, which states "The tool must enrich the given gridmap by showing and defining semantic navigation areas". In the results, one can see that crossing walls are detected. There are also corridors that are detected as "crossing walls" however. The combination of how a crossing wall is detected and how false positives can have a large impact on the final result are the root cause of this problem. Since the crossing walls are detected in the step where the turn is made often walls just "before" and "after" the turn are also detected as crossing walls. A solution for this would be to define the area which is in a "crossing" as the area under the path the robot travels to reach the new contour.

The next problem with the second requirement comes with the method of defining the navigation areas in the gridmap. Currently, the areas are defined by showing which walls surround the crossing/corridor. A desired result would be an algorithm that defines each area with points that can form a polygon. Those coordinates of the polygon which make up each area is useful for the navigation algorithm used by the robot. Currently, the robot uses polygons defined by coordinates to know where each navigation area starts and ends. Thus the method of just finding the walls containing each area would not suffice. There needs to be an improvement on this part (defining where each area is exactly) to make the tool meet this requirement. A suggestion for this would be to fit contours between the walls of the crossing, and determining what polygon spans the crossing space. But at this moment this has not been managed.

The third requirement, which states "the tool must be robust; it must work for any gridmap" is not met to its full extend. It is possible to use any gridmap and get results. In every gridmap used, all spots in the gridmap are covered. There lies a problem in the method of following branches, since at the moment the next step the algorithm takes is not necessarily in the same branch. Because of this, the algorithm tends to reach certain areas more often. This then increases exponentially because an area which is visited more also creates more possible positions, and thus keeping that area more active. Therefore, sometimes the weighing factor used to create the crossing walls visualisation is not truly irrefutable. This symptom is visible in Figures 6.1 and corresponding crossing walls visualisation Figure 6.3. The left side of the corridor in the gridmap is not very notable, even though it has a clear crossing.

When a gridmap with very wide corridors is used, a large number of false positives will be given. It can however be discussed that these very wide corridors have in fact the same limitations for the navigation of the robot as crossings. When the robot is in a position were it is able to make a multitude of turns to the side, it is also possible to have this corridor meet the "crossing discription". Since as explained in the "Navigation areas", found in Section 2, a crossing is defined as an area in which the robot has limited perception of possible events happening, for example, another robot approaching from the side. In wide corridors another robot or human can approach it from the sides, and the area is then correctly defined as a crossing. This shows that the definitions of the different navigation areas are disputable and can be altered to change the approach of the desired algorithm.

The fourth and fifth requirements are met by the algorithm since the footprint and its turning radius are taken into account when trying to find free space for the robot to move to. The sixth requirement is a time requirement, which is also met. On a standard laptop the algorithm can be stopped within 10 minutes to get a useful result. The seventh requirement, which states that the tool must require minimal user input is met to a certain extent. It is necessary for the user to input three variables: the turning radius, the size of the robot and finally the starting location of the robot. This number could be reduced

if the variables can be known from a given gridmap. Another way to reduce the amount of input variables is to make the starting position any random location in which the robot can start the algorithm. If the robot fits in the position and is able to form branches, it suffices.

Recommendations for improving the algorithm

For the algorithm to meet or exceed all stated requirements, the following recommendations are made. First, the method of defining (with polygons/coordinates) what area has what label must be implemented. This way, the results are useful for the robots navigation algorithm. The method also needs improvement on detecting false positives. For this to happen the process of turning and detecting walls must be altered. Lastly, in an ideal situation the algorithm will know the size of the actual robot (which is constant) and its corresponding turning radius, and change the size of the simulated robot according to the given gridmap and create the starting position itself. This way there is no required user input.

Recommendations for the process

Evaluating the process of creating the algorithm, too much time was spent in the wrong direction in the start-up phase. There have been difficulties in finding the right method, and a significant part of the literature research, and finding the best possible method to tackle the problem, was focused on tactics involving neural networks and machine learning. After talking with experts on the subject the choice was made to move to a method relying on free space. This method seems very promising and the results show that it is possible. There has however been too little research on how to implement this correctly, since due to time restrictions it was important to start on the implementation phase. To improve the quality of the algorithm, more research should be don on how to implement such a “Free space finding” algorithm.

After the problem at hand was understood correctly and the method was derived, the implementation phase started. Looking back, this phase was too unstructured. It would be beneficial to increase the time spent on laying out the desired method and corresponding algorithm in paper or in for example a flowchart, before starting to code. This way, it could be known beforehand that “defining the actual navigational areas” would come with some problems.

8 Conclusion

To find the best solution for setting up navigation areas in the gridmap, different methods were assessed. After defining what the problem exactly was; Setting up the navigation areas for the robot in new, unknown areas is time consuming and requires user input, and why the solution for the problem is not straightforward, the requirements were formed. With these requirements and problems in mind the different methods were compared. The methods concerning recognizing different rooms were discarded since this method relies on recognizing rooms, not labeling areas. The method relying on using raycasting, to compare the scans of new areas to known scans, was discarded since this method relies on training the algorithm. After comparing the different methods, the choice was made to shift away from methods relying on machine learning and to create an algorithm which focuses on finding free space in the gridmap. Free space being available positions for the robot after a turn or when moving straightforward. By defining what rules apply for what area the free space method enabled finding these different navigation areas. The method relies on creating, and traveling through different branches, to find free space to its sides along the way.
The created algorithm provides insights into how free space is used to find semantic spaces in a gridmap. The method of using free space is proven to be useful for finding these areas, but because of the false turns inside of the corridors, there needs to be improvement on the set rules for defining the areas and the method of saving what coordinate/pixel belongs to what area. It is especially important to find a method that defines exactly, with coordinates that form a polygon, where which area is, instead of only finding the walls surrounding this area. The algorithm can also be improved by finding the relative size (relative to the gridmap) of the robot. If these problems are solved the method is robust without requiring training data, since the algorithm works with absolute rules, defined steps and areas. This also eliminates the risk of finding a situation that a machine-learning algorithm can not be trained for. Therefore it can be concluded that this method has advantages over other available methods.

References

[1] CBS. Arbeidsmarkt zorg en welzijn, cbs.nl/nl-nl/dossier/arbeidsmarkt-zorg-en-welzijn, 2021.

[2] Richard Bormann, Florian Jordan, Wenzhe Li, Joshua Hampp, and Martin Hägele. Room segmentation: Survey, implementation, and analysis. In *Proceedings - IEEE International Conference on Robotics and Automation*, volume 2016-June, pages 1019–1026, 2016.

[3] Óscar Martínez Mozos. *Semantic labeling of places with mobile robots*. PhD thesis, Albert-Ludwigs-Universität Freiburg, 2010.

[4] Balázs Kégl. The return of AdaBoost.MH: multi-class Hamming trees. 12 2013.

[5] Python. About, https://www.python.org/about/.

[6] OpenCV. OpenCV-Python, https://opencv.org/, 2021.

[7] OpenCV. OpenCV Function; pointPolygonTest(), https://docs.opencv.org/4.5.2/d3/dc0/group__imgproc__shape.html#

A Appendix

A.1 Code

A.1.1 Function dominantdirection

```
def dominantdirection(r_pos,map):
  """
  Get the dominant direction (longest unobstructed distance and its corresponding angle)
     for a given position in the gridmap
  """
  x = r_pos[0][0]
  y = r_pos[0][1]
  lengthperangle = {}
  length = 0
  for angle in range(0,360,1):
    while map[y][x] > 254 #Continue raycasting until a grey/black pixel is detected
      length += 1
      x = x + cos(angle)
      y = y + sin(angle)
    lengthperangle[angle] = length
  dominantdirection = max(lenghtperangle)
  return dominantdirection
```

A.1.2 Function getcorners

```
def getcorners(r_pos):
  """
  Returns position of the four corners with input the position (x,y) and angle of the
    robot.
  ---------
  """
  r_size = (45,30) #Define Robot Size
  robot = [None] * 4
  robot[0] = round( r_pos[0][0] - (r_size[0]/2 * cosd(r_pos[1])) - (r_size[1]/2 *sind(
    r_pos[1])) ), round( r_pos[0][1] - (r_size[0]/2 * sind(r_pos[1])) + (r_size[1]/2 *
    cosd(r_pos[1])) )
  robot[1] = round( r_pos[0][0] + (r_size[0]/2 * cosd(r_pos[1])) - (r_size[1]/2 *sind(
    r_pos[1])) ), round( r_pos[0][1] + (r_size[0]/2 * sind(r_pos[1])) + (r_size[1]/2 *
    cosd(r_pos[1])) )
  robot[2] = round( r_pos[0][0] + (r_size[0]/2 * cosd(r_pos[1])) + (r_size[1]/2 *sind(
    r_pos[1])) ), round( r_pos[0][1] + (r_size[0]/2 * sind(r_pos[1])) - (r_size[1]/2 *
    cosd(r_pos[1])) )
  robot[3] = round( r_pos[0][0] - (r_size[0]/2 * cosd(r_pos[1])) + (r_size[1]/2 *sind(
    r_pos[1])) ), round( r_pos[0][1] - (r_size[0]/2 * sind(r_pos[1])) - (r_size[1]/2 *
    cosd(r_pos[1])) )
  return robot
```

A.1.3 Function findwall

```
ef findwall(robot,map,contourspace):
  """
  Returns all coordinates of the walls in the contour that is assessed at this
  iteration
  """
  wallsincontour = []
  x_min,x_max,y_min,y_max = min_x(robot),max_x(robot),min_y(robot),max_y(robot)
  contours ,_ = cv.findContours(contourspace, cv.RETR_TREE, cv.CHAIN_APPROX_SIMPLE)

  for i in range(x_min,x_max,1):
    for j in range(y_min,y_max,1):
      if 0 < map[j][i] < 210:
        if cv.pointPolygonTest(contours[1], (i,j), False) == 1:
```

```
          wallsincontour.append(None)
        else:
          continue

      if map[j][i] == 0:
        if cv.pointPolygonTest(contours[1], (i,j), False) == 1:
          wallsincontour.append((i,j))
        else:
          continue

      else:
        continue
  return wallsincontour
```

A.1.4 Algorithm results visualisation

```

#Create empty space to visualize results
wallspace_weight = np.zeros((map.shape[0], map.shape[1]), dtype=np.uint8)

rank_coord = {} #Empty dictionary to keep track of number of occurrunces per coordinate

for coord in crossing_walls:
  if coord not in rank_coord and coord != None:
    rank_coord[coord] = 1
  elif coord in rank_coord and coord != None:
    rank_coord[coord] = rank_coord[coord] + 1

#Factor which must be scaled according to how long the algorithm had been running
rank_factor = 8

for key in rank_coord:
  if rank_coord[key] > rank_factor * 6:
    wallspace_weight[key[1]][key[0]] = 255 #White
  elif rank_coord[key] > rank_factor * 5:
    wallspace_weight[key[1]][key[0]] = 200
  elif rank_coord[key] > rank_factor * 4:
    wallspace_weight[key[1]][key[0]] = 150
  elif rank_coord[key] > rank_factor * 3:
    wallspace_weight[key[1]][key[0]] = 75
  elif rank_coord[key] > rank_factor * 2:
    wallspace_weight[key[1]][key[0]] = 50
  elif rank_coord[key] > rank_factor * 1:
    wallspace_weight[key[1]][key[0]] = 25 #Very dark grey

cv2_imshow(wallspace_weight)
```

A.1.5 Full Algorithm

```
import os
import random
import matplotlib.pyplot as mpl
from matplotlib import font_manager
from PIL import Image, ImageDraw, ImageFont
import numpy as np
import PIL as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.image import *
import matplotlib.pyplot as plt2
import math
from __future__ import print_function
from __future__ import division
from google.colab.patches import cv2_imshow
import cv2 as cv
```

```
17
18 from google.colab import drive
19 drive.mount('/content/drive')
20
21 # Define path
22 path = 'drive/MyDrive/Docs/BEP'
23 map_png = '/content/drive/MyDrive/Docs/BEP/map_scan3.png'
24
25 def convert2array(img: Image):
26     """
27     Convert an image to array
28     ---------
29     img : Image
30         Image to convert to array
31     Returns
32     -------
33     features : list/matrix/structure of float, float between zero and 255
34         Extracted features in a format that can be used in the image classifier.
35     """
36
37     # <ASSIGNMENT: Implement your feature extraction by converting pixel intensities to
       features.>
38     img = img.convert("L")  # converting image to greyscale (in case the image was color
       )
39
40     pix = np.array(img)  # converting image to array filled with array
41
42     return pix
43
44 def sind(angle):
45   sin_r = np.sin(np.deg2rad(angle))
46   return sin_r
47
48 def cosd(angle):
49   cos_r = np.cos(np.deg2rad(angle))
50   return cos_r
51
52 def min_x(robot):
53   x_min = min(robot)[0]
54   return x_min
55
56 def max_x(robot):
57   x_max = max(robot)[0]
58   return x_max
59
60 def min_y(robot):
61   y_min = min(robot, key=lambda x:x[1])[1]
62   return y_min
63
64 def max_y(robot):
65   y_max = max(robot, key=lambda x:x[1])[1]
66   return y_max
67
68 def getcorners(r_pos):
69   """"
70   Returns position of the four corners with input the position (x,y) and angle of the
       robot.
71   ---------
72   """
73   r_size = (50,36) #Define Robot Size
74   robot = [None] * 4
75   robot[0] = round( r_pos[0][0] - (r_size[0]/2 * cosd(r_pos[1])) - (r_size[1]/2 *sind(
       r_pos[1])) ), round( r_pos[0][1] - (r_size[0]/2 * sind(r_pos[1])) + (r_size[1]/2 *
       cosd(r_pos[1])) )
76   robot[1] = round( r_pos[0][0] + (r_size[0]/2 * cosd(r_pos[1])) - (r_size[1]/2 *sind(
       r_pos[1])) ), round( r_pos[0][1] + (r_size[0]/2 * sind(r_pos[1])) + (r_size[1]/2 *
       cosd(r_pos[1])) )
77   robot[2] = round( r_pos[0][0] + (r_size[0]/2 * cosd(r_pos[1])) + (r_size[1]/2 *sind(
       r_pos[1])) ), round( r_pos[0][1] + (r_size[0]/2 * sind(r_pos[1])) - (r_size[1]/2 *
       cosd(r_pos[1])) )
```

```
78    robot[3] = round( r_pos[0][0] - (r_size[0]/2 * cosd(r_pos[1])) + (r_size[1]/2 *sind(
        r_pos[1])) ), round( r_pos[0][1] - (r_size[0]/2 * sind(r_pos[1])) - (r_size[1]/2 *
        cosd(r_pos[1])) )
79    return robot
80
81
82  def findwall(robot,map,contourspace):
83    """
84    Returns all coordinates of the walls in the contour that is assessed at this
85    iteration
86    """
87    wallsincontour = []
88    x_min,x_max,y_min,y_max = min_x(robot),max_x(robot),min_y(robot),max_y(robot)
89    contours ,_ = cv.findContours(contourspace, cv.RETR_TREE, cv.CHAIN_APPROX_SIMPLE)
90
91    for i in range(x_min,x_max,1):
92      for j in range(y_min,y_max,1):
93        if 0 < map[j][i] < 210:
94          if cv.pointPolygonTest(contours[1], (i,j), False) == 1:
95            wallsincontour.append(None)
96          else:
97            continue
98
99        if map[j][i] == 0:
100         if cv.pointPolygonTest(contours[1], (i,j), False) == 1:
101           wallsincontour.append((i,j))
102         else:
103           continue
104
105       else:
106         continue
107   return wallsincontour
108
109 def turning(r_pos,turn,lengthfactor):
110   tur_rad = 40
111   r_testpos = [None]*2
112
113   AB = 2 * tur_rad * sind(0.5 * abs(turn))
114   curvelength = (turn/360)*2*math.pi*tur_rad
115
116   if turn < 0:
117     remaining_dist = lengthfactor * 1.25*math.pi*tur_rad + curvelength
118   elif turn >= 0:
119     remaining_dist = lengthfactor * 1.25*math.pi*tur_rad - curvelength
120
121   r_testpos[1] = r_pos[1] + turn
122   r_testpos[0] = (r_pos[0][0] + AB * cosd(-r_pos[1] - 0.5*turn) + (remaining_dist * cosd
        (r_testpos[1])) ), (r_pos[0][1] - AB * sind(-r_pos[1] - 0.5*turn) + (remaining_dist
        * sind(r_testpos[1])))
123
124   return r_testpos
125
126 map = convert2array(Image.open(map_png))
127 map_vis = convert2array(Image.open(map_png))
128 map_corners = convert2array(Image.open(map_png))
129 map_walls = convert2array(Image.open(map_png))
130
131 ### Create empty space to draw contour.
132 contourspace = np.zeros((map.shape[0], map.shape[1]), dtype=np.uint8)
133
134
135 ### Current Position
136 r_pos = [None] * 2
137 r_pos[0] = [558,187]
138 r_pos[1] = 10
139
140 robot = getcorners(r_pos)
141 for i in range(4):
142   cv.line(contourspace, robot[i], robot[(i+1)%4], ( 255 ), 2)
143   cv.line(map_vis, robot[i], robot[(i+1)%4], ( 0 ), 3)
```

```
144
145
146 tur_rad = 35
147 r_testpos = [None]*2
148
149 corridor_walls = []
150 crossing_walls = []
151 pos_available = [r_pos]
152 numoffullturns = 0
153 count = 1
154 while len(pos_available)>0:
155
156   r_pos = pos_available.pop(0)
157
158
159   ##RIGHT
160   possible_rturn = []
161   addrturn = True
162   wallsinrturn = []
163   for turn in range(30,130,10):
164     contourspace_btwn = np.zeros((map.shape[0], map.shape[1]), dtype=np.uint8)
165     contourspace_test = np.zeros((map.shape[0], map.shape[1]), dtype=np.uint8)
166     contourspace_ravg = np.zeros((map.shape[0], map.shape[1]), dtype=np.uint8)
167     r_rtestpos_btwn = turning(r_pos,turn,0.5)
168     test_rrobot_btwn = getcorners(r_rtestpos_btwn)
169     for i in range(4):
170       cv.line(contourspace_btwn, test_rrobot_btwn[i], test_rrobot_btwn[(i+1)%4], (255),
      1)
171     onwall_rbtwn = findwall(test_rrobot_btwn,map,contourspace_btwn)
172
173     r_rtestpos = turning(r_pos,turn,0.7)
174     test_rrobot = getcorners(r_rtestpos)
175     #Drawing it in the visual and contourspace
176     for i in range(4):
177       #cv.line(map_vis, test_rrobot[i], test_rrobot[(i+1)%4], (0), 1)
178       cv.line(contourspace_test, test_rrobot[i], test_rrobot[(i+1)%4], (255), 1)
179
180     #Check if tested position is on a wall, if not, add to possible turns
181
182     onwall_r = findwall(test_rrobot,map,contourspace_test)
183     wallsinrturn.extend(onwall_r)
184     if len(onwall_r) == 0 and addrturn is True and not onwall_rbtwn:
185       possible_rturn.append(turn)
186     elif len(onwall_r) != 0 and possible_rturn:
187       addrturn = False
188
189     #if there are possible turns, take the average and make that turn a new position.
190   if possible_rturn:
191     avgrturn = sum(possible_rturn) / len(possible_rturn)
192     r_rnewpos = turning(r_pos,avgrturn,0.7)
193
194     crossing_walls.extend(wallsinrturn)
195     crossing_walls.extend(wallsinlturn)
196
197     newpos_rrobot = getcorners(r_rnewpos)
198     for i in range(4):
199       cv.line(map_vis, newpos_rrobot[i], newpos_rrobot[(i+1)%4], (100), 1)
200       cv.line(contourspace_ravg, newpos_rrobot[i], newpos_rrobot[(i+1)%4], (255), 1)
201
202     onwall_ravg = findwall(newpos_rrobot,map,contourspace_ravg)
203
204     if len(onwall_ravg) == 0:
205       pos_available.append(r_rnewpos)
206     else:
207       cv.line(map_vis, newpos_rrobot[i], newpos_rrobot[(i+1)%4], (0), 5)
208
209   ##STRAIGHT
210   possible_str, wallsin_str = [] , []
211   addstr = True
212   strfac = 0.4
```

```
  if random.randint(0,100) > 95:
    r_pos[1] = r_pos[1] + 180
    numoffullturns = numoffullturns+ 1
  for turn in range(-30,30,10):
    contourspace_test = np.zeros((map.shape[0], map.shape[1]), dtype=np.uint8)
    contourspace_stravg = np.zeros((map.shape[0], map.shape[1]), dtype=np.uint8)

    r_strtestpos = turning(r_pos,turn,strfac)
    test_strrobot = getcorners(r_strtestpos)
    #Drawing it in the visual and contourspace
    for i in range(4):
      #cv.line(map_vis, test_strrobot[i], test_strrobot[(i+1)%4], (0), 1)
      cv.line(contourspace_test, test_strrobot[i], test_strrobot[(i+1)%4], (255), 1)

    #Check if tested position is on a wall, if not, add to possible turns
    onwall_str = findwall(test_strrobot,map,contourspace_test)
    wallsin_str.extend(onwall_str)
    if len(onwall_str) == 0 and addstr is True:
      possible_str.append(turn)
    elif len(onwall_str) != 0 and possible_str:
      addstrturn = False

    #if there are possible turns, take the average and make that turn a new position.
  if possible_str:
    avgstr = sum(possible_str) / len(possible_str)
    r_strnewpos = turning(r_pos,avgstr,strfac)

    newpos_strrobot = getcorners(r_strnewpos)
    for i in range(4):
      cv.line(map_vis, newpos_strrobot[i], newpos_strrobot[(i+1)%4], (0), 1)
      cv.line(contourspace_stravg, newpos_strrobot[i], newpos_strrobot[(i+1)%4], (255),
    1)

    onwall_stravg = findwall(newpos_strrobot,map,contourspace_stravg)

    if len(onwall_stravg) == 0:
      pos_available.append(r_strnewpos)
    else:
      cv.line(map_vis, newpos_strrobot[i], newpos_strrobot[(i+1)%4], (0), 1)

  ##LEFT
  possible_lturn = []
  wallsinlturn = []
  addlturn = True
  for turn in range(-130,-60,10):
    contourspace_btwn = np.zeros((map.shape[0], map.shape[1]), dtype=np.uint8)
    contourspace_test = np.zeros((map.shape[0], map.shape[1]), dtype=np.uint8)
    contourspace_lavg = np.zeros((map.shape[0], map.shape[1]), dtype=np.uint8)
    r_ltestpos_btwn = turning(r_pos,turn,0.5)
    test_lrobot_btwn = getcorners(r_ltestpos_btwn)
    for i in range(4):
      cv.line(contourspace_btwn, test_lrobot_btwn[i], test_lrobot_btwn[(i+1)%4], (255),
    1)
    onwall_lbtwn = findwall(test_lrobot_btwn,map,contourspace_btwn)

    r_ltestpos = turning(r_pos,turn,0.7)
    test_lrobot = getcorners(r_ltestpos)
    #Drawing it in the visual and contourspace
    for i in range(4):
      #cv.line(map_vis, test_lrobot[i], test_lrobot[(i+1)%4], (0), 1)
      cv.line(contourspace_test, test_lrobot[i], test_lrobot[(i+1)%4], (255), 1)

    #Check if tested position is on a wall, if not, add to possible turns
    onwall_l = findwall(test_lrobot,map,contourspace_test)
    wallsinlturn.extend(onwall_l)
    if len(onwall_l) == 0 and addlturn is True and not onwall_lbtwn:
      possible_lturn.append(turn)
    elif len(onwall_l) != 0 and possible_lturn:
      addlturn = False

```

```
    #if there are possible turns, take the average and make that turn a new position.
  if possible_lturn:
    avglturn = sum(possible_lturn) / len(possible_lturn)
    r_lnewpos = turning(r_pos,avglturn,0.7)

    crossing_walls.extend(wallsin_str)
    crossing_walls.extend(wallsinrturn)
    crossing_walls.extend(wallsinlturn)

    newpos_lrobot = getcorners(r_lnewpos)
    for i in range(4):
      cv.line(map_vis, newpos_lrobot[i], newpos_lrobot[(i+1)%4], (100), 1)
      cv.line(contourspace_lavg, newpos_lrobot[i], newpos_lrobot[(i+1)%4], (255), 1)

    onwall_lavg = findwall(newpos_lrobot,map,contourspace_lavg)

    if len(onwall_lavg) == 0:
      pos_available.append(r_lnewpos)
    else:
      cv.line(map_vis, newpos_lrobot[i], newpos_lrobot[(i+1)%4], (0), 1)
  count = count + 1

cv2_imshow(map_vis)


#Create empty space to visualize results
wallspace_weight = np.zeros((map.shape[0], map.shape[1]), dtype=np.uint8)

rank_coord = {} #Empty dictionary to keep track of number of occurrunces per coordinate

for coord in crossing_walls:
  if coord not in rank_coord and coord != None:
    rank_coord[coord] = 1
  elif coord in rank_coord and coord != None:
    rank_coord[coord] = rank_coord[coord] + 1

#Factor which must be scaled according to how long the algorithm had been running
rank_factor = 19

for key in rank_coord:
  if rank_coord[key] > rank_factor * 6:
    wallspace_weight[key[1]][key[0]] = 255 #White
  elif rank_coord[key] > rank_factor * 5:
    wallspace_weight[key[1]][key[0]] = 200
  elif rank_coord[key] > rank_factor * 4:
    wallspace_weight[key[1]][key[0]] = 150
  elif rank_coord[key] > rank_factor * 3:
    wallspace_weight[key[1]][key[0]] = 75
  elif rank_coord[key] > rank_factor * 2:
    wallspace_weight[key[1]][key[0]] = 50
  elif rank_coord[key] > rank_factor * 1:
    wallspace_weight[key[1]][key[0]] = 25 #Very dark grey

cv2_imshow(wallspace_weight)
```