Intelligent and Mobile Robotics - Final Project

Path Finder

Vedran Semenski

***Abstract* - This paper presents and describes an implementation of a solution (robotic agent) for the Path Finder competition. One of the main problems faced in this competition are mapping of the robots environment, calculating the shortest (return) path and developing behaviours that can accomplish these goals . The robot used is the same type of robot used for the 2nd project but without the use of the beacon sensor. This agent was developed for a project that is part of the "Intelligent and mobile robotics" course and is the last of three in total.**

I. INTRODUCTION

The goal of this project was to develop a robotic agent that could solve the challenge of the Path Finder competition. This includes developing a solution that can efficiently explore and map any labyrinth (given a size limit), find the target area, find the shortest path to a destination. These challenges provided a good opportunity to use all the knowledge gained during the theoretical classes and show that a set of complicated problems could be solved with limited time, simple and noisy sensors and a simple robot. This solution was developed using C in Genie text editor in a Linux Ubuntu operating system. This project is a continuation of the robotic agent developed for the 2nd project as the basic platform is expanded and further enhanced. All aspects of the development, behaviours, features, limitations and test results will be presented and explained in this document.

II. OBJECTIVES

The main objective is to use a DETI robot (same type as the ones used on the 2nd assignment) to solve a labyrinth defined in an area of 3,6x3,6m; the areas of departure and arrival are marked in black (on a white background). The labyrinth is built on a cell grid of 45cmx45cm. The robot must explore the labyrinth until it finds the area of arrival, and then return to the departure through the shortest path, using the knowledge acquired in the exploration phase. No beacons are used.

Additionally keeping in mind good programming and development principles there is always the challenge to develop a solution that works but is also well structure, reusable, well documented and easily understandable. Considering that this is also an embedded system and all of the additional challenges and problems that come with this fact, all of the previously mentioned additional goals are even more important.

III. ROBOTIC AGENT

In this case particular case the robotic agent is an autonomous piece of software that interacts with the outside world by reading sensor readings and controlling motors that enable the robot to move.

The solution communicates with the robot via a library provided by the project supervisors. After compiling the solution with the provided compiler ("*pcompile*") the generated hex file is uploaded to the robot using a USB cable and the robot is ready for testing.

Usage of the libraries functions is abstracted and separated into 2 basic parts. In the main function the initialisation calls are made. In the *sensors.c* file all the communication regarding sensors and sensor data is handled and all sensor data is stored in custom structures. Basic movements regarding the left and right motors are handled in the *movoments.c* file.

The rest of the main functionality regarding the robotic agent are in 2 other files: the *behaviours.c* file which contains all the behaviours of the robotic agent and *map.c* file which contains all the functionality regarding the internal mapping of the robots environment.

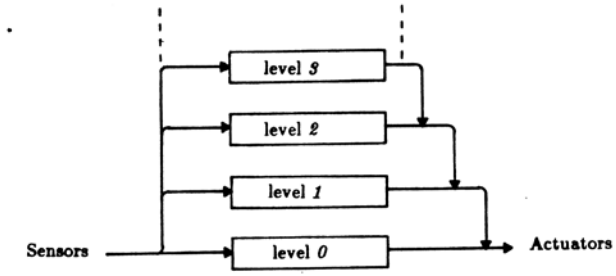
Additionally there is a library used to for the compass sensor (compass library) that was added to the robotic agent by professor Azevedo, a *dijkstra.c* file that contains Dijkstra's shortest path algorithm used by functions in *map.c* to calculate the shortest path to a destination and a *helper:functions.c* file containing a range of generic and useful functions continuously called by others mostly for angel related calculations and conversions.

This kind of separation of functionality and abstractions of the communication with the actual robot via the provided library has been done so the solution could be modular, meaning that changing a library for communicating with the robot (like sensor functions of movement functions) doesn't necessarily mean making a lot of changes in the agent. Usage of library functions are contained in a few functions each one used is called from one place so migration to another robot (taking in account it has all or most of the capabilities of this robot) would be as easy and straightforward.

IV. SUBSUMTION ARCHITECTURE

The subsumtion architecture implies two main things. The first one is that all of the agents behaviours should be organized in such a way that every behaviour has a certain set of conditions that have to be satisfied if the behaviour is to be executed. The second one is that all of the behaviours should be organized in a prioritized list manner and the behaviour that is executed must be the one with the highest priority from a set of behaviours which have satisfied their set of behaviours. [2]

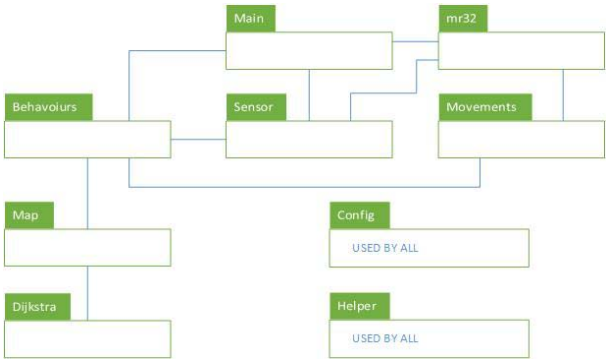
This architecture is implemented by organizing behaviours in separate functions inside a separate *"behaviours.c"* script. An array of behaviour IDs is organized in a prioritized manner. Each behaviour is first tested with a "*test_behaviour_name*" function and the behaviour with the highest priority is than executed. It should also be mentioned because of easier implementation a priority value of 0 is the highest priority and priority value of 5 is lower in priority than 4.3.2 etc. so a bit contra intuitive at first. The solution is written in such a way so it is easily reconfigurable and enables the programmer to easily modify the way behaviours are tested and executed. One benefit of this layout is exploited in a way that the behaviours aren't always prioritized in the same way and not all behaviours are available for testing and execution at all times. This way of executing behaviours is completely in compliance with the subsumtion architecture although some behaviours have internal states and can be viewed as having sub-behaviours that progress from one to another like mini state machines. These behaviours are more complex and need this kind of functionality or sub-division to function properly. Although this working principle is contrary to the subsumtion architecture and some aspects of enabling the behaviours more as state machine globally the robotic agent still maintains the subsumtion architecture. All of this will be explained further in this document. The implementation of this architecture was already done and successfully used on the 2nd project and as such was used with minimal modifications. The main modifications in this area were done regarding building additional behaviours, refactoring code and making improvements.



Picture 1. Subsumtion architecture

V. SOLUTION CONCEPT

As the robots functionality is separated into separate files that are organized in such a way that is similar to classes in object oriented languages this solution architecture and its dependencies can be better described with a simple class diagram.



Picture 2. File diagram / class diagram

The layout of the modules is more or less exactly the same as the layout and dependencies of the files which is because the files were organized a that that would accommodate the architecture and layout of the modules. This was done keeping the principles of objective design in mind to get modular, quality piece of software and a complete product in the end.

The main workflow of the robotic agent consists of the following. Firstly everything is initialized with a call from the main function. After initialisation has finished the behaviour module is called every cycle. The behaviour module requests sensor readings from the sensor module every time it is called and runs test functions for all behaviours and executes the behaviour which can has the highest priority among those that can be executed. Behaviours call the movement module to move the robot in the desired way and call the map module to build a map of the environment and to retrieve shortest paths to destination fields. The sensor module reads sensor data using the provided library (mr32) and provides filtered and normalized data to anyone who requests it. The map module contains functionality for storing and maintaining a map and functionality for retrieving shortest paths and outputting the map to the standard output. An additional helper module contains useful functions that are used by all modules. The functions it contains are mainly for operations regarding angle calculations and some additional functions for copying arrays and others.

VI. SENSOR MODULE AND MOVEMENTS

The sensor and movements modules are the only ones that are connected and use functions from the provided library. This provides an option for exchanging libraries without much interference in the code itself.

The sensor module provides many functionalities for providing more usable and accurate sensor readings. All of the values that the sensor module reads are normalized ,stored in a circular buffer and using the buffer the readings are filtered. The buffer size is currently set to 5 but changing it to another value is simple and requires only the changing of one defined parameter. The buffer is filled on a circular principle which means that newer values get stored and older ones are replaced. Filtering of the readings is done differently for different readings. The filtration method used for all sensor readings accept the obstacle sensors is a weighted average method. The average is measured from all readings contained in the buffer but the most recent readings a have a more significant impact. For example the oldest reading in the buffer will be multiplied by the coefficient of 1 and the newest will have a coefficient of 5 which means it has 5 times the influence of the oldest reading. Angle readings proved to be challenging because averaging of angles doesn't necessarily mean the averaging of the angle values (the angle between 5° and 355° is 0° but an average gives 180°). Because of this, the x and y values were taken using sin() and cos() values and the angle was retrieved by calculating the value of the end point. For the obstacle sensors, the readings were filtered using a median filter. It is implemented just by ordering the buffer values in a ascending list and the middle value (or the average of two middle values if the buffer size is an even number) is taken as the referent value.

The normalisation of values is simple for most readings and doesn't need much explanation. The obstacle sensors are the ones that proved to be the most noisy and most difficult to get reliable and accurate readings from. The formula used is the one provided in the document containing the datasheets for the DETI robot.

Formula:

$$distance = 6200/(ValADC-80)$$

Additionally there are high and low limits put in place so when the sensor gets noise or bad readings from time to time, the impact of that value is contained within some beaneries. these boundaries mean that readings that represent a distance value that is more than 70 cm or less than 10 cm they are limited to those boundaries.

Additional functionality provides the sensor module to get offset values for the sensor readings that could represent accumulation errors which could be detected by another part of the robotic agent. These offset values are added to the original sensor readings and as such give flexibility to this module. This is used for the position values because they tend to accumulate the error over time because of the odometry principle they are calculated from. The behaviour module detect the error distance and reports it to the sensor module and the sensor module adjusts all future reading accordingly.

The movements module contains a lot of functions for many types of movements. Main movements are used for rotating, going in a straight line, going in a curve line... All end movement is filtered through one function in which a primitive buffer is implemented. This buffer doesn't allow the robot to speed up or slow down the motors rapidly and instead allows the speed to change incrementally with a maximum limit of 5. An additional condition is set so if for instance one motor wants to increment its speed a large amount and the other jus a tiny amount the function lets one side catch up to the other side and that way the resulting movement is more natural, it flows much smoother, strange behaviours are avoided and the accumulation of error on the odometry position sensors are minimized.

VII. Mapping

The mapping module contains all of the functionality needed to build , maintain and retrieve information from the robots environment. It is built in such a way that allow it to be easily changed and tweaked to suit the current purpose it needs to fulfil. The main building blocks are fields and connections. These are equivalent to nodes and vertices in standard graph terminology. Fields have an array of connections, a real world position on the map and a state. The number of connections that the fields have can be changed and increased for more accurate and full map just changing one defined constant but in this implementation the number is set to 4. These connections represent connections to other fields and in this case 4 directions (0°, 90°, 180° and 270°). The connections contain pointers to a field and also a state. The value of a connection's state and a field's state can be: FREE, UNDEFINED or OCCUPIED. This state represent the information the robot has learned (of UNDEFINED if it hasn't) about that particular field or connection. The fields state represents the state of that filed or gives us information if the robot can go there or not and the connection's state represents that the robot cannot move in that direction. Upon adding a new field the map module checks if a field already exists and just updates the map accordingly. The map is stored as a simple array of fields. Because of the robots memory limitations the number is limited to 128 points.

For retrieving the shortest path Dijkstra's algorithm is used. Although the A* algorithm would be more efficient for calculating the final shortest path for returning to the starting point Dijkstra's algorithm can be used to find the shortest path to unexplored fields and because of that it was chosen over the A*. The labyrinths that the robot is intended to solve have approximately 25 or a maximum of 64 fields which is almost instant to compute with Dijkstra's algorithm. If future use needs a different algorithm it can easily be changed just by adding it and modifying the call to that algorithm. The call is done by providing a weight/connection array to the algorithm which returns the list of shortest distances and array of "previous" fields for each one which is then used to reconstruct the wanted shortest path. The last feature of the map is outputting it to the standard output using printf and drawing it in lines using a list of characters with meanings: S(Start), F(Finish), O(free field), X(occupied field), - (horizontal free connection), | (free connection), x(occupied connection), ?(undefined/unexplored/unknown connection).

Example of output:

```
        x
      xOx
   x  x  |
  ?S-O-OxX
   ?  x |
     ?F?
      ?
```

VIII. Behaviours

Every behaviour is associated with its test, reset and execution function. All of them can be found in the *behaviour.c* file. The behaviours are organized in a prioritized list but that list isn't static. The priority of the behaviours changes depending on the situation the robotic agent finds itself.

List of behaviours:

- AvoidColision - "Obstacle avoidance"
- StopAtBeaconArea - "Stop at the beacon area"
- StopAtStartingPosition - "Stops when the robot is at the starting position"
- FollowPath - "Moves along a given path"
- ReturnHome - "Returns the robot to the starting position field"
- ExploreLabyrinth - "Explores and maps the robots surroundings"
- CorrectPosition - "Corrects the robots position and centres it towards the middle of the field"

AvoidColision behaviour activates when the robot comes very close to an obstacle. The actions consists of the robot stopping and rotating itself around its axis in the direction of the side that is further away from the obstacles. If the right obstacle distance sensor shows that there is an obstacle closer to that side compared to the left sensor the robot will turn left and vice versa.

StopAtBeaconArea activates when the ground sensor shows that the robot arrived to the beacon area. The behaviour just consists of the robot stopping.

StopAtStartingPosition activates when position of the robot is approximately equal to its starting position. The behaviour just consists of the robot stopping.

FollowPath activates when a path is given to it and remains active until the destination is reached. The behaviour consists of moving from point to point until the robot reached the destination. This behaviour also has an additional feature in that it detects if the given path has long straights meaning if there are multiple consecutive points laying on the same line it skips the points in the middle and travels from one end the other.

ReturnHome behaviour has no conditions that need to be satisfied for the behaviour to activate so it can always be called and it consists of setting the return path and thus enabling the *FollowPath* behaviour to activate and return the robotic agent to the starting position.

CorrectPosition activates when the robot detects an accumulation error on the position sensor that works off of the stepper motor. It detects the error when the sensor readings indicate that it is to close or to far form a wall. Because it moves from cell to cell it should always be cantered in the cell and the distance between it and a wall that is blocking a path should always be the same. This is checked periodically and sent to the sensor module (*sensor.c* file) which than stores the accumulated error and

takes it in consideration when returning readings to the behaviours module. When the error is greater than 5cm the behaviour is activated. The behaviour consists of moving the robot to the current position it should be in. Because the error is sent to the sensor module. It than compensated for the error and the position is no longer correct.

ExploreLabyrinth behaviour also has no condition for its activation meaning it can be activated at all times. The behaviour is the most complex one out of all other behaviours and can be viewed as a set of smaller sub-behaviours that change like a state machine. The behaviour has an internal state which it maintains and according to that state a sub behaviour is chosen.

List of the behaviours sub-behaviours (internal states):

- *Moving to the next point*
- *Finding next unexplored field*
- *Discovering and adding new field*

Moving to the next point is the initial state and the behaviour consists of the robot moving to the field that is marked as the next point. During this movement (about $2/5^{s}$ of the way) it checks the left and right sensor readings and determines if the left or right neighbours of the next field are occupied. It adds them if this condition is satisfied. Once it reaches the destination field it changes the state to the next one (*Finding next unexplored field*).

Finding next unexplored field goes through the list of neighbours and checks if there is an unexplored/undefined connection and advances to the next behaviour in case it finds it. If there are no unexplored connections in the current field it calls a function to calculate and set the path to the nearest field with an unexplored connection and isn't activated until the *FollowPath* behaviour finishes. After it finishes it finds the unexplored field and continues to the next state/behaviour (*Discovering and adding new field*).

Discovering and adding new field behaviour rotates to the unexplored direction and takes accurate sensor readings (refreshes the buffer) . With those readings it determines if the neighbouring field is free or occupied. if the field is free it sets it as the next field and continues on to the first state (*Moving to the next*). In case the field is occupied it calls a function to check if the robot is cantered in the field which internally activates the *CorrectPosition* behaviour and sets the state to the second one (*Finding next unexplored*).

These behaviours provide the robotic agent with a lot of options and enable it to accomplish all of the tasks it needs to do. The behaviour priority list is changed from the initial one for the return journey which can be seen the lists below.

Initial (default) priority list for finding the beacon area:

1. StopAtBeaconArea
2. AvoidColision
3. CorrectPosition
4. FollowPath
5. ExploreLabyrinth

Return priority list for returning to the starting point:

1. StopAtStartingPointArea
2. AvoidColision
3. CorrectPosition
4. FollowPath
5. ReturnHome

All of these behaviours result in a behaviour that simply makes the robot move from field to field, stop and search for unexplored areas. It always moves in straight lines and only parallel to the x and y coordinates. It rotates 90 or 180 degree angles and it detects accumulation of error on the x and y position readings which it reports to the sensor module. Once it finds the beacon area it changes the priority list and returns home with the shortest path. It also checks for error accumulation during the way and once it returns to the starting field it outputs the map to the standard output.

IX. FEATURE LIST

Along with solving the main challenges that this project required there is a significant number of additional features that help the final solution to become more reliable, advanced and minimize the impact of sensor noise, error accumulation...

The sensor readings provided by the robots sensors have a significant amount of noise and the data that they provide isn't always intuitive to interpret and as such give problems during tuning phases of developing this agent. The main problems come from the 3 IR obstacle sensors. The readings that they provide are not good for interpreting distance and because of this they have to be normalized to be converted roughly into centimetres.

The movements of the robot are buffered and result in a smaller accumulation of error. The behaviours check the robots position when it reaches an obstacle and correct it's position accordingly to centre the robot in the middle of the field by reporting the error to the sensor module.

All of the coding and the final software product is well documented using Doxygen style documentation, it complies with the MISRA C coding standards. It was structured and built using objective oriented design principles and other good programming practices. The resulting code is modular, understandable and not prone to errors or bugs.

X. Future work

Further development of this project would be in the development of the map module. The module could easily be modified to convert the current point positions into fields that contain multiple points inside of them. This could reduce memory consumption and make the movement across mapped out open areas more efficient.

The biggest improvement would come with more accurate (and less noisy) obstacle sensors and a position measuring system that wouldn't accumulate errors along the way. Different algorithms can be used and tested for different search goals and the robot could be configured to search out larger areas. One of the main areas that could benefit with these changes is speed. Currently the robot cannot move much faster than it does because it accumulates error on the position readings and it needs to stop and refresh the readings buffer to get reliable readings before it determines if a field is occupied or free.

XI. Development difficulties, choices and testing

The c environment in embedded systems is very limited in it's very nature. A lot of work goes in and a lot of time is spent doing things that could be done much faster in an object oriented language like Java or C#. The problem of not being to use real recursion and malloc is also limiting and has to be taken in account during development. The testing of the robot if also time consuming and a lot of tests need to be done to get satisfying feedback and useful information. The choice to make the robot move from point to point, stop and take readings was made because of testing results. The results indicated that the sensor readings aren't reliable enough to keep the robot moving at all times. This has the consequence of the robot being relatively slow but was found that it s the only way that it can reliably work. Memory issues with the robot limit the robot to about 128 points on the field.

A compass sensor was installed and implemented on the robot agent but isn't used in the final solution. Even if the readings are precise and accurate the problem comes from the fact that the position of the robot is calculated using odometry and there is no way to internally set and every time consider the offset of the error. The sensor module even hat it implemented and it can get the offset but the odometry readings rely on the angle it has and as such cannot be changed.

While in development the robot initially mapped the labyrinth using smaller cells so it could function in various types of mazes. This is still an option and can be easily changed and added but the final solution performed better using the whole cell as the used field.

XIII. Conclusion

The subsumtion architecture provides a very flexible platform for expanding the behaviour library and further developing and improving the agent. It also enables easy tuning, especially with a large library of behaviours. The reactive aspect of the agent simplifies the implementation but proves to result in a very satisfying behaviour of the agent. The second benefit of the reactive aspect is that the agent does not waste time on complex calculations and as a result minimizes the time from receiving data from the sensors and the time of executing the behaviours.

This project provided a very good platform to test and implement skills learned on this course. It incorporates a lot of non trivial challenges and is a kind of a proving ground to present all of the skills learned on the course.

Above these challenges, the challenge to make a reliable, modular and quality solution that not only works but can be understood by others and improved upon is always present. Combining principle learned also on other courses during my study is always a goal and a challenge onto itself and which in my opinion were satisfied in most part. This project although simple to initially plan and make a rough sketch proved to be a real challenge and the result although never perfect is satisfying.

XIV. References

[1] -" Final Projects
Robótica Móvel e Inteligente 2014/2015- Project 7 - PathFinder", http://elearning.ua.pt/pluginfile.php/48275/mod_resource/content/3/Lista_trabalhos.pdf, January 2015.